Software Engineering

Saros

Verteiltes Debugging

Umut Pir Erdogan

Matr. 3417590

Betreuer: Prof. Dr. Lutz Prechelt
Betreuender Assistent: Dr. Karl Beecher

Institut für Informatik, Freie Universität Berlin, Deutschland

22. November 2010

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, sind als solche gekennzeichnet. Die Zeichnungen oder Abbildungen sind von mir selbst erstellt worden oder mit entsprechenden Quellennachweisen versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner Prüfungsbehörde eingereicht worden.

Berlin, den 22. November 2010

(Umut Pir Erdogan)

Zusammenfassung

Abriss

In dieser Arbeit soll die Saros Paar-Programmierungsanwendung um die Funktionalität eines Debuggers erweitert werden. Anhand des Java Debuggers für Eclipse soll die Machbarkeit eines solchen Vorhabens ermittelt und die dabei entstandenen Probleme und Lösungsstrategien aufgezeigt werden. Das Ergebnis soll ein in möglichst vielen Aspekten zum Single-User Debugger gleichwertiger Multi-User Debugger in Eclipse sein. Das langfristige Ziel dieses Vorhabens ist es, ein Programm im Team debuggen und damit Nebenläufigkeitsaspekte untersuchen zu können.

Danksagung

Diese Arbeit widme ich meiner Mutter, die mir in der langen Zeit immer zur Seite gestanden hat und die ich von ganzem Herzen liebe. Außerdem möchte ich meinen Brüdern danken, die mich ständig motiviert und zeitweise auch gestresst haben.

Ferner möchte ich Prof. Dr. Lutz Prechelt für das mir entgegengebrachte Verständnis und meinem Betreuer Dr. Karl Beecher für seine Unterstützung und Betreuung danken.

Nicht zuletzt geht mein Dank dem Saros Team, dass mich immer tatkräftig unterstützt hat.

Inhaltsverzeichnis

KAPITEL 1

Einleitung

1.1 Motivation

Die Fehlersuche in einem Programm ist meist mit viel Aufwand verbunden und es gibt verschiedene Techniken die man dabei einsetzen kann.

- Einfügen von Ausgabeanweisungen bzgl. Variablenwerten oder als Mittel zur Codeverfolgung (Tracing)
- Benutzen eines Loggingframeworks wie, z.B. log4java von Apache
- Schreiben von Tests nach dem Muster des JUnit Frameworks, welche einen programmgestuerten Abgleich von Eingabe zu erwarteten Ausgabe durchführen; sogenannte Black-box Tests.
- Codereview durch erklären, betrachten und analysieren des Quellcodes durch mehrere Programmierer.
- Benutzen von Assertions, welche in der Testphase des Programmes aktiviert werden können und erwartete Nebenbedingungen ausdrücken, die bei Korrektheit der Codeausführung gegeben sein sollen. Diese lassen sich in der späteren Deployphase auf Compiletimeebene wieder abschalten und haben dadurch keinen negativen Einfluß auf die Ausführungsgeschwindigkeit.

Es gibt Fehler, welche zur Kompilierzeit vorhanden sind (Errors und Warnings) und Fehler die zur Laufzeit entstehen. Zu Letzteren zählen z.B. Exceptions und Runtimeerrors. Daneben gibt es aber auch Fehler, die nur unregelmäßig und nichtdeterministisch auftauchen. Diese Fehler resultieren zum größten Teil aus Nebenläufigkeitsaspekten von Programmen. Zur Lokalisierung dieser Art von Fehlern bedient man sich sogenannter Debugger. Mit Ihnen kann man ein Programm schrittweise abarbeiten lassen oder aber durch Haltepunkte[1] kritische Abschnitte näher untersuchen. Die in modernen Programmen gerne eingesetzte Nebenläufigkeit von Threads ist von nichtdeterministischer Natur, so daß es Konfliktsituationen geben kann, die nur selten beobachtet werden können. Aus diesem Umstand ensteht der Wunsch, verschiedene Situation dieser parallelität in Szenarien deterministisch durchspielen zu können. Dazu wird statt des Prozessors als Schedulingautorität der Benutzer des Debuggers in diese Rolle versetzt. Dieser kann nun zu einer Zeit immer nur einen

[1] Breakpoints

Thread per Einzelschritt steuern. In diesem Sinn kann er eine zeitscheibenbasierte parallele Threadausführung emulieren. Möchte man jedoch echte Parallelität von Threads, wie bei Mehr-Kern CPU möglich, nachspielen, benötigt man einen oder mehrere weitere Benutzer für den Debugger. Dieser müßte dazu multi-user-fähig sein. Damit die Benutzer parallel diesem Debugger Befehle geben können, bedarf es eines Ein-/Ausgabe-Multiplexings der Debuggeranwendungslogik mit seinen Benutzerschichten. Jeder Benutzer betrachtet denselben Debugger über eine eigene Benutzerschnittstelle. Diese Benutzerschnittstellen können sich auf verschiedene Prozessoren verteilen und kommunizieren ihre Ein-/Ausgaben mit der Debuggeranwendungslogik. Dadurch wird der Multi-User Debugger zu einem verteilten Multi-User Debugger klassifiziert.

1.2 Ziel

Es soll der Java Debugger der Eclipse Entwicklungsumgebung Multi-User fähig gemacht werden und als solcher in einem verteilten Benutzerszenario genutzt werden. Ziel soll es sein, einer Gruppe von Entwicklern beim gemeinsamen Debuggen ein Tool in die Hand zu geben, welches Mehr-Benutzer Steuerung eines Debuggers explizit unterstützt und somit Szenarien der Nebenläufigkeit deterministisch untersuchen läßt. Hier kann man z.B. Producer-Consumer Situation nachspielen oder aber Deadlock Situation erproben und Fehlertoleranz von Programmen validieren. Beispiele zum Einsatzzweck lassen sich viele finden. Weiter soll sich der neue Debugger nahtlos in das bestehende Saros Plug-in integrieren und sich dessen Infrastruktur bedienen. Somit sollte es ein neues Feature des Teamkollaborationstools Saros werden und dessen Einsatzspektrum dadurch erweitern.

1.3 Verwandte Arbeiten

1.3.1 Rendezvous Architecture

Die Rendezvous Architektur ist ein Framework zur Erstellung von Mult-User Anwendungen, welche sog. „conversational probs“ aus dem Umfeld von Gruppenmeetings in elektronischer Form repräsentieren sollen. Entsprechend dieser Zielsetzung ergeben sich bestimmte Anforderungen an solche Anwendungen und an das Framework, das deren Erstellung unterstützen soll. Die aktuelle Implementierung des Frameworks sieht eine zentralistische Architektur [7, S. 100]vor, bei der es einen „shared abstraction“ genannten dialogunabhängigen Applikationskern gibt der mit mehreren untereinander unabhängigen Benutzerschnittstellen bedient wird. Dabei befinden sich all diese Bestandteile der Anwendung auf einem Prozessor und in einem Adreßraum. Die Benutzerschnittstellen werden per deklarativer Grafikbibliothek als Bäume modeliert und stellen ihrerseits ein Datenmodell dar, welches mit dem abstrakten Grafiksystem X-Window gerendert und im Netzwerk an Terminals der zugehörigen Benutzer ausgesandt wird. Events, welche die Benutzerschnittstelle generiert, beziehen sich hierbei auf die Objekte der deklarativen Grafikbibliothek. Also wird durch dieses Vorgehen eine zusätzliche Abstraktion neben dem X-Window System auf Frameworkseite geschaffen. Problematisch an dieser Art der zentralisierten Realisierung ist die fehlende Skalierbarkeit bei steigender Benutzerzahl, da sich alle Benutzer einen Prozessor und einen Hauptspeicher teilen müßen. Von den Entwicklern wird dieser Umstand realisiert und ausgesagt, daß es prinzipiell kein Problem sei Ihr Framework auf eine verteilte Basis zu stellen, in der die UI’es auf verschiedenen Prozessoren lokal ausgeführt würden und mit dem zentralen Applikationskern kommunizieren könnten. Das

Framework leitet sich aus dem Prinzip des MVC Muster ab, wobei View und Controller zusammengfasst sind in der Dialogschicht und der Apllikationskern dem Model enstpricht. Im Gegensatz zu MVC gibt es aber noch sogenannte Link Objekte, welche die Konsistenz der verschiedenen Benutzerdialoge untereinander durch Synchronisierung mit dem Allplikationsobjekt vollautomatisch Regeln. Dazu definiert der Programmierer sogenannte Regeln, welche lokale Kopien von Objekten aus dem Applikationskern, die im Benutzerinterface angezeigt und manipuliert werden können, miteinander verknüpft und einen automatischen Abgleich bei Änderung durchführt. Im MVC Muster man dies als Programmierer selbst überwachen und über Callback Funktionen realisieren. Das defineren der Regeln erfolgt in der Sprache Common Lisp und CLOS. Aus dem Vorhandensein dieser drei Objektklassen der Rendezvous Architektur: Abstract, Link und View ergibt sich auch deren alternativer Name als ALV Paradigma.

1.3.2 Multi-User Suite

Von der Purdue Universität stammt die Entwicklung eines allgemeinen Frameworks zur Implementierung von Mehr-Benutzer Benutzerschnittstellen. Das Framework bietet die Möglichkeit bestehende Single-User Benutzerprogramme für den Mehr-Benutzerbetrieb weiterzuverwenden. Dabei müssen diese Single-User Programme das zuvor entwickelte Single-User Suite verwendet haben. Die Grundlegende Strategie an diesem Ansatz ist, das die Single-User Applikation als ein persistentes Objekt in einem Netzwerkdateisystem für alle Benutzer zugreifbar ist. Die einzelnen Benutzer besitzen sogenannte Dialog Manager, welche mit dem persistenten Applikationskern kommunizieren können. In diesem Sinn entspricht das Framework einem Hybrid aus zentralisierter Datenhaltung und verteilter replizierter Benutzerschnittstelle. Bei Eingaben kommt es zu Ad-hoc Sperrungen bzgl der Variablen in der Applikation, wann immer ein Benutzer über seine grafische Schnittstelle diese beginnt. Diese Sperren sind exklusiv und werden automatisch wieder freigegeben, sobald das Eingabefeld mit Return bzw. Focuswechsel bestätigt wird. Dieses Konzept ist allgemein von Datenbanksystem her bekannt, wo auch Änderungen in einem Transaktionsprozeß erfolgen und erst dann für andere Benutzer sichtbar werden [3, S. 31].
Einige der charakteristischen Merkmale sind:

- Die Architektur ist Kombination aus zentralisiertem und repliziertem Ansatz, da es den Applikationskern als zentrales Objekt („application object“) verwaltet und die Benutzerschnittstellen („dialogue managers“) in eigenen Betriebsystemprozeßen verteilt ausführen läßt [3, S. 13].
- Applikationskern ist kein normales Objekt im Hauptspeicher, sondern wird als persistentes Objekt ins Dateisystem instantiiert.
- Editorbasiertes Dialogsystem, welches nur einfache grafische Benutzerschnittstellen ermöglicht
- Suite MU Architektur führt Berechnungsfunktionen im Applikationsobjekt zentral aus und bedingt dadurch eine Verzögerung bzgl. des Erhalts des Ergebnisses [3, S. 13]
- Änderungen durch Benutzer sind nicht sofort, sondern erst bei Eingabebestätigung (commit) bei anderen Benutzern sichtbar.
- Voreingestelltes Kollaborationstransparentes Schema, welches mittels vorhandener Primitive stufenweise wieder aufgehoben werden kann. Prinzip der „incremental collaboration awareness“. [3, S. 3]

1.3.3 Saros

Saros ist ein Kollaborationswerkzeug für Programmierer, das sich als Plug-in in die Entwicklungsumgebung Eclipse integriert. Es wurde von Riad Djemili [4] im Jahr 2006 an der Freien Universität Berlin im Rahmen einer Diplomarbeit entwickelt und von weiteren Studenten um neue Fähigkeiten erweitert [14]. Charakteristisch für Saros ist seine Echtzeitfähigkeit von Textmanipulationen, welche sich auszeichnet durch die Eigenschaft, daß sich lokale Textmanipulationen sofort auf das lokale Dokument auswirken und dann in einem nachgelagerten Prozeß hinsichtlich Konsistenzerhaltung geeignet nachverarbeitet werden [18]. Die einzelnen Benutzer erhalten dadurch eine nahezu Verzögerungsfreie Programmierumgebung und über einen enstprechenden Algorithmus wird durch eine zentrale Instanz, den sogenannten Host, die Konsistenz der Textdokumente sichergestellt. Saros besitzt eine replizierte Architektur und besitzt ähnlich zu Suite ein editorbasiertes Änderungsschema. Anders zu diesem hat es aber nicht die Einschränkung, das Änderungen erst nach impliziter Eingabebestätigung (commit) übertragen und kommuniziert werden, sondern real-time erfolgen. Dabei kann es natürlich zu Konflikten wegen der dabei zu berücksichtigenden Nebenläufigkeiten von Benutzereingaben kommen. Um diesen Konflikten zu begegnen gibt es eine zentrale Programminstanz, welcher die Eingaben synchronisiert und das Ergebnis anderen mitgliedern zur Nachfolziehung mitteilt. Also kommunizieren alle Benutzer nicht direkt untereinander, sonder werden durch ein zentrale Programminstanz, den sog. Host, vermittelt. Die Synchronisation der Eingaben geschieht durch eine Transformation dieser mittels Jupiter-Algorithmus. Peers (zu erklären) führen also Änderungen zunächst lokal aus und schicken diese dann an den Host, der sie in einer seriellen von der Eingangsreihenfolge abhängigen Reihenfolge auf einem zentralen Projekt (Dokument) ausführt. Da nun alle Peers zuerst ihre lokalen Änderungen ausführen und dann erst mitteilen, muß bei parallen Eingaben am selben Dokument der Host für jeden Peer eine transformation der Fremdeingaben der anderen Peers durchführen, so das diese transformierten Eingaben nach der Anwednung auf den jeweiligen Peers alle zum gleichen Dokumentenzustand führen wie das zentrale Dokument beim Host.

1.4 Aufbau der Arbeit

Im ersten Kapitel beschreibe ich die Strukturen und Anforderungen, die nötig sind, um sich der Problemstellung zur Implementierung eines Multi-User Debuggers bewußt zu werden. Zuerst erkläre ich die unterschiede zur zentraliserten und replizierten Datenhaltung, anschließend Stelle ich Merkmale von Debuggern und ihre Arbeitsweise dar und beende diese Beschreibung am Beispiel von Java mit der Darstellung der Java Debug Platform Architecture (JPDA). In einem Zwischenschritt gebe ich ein kleines Tutorial für Eclipse, welches die Konzepte Extension-Points, Extensions und Adapter beschreibt. Anschließend stelle ich das generische Debuggingframework in Eclipse vor. Am Beispiel des Java Debuggers greife ich die JPDA-Architektur wieder auf und zeige wie diese mit dem Eclipse Debuggingframework zusammenarbeit. Zum Abschluß dieses Kapitels erfolgt eine Analyse der Implementierungsmöglichkeiten zur Realisierung des Multi-User Debuggers und eine entsprechende Entscheidungsfindung.

Das nachfolgende Kapitel widmet sich schließlich der eigentlichen Implementierung. Es beschreibt die Anforderungen an das zu erstellende Programm in einem Lastenheft und behandelt Fragen zu möglichen Konflikten, zur Serialisierung, Konsistenz und zum repliziertes Starten des Debuggers. Dabei werden verwandte Konzepte in der Implementierung von verteilten Applikationen beschrie-

ben und der Lösungsweg für eine vollständige Implementierung des Debuggers aufgezeigt.

Das letzte Kapitel fasst die Ergebnisse der Diplomarbeit zusammen und gibt einen Ausblick auf die nächsten Entwicklungsschritte und welche Konzepte man anwenden kann.

KAPITEL 2

Entscheidungsfindung

2.1 Verwaltung von Daten in Mehrbenutzerprogrammen

In Mehrbenutzerapplikationen gibt es zwei Hauptansätze wie man mit gemeinsamem Datenbestand und Zugriff umgehen kann [3, S. 12-13]. Beim zentralisierten Ansatz teilen sich alle Benutzer einen zentral gehaltenen Datenbestand. Hierbei muss der Zugriff auf diesen kanalisiert und serialisiert werden. In diesem Ansatz ist der Datenbestand ein Flaschenhals und begrenzt die Skalierbarkeit und Reagibilität des Antwortzeitverhaltens. Der Vorteil des zentralisierten Ansatzes ist die automatisch gegebene Konsistenz der Daten unter allen Teilnehmern. Dahingegen besitzt beim replizierten Ansatz jeder Benutzer eine eigene Kopie des Datenbestandes auf den es exklusiven Zugriff hat. Die Vorteile dieses Ansatzes sind das bessere Antwortzeitverhalten des Programmes und die damit einhergehende bessere Skalierbarkeit [7, S. 95]. Der Grund für die bessere Skalierbarkeit ist der Cachecharakter der replizierten Daten, welcher den Kommunikationsbedarf gegenüber anderen Ansätzen niedrig hält. Ein wesentliches Problem des replizierten Ansatzes ist die Erhaltung der Datenkonsistenz unter allen Teilnehmern. Dies kann durch geeignete Synchronisationsverfahren, welche entweder mit Sperren arbeiten oder aber einzelne Aktionen zurücksetzen, gewährleistet werden [7, S. 98f].

Die vorliegende Diplomarbeit verfolgt einen hybriden Ansatz, der auch in Saros [18] und Multi-User Suite [3, S. 13] Einzug fand, um die jeweiligen Vorteile der oben beschriebenen Ansätze miteinander zu vereinen. Dieser ergibt sich durch Replikation des Datenbestandes an Breakpoints [1] und zentralem Koordinator (Host). Die Konsistenz der Daten wird über eine Sternkommunikation von Host und Peers gewährleistet. Hierbei tauschen Peers untereinander keine Daten aus, sondern kommunizieren die Änderungsmitteilungen immer an den Host, welcher diese verarbeitet und allen Peers mitteilt. Der Debugger-back-endwird zentral vom Host gehalten. Die Peers kommunizieren über Proxy-Debugger-back-ends mit dem Debugger-back-enddes Hosts [3, vgl. S. 13].

[1] in lokalen BreakpointManagern

2.2 Übersicht über DebuggingCapabilities eines Java Debuggers

- Step-over: Der Debuggee führt den aktuellen Codeblock aus und suspendiert den gewählten Thread anschließend wieder.
- Step-in: Der Debuggee führt die erste Anweisung des Codeblocks aus und suspendiert anschließend im ausgewählten Thread.
- Step-out: Der Debuggee kehrt aus einem Step-in Codeblock wieder aus. Dabei wird der Thread auf der nächsten Zeile des zuvor erfolgten Step-in wieder angehalten.
- Pause: Der Thread oder die gesamte Laufzeitumgebung des Debuggee wird angehalten.
- Drop-to-frame: Debugger springt wieder zur ersten Zeile des Codeblocks, ausgehend vom letzten Step-in oder vom Programmstart des ausgewählten Threads.
- Breakpoints: Markieren Codezeilen, welche vor Ausführung zur Suspendierung des Threads oder der gesamten Laufzeitumgebung des Debuggees führen.
- Watchpoints: Spezieller Untertyp eines Breakpoint, der den Zugriff auf eine Variable überwacht und einen auf diesen zugreifenden Thread suspendiert.
- ClassPrepareBreakpoint: Breakpoint, welcher einen Thread suspendiert, sobald diese das Laden der Klassse verursacht.

2.3 Definition eines Debuggers

Ein Debugger ist ein Programm, welches ein anderes Programm schrittweise ausführen kann. Während dieses Ausführungsvorgangs bietet es außerdem die Möglichkeit Strukturen des Programmes, wie z.B. vorhandene Threads, Namen von geladenen Klassen/Modulen, Variablen und anderen noch möglichen Strukturen abzufragen oder sogar diese zu modifizieren. So ist es z.B. bei einigen Implementierungen der Java Virtual Machine (JVM) möglich Codeteile von Klassen ohne Neustart zur Laufzeit zu ersetzen (Hot Code Replacement). Im Allgemeinen wird aber nur das Verändern von Variablenwerten unterstützt. Diese Debuggereigenschaften werden als Debugging Capabilitiesbezeichnet und werden direkt von der Laufzeitumgebung des debuggten Programmes bereitgestellt, im Fall von Java von der JVM in der das gedebuggte Programm läuft. Die Rolle der Laufzeitumgebung des gedebuggten Programmes in der Beziehung mit dem ihn steuernden Debugger bezeichnet man als Debuggee. Jeder Debuggee besitzt unterschiedliche Merkmale, welche als Debugging Capabilitiesbezeichnet werden. Der Debugger kann nun entweder als Konsolenanwendung oder als GUI-Anwendung mit dem Benutzer kommunizieren. Unabhängig davon findet eine weitere Differenzierung eines Debuggers in einen sog. front-endund back-endstatt, falls die Benutzereingaben nicht direkt an den Debuggee weitergeleitet sondern vermittelt werden. Mögliche Gründe hierfür können z.B. sein:

- der Wunsch einer Trennung von Benutzerschnittstellenlogik und Debuggerlogik (MVC Pattern),
- die Schaffung neuer Debugging Capabilities(virtuelle Capabilities) aus real vorhandenen, primitiven Debugging Capabilities,
- der Wechsel des Kommunikationsprotokolls beim Debuggen zwecks Vereinfachnug (Abstraktion) und

- die Unterstützung von Fremdprogrammiersprachen (verschieden zur Sprache des Debuggee Programmes) zur Implementierung von Debuggern.

Der front-end-Part des Debuggers kommuniziert seine Befehle statt an den Debuggee an seinen back-end. Das back-endstellt einen Proxy für den Debuggee dar. Aufgrund dieser Proxyeigenschaft des back-endkann es Transformationsfunktionen wahrnehmen, die sowohl einen Wechsel des Debugging-Kommunikationsprotokolls als auch den Aufbau virtueller Capabilities aus vorhandenen ermöglichen. Weiter unten wird am Beispiel der JPDA Architektur von Java diese Unterscheidung in front-end und back-end wieder aufgegriffen. In der nun folgenden Betrachtung der Kommunikation des Debuggers mit dem Debuggee wollen wir die Kommunikation von Benutzer und Debugger außen vor lassen und den Debugger als autonom agierend annehmen.

2.4 Ablauf eines Debugging-Prozesses

Im Folgenden wollen wir annehmen, dass sich Debugger und Debuggee eines Nachrichtenkanals in der Form eines Sockets als Medium bedienen. Diese Socketverbindung sei außerdem vollduplex fähig [2]. Als Beispiele für solche Socketverbindungen sind z.B. Unixdomain- und TCP-Sockets zu nennen. Da eine solche Socketverbindung einen verbindungsorienierten Kommunikationskanal darstellt, gibt es zwei mögliche Arten wie ein Debuggingvorgang starten kann:

1. **Debugger in Serverrolle.** Nachdem der Debugger gestartet ist, legt er einen Socket an, gibt die Informationen des Sockets aus und wartet auf eingehende Verbindungsanfrage des Debuggees. Dabei kann der Debugger auch aktiv den Debuggee im Debugmodus mit der Information des wartenden Sockets starten. Dies erfordert die Eigenschaft, dass der Debugger multi-threaded ist.
2. **Debugger in Clientrolle.** Der Debuggee wird im Debugmodus mit Angabe des zu benutzenden Sockets gestartet und wartet auf eine eingehende Verbindungsanfrage des Debuggers, bis zu dieser wird das Programm nicht ausgeführt. Der Debugger wird nun mit dieser Information gestartet und verbindet sich direkt mit dem Debuggee. Wie im ersten Fall auch, kann der Debugger selbst aktiv den Debuggee starten und sich anschließend mit diesem verbinden. In diesem Fall muss der Debugger nicht einmal multi-threaded sein.

Nach der Verbindungsherstellung installiert der Debugger seine Break- und Watchpoints [3] und andere von den Debugging Capabilitiesunterstützten Eigenschaften im Debuggee und gibt die Programmausführung frei. Im weiteren Verlauf wird die Programmausführung entweder Schrittweise ausgeführt und angehalten oder aber ganz ausgeführt und von Break- und Watchpoints ereignisgesteuert angehalten. Hier kommt nun eine weitere wichtige Eigenschaft in der Kommunikation zum Vorschein: die Asynchronität von Nachrichten. Zunächst einmal muss man wissen, dass man Variablenwerte **nur** im Zustand der Programm- oder Threadsuspendierung abfragen und verändern kann. Eine solche Anweisung stellt eine Anfrage bzw. ein Kommando an den Debuggee dar, auf das der Debugger eine Antwort vom Debuggee, entweder in Form des Variablenwertes oder einer Bestätigung, erwartet. Nehmen wir nun den einfachen Fall an, dass es sich bei dem debuggten Programm um ein Single-Threaded Programm handelt. Eine Step-Anweisung kann, wenn es sich beispielweise um die Ausführung eines Codeblocks [4] handelt, eine unbestimmt lange Ausführungszeit haben.

[2] impliziert Bidirektionaliät
[3] Ein Watchpoint ist eine Unterart eines Breakpoints und wird nachfolgend nicht gesondert behandelt.
[4] eine Funktion

In dieser Zeit kann der Debugger in der Step-Ausführung solange blockieren bis die Rückmeldung vom Debuggee über das Ausführungsende des Steps zurückkommt. Dadurch gebe es in dieser Zeit keine Möglichkeit eine Anfrage zum Wert einer Variablen an den Debuggee zu stellen. Hier gebe es also keine Probleme. Was aber passiert, wenn es sich beim Programm im Debuggee um ein Multi-Threaded Programm handelt? Hier kann es durchaus zu Situationen kommen, in denen ein bestimmter Thread angehalten ist und man dadurch die Möglichkeit hat Werte von darin ausgeführtem Code abzufragen und ein anderer Thread parallel weiter arbeitet. Es kann nun passieren, dass sich genau zur selben Zeit wie zur Abfrage des Variablenwertes ein asynchrones Ereignis des laufenden Threads, wie z.B. Threadsuspendierung aufgrund eines Breakpoints, ankündigt. Der Debugger muss nun die Antwort des erfragten Variablenwertes von der ansynchron erhaltenen Nachricht unterscheiden und verarbeiten können. Diesem Umstand der Parallelität von synchronen Antworten auf Anfragen und asynchronen Ereignissen kann nun auf zwei Arten Rechnung getragen werden. Erstens man benutzt zwei Socketverbindungen, die eine für eintreffende asynchrone Nachrichten seitens des Debuggees [5] und die andere für synchrone Antworten auf Anfragen durch den Debugger [6]. Zweitens der Debuggee markiert seine ausgehenden Antworten bzgl. des Typs „Antwort auf Anfrage“ oder „asynchroner Eventtyp“. Letztere Variante benutzt z.B. das JDWP-Protokoll in der JDPA-Architektur von Java.

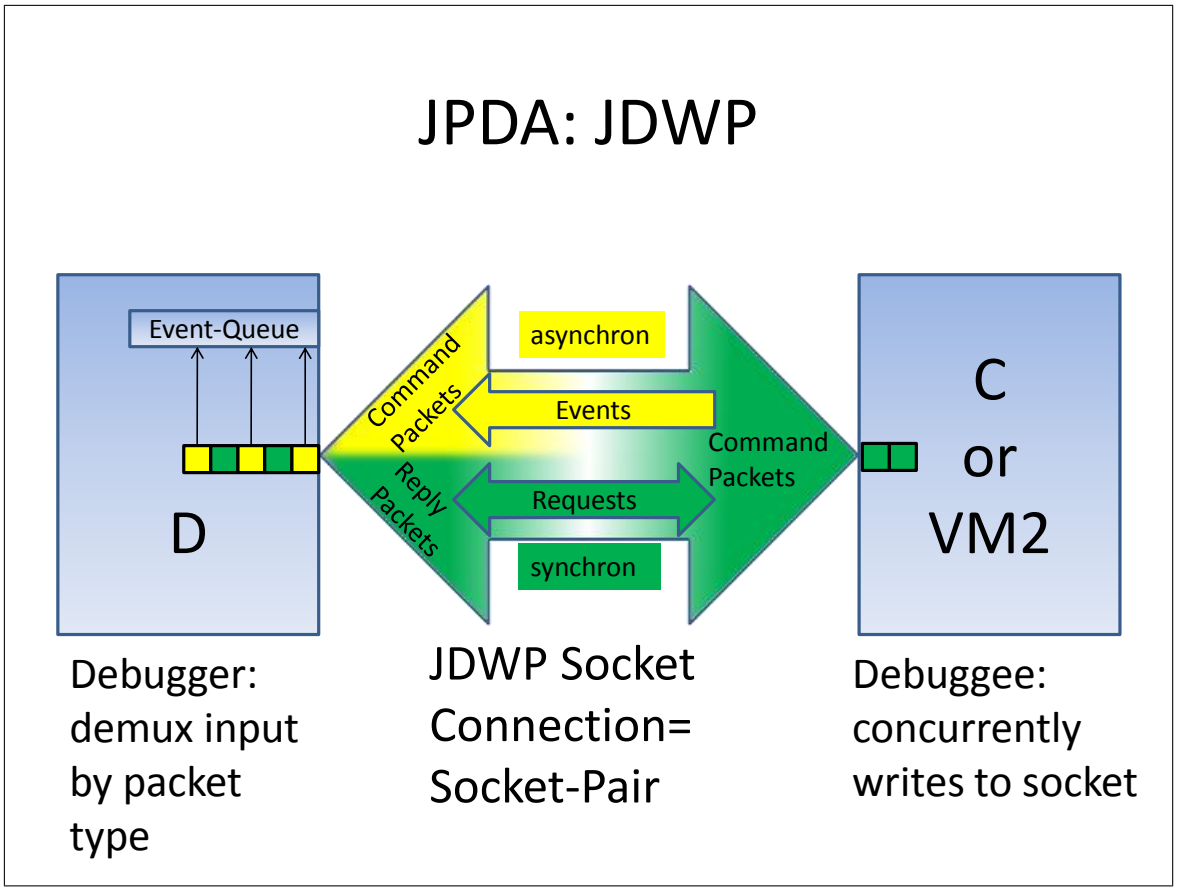

Abbildung 2.1: JDWP-Kommunikation. Symbolik siehe Abbildung 2.3

[5] dieser Kanal kann sogar unidirektional sein
[6] hier genügt Halbduplexfähigkeit

2.5 Java Debugging Architektur (JPDA)

Java bietet zur Implementierung von Debuggern verschiedene aufeinander aufbauende Ebenen von Interfaces an. Diese Architektur bezeichnet man als die Java Platform Debug Architecture (JPDA). Man untercheidet dabei die Ebenen:

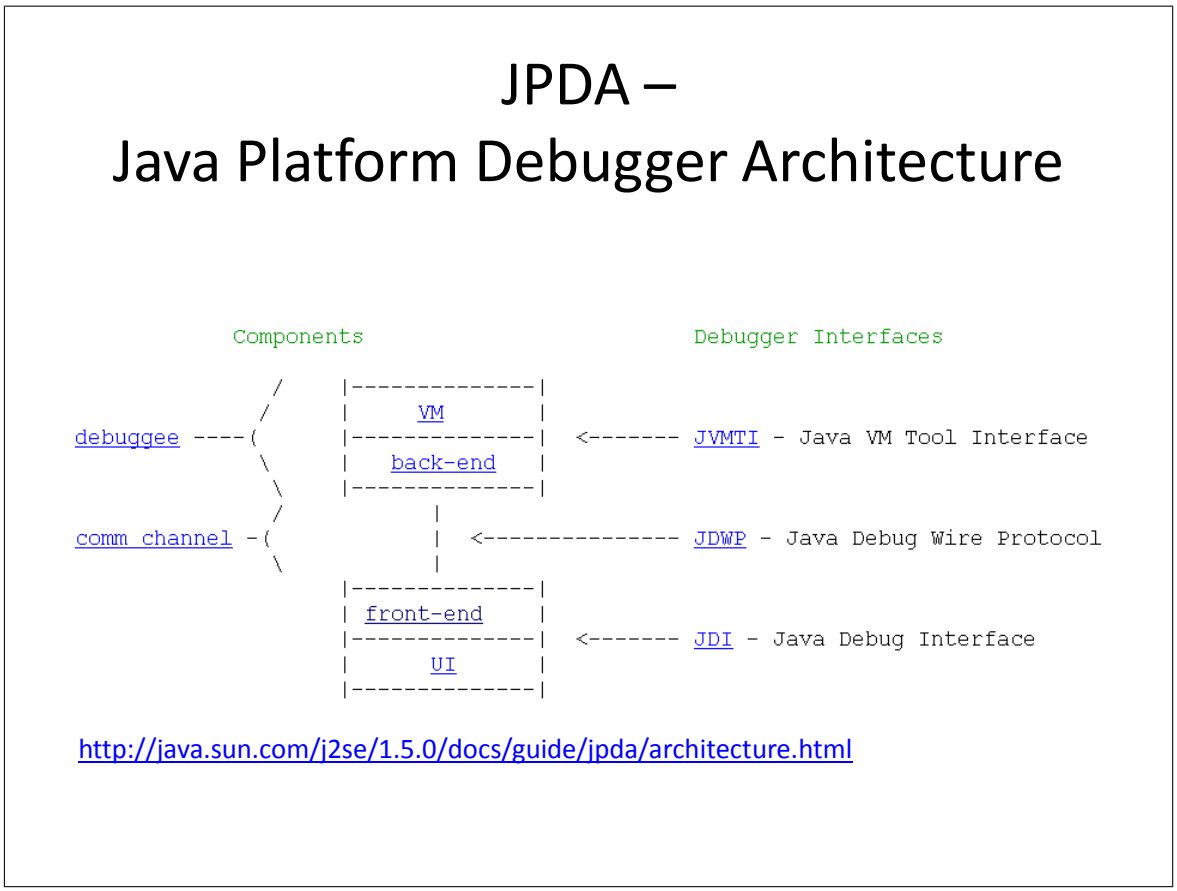

Abbildung 2.2: Die drei Schichtenarchitektur von JPDA

- JVMTI - Java Virtual Machine Tool Interface. Eine Schnittstelle zur Implementierung von Debuggern, welche im selben Prozess laufen wie das gedebuggte Programm. Ein Debugger, welches diese Schnittstelle benutzt, kann in einer beliebigen Programmiersprache geschrieben werden und benutzt entsprechende Bibliotheksaufrufe um auf diese zuzugreifen.
- JDWP - Java Debug Wire Protocol. Eine ebenfalls programmiersprachenunabhängige Schnittstelle, welche ein Kommunikationsprotokoll darstellt, mit der ein Debugger die virtuelle Maschine des gedebuggten Programms steuern kann. In dieser Konstellation ist es aber auch möglich, dass der Debugger nicht direkt auf die virtuelle Maschine zugreift, sondern über einen Mediator, den sogenannten Debugger-back-end, vermittelt wird. Man unterscheidet die beiden Rollen Debugger und Debuggee. Der Kommunikationspartner des Debuggers ist sein Debuggee [7]. Falls dieser Debuggee nicht der virtuellen Maschine mit dem gedebuggten Pro-

[7] in diesem Kontext wird der Debugger als aktives Objekt ohne Benutzersteuerung betrachtet

gramm entspricht, dann nennt man den Debuggee das back-enddes Debuggers und selbigen den front-end des Debuggers.

- JDI - Java Debug Interface. Dies ist eine Schnittstelle, welche von rein in Java geschriebenen Debuggern genutzt werden kann. Ein Debugger, der diese Schnittstelle benutzt, ist der sogenannte front-enddes Debuggers und das diese Schnittstelle implementierende Programm dessen back-end. Das back-endwiederum kann entweder direkt mit der virtuellen Maschine kommunizieren oder selber ein front-end eines zwischengeschalteten back-ends darstellen. Zusammenfassend gilt, dass der Debugger immer mit dem Debuggee kommuniziert bzw. diesen steuert. Im Falle, dass der Debuggee nicht die virtuelle Maschine ist, stellt der Debugger den front-endund der Debuggee den back-end-Part des so zusammengsetzten (zusammenarbeitenden) Debuggers dar.

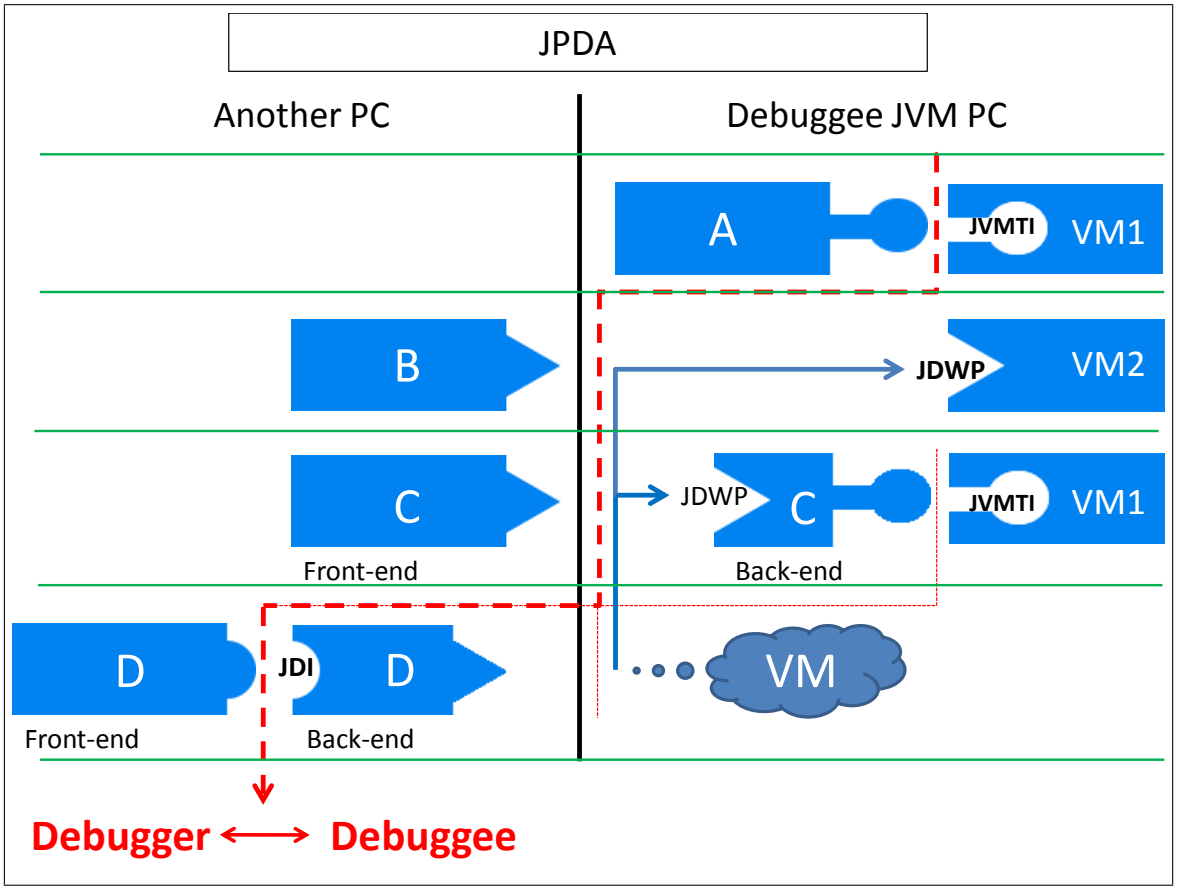

Abbildung 2.3: Aufeinander aufbauende Schnittstellen

Diese Dreischichtenarchitektur von Interfaces ähnelt in gewisser Weise dem OSI Referenzmodell [15, S. 53-57], in dem ebenfalls nur benachbarte Schichten Informationen untereinander austauschen und ansonsten unabhängig sind. Diese Modularität wird dazu benutzt um auf höheren Ebenen (komplexere/höher entwickelte) abstraktere Konstrukte und Funktionen anzubieten als auf niedrigeren. Damit wird dem Gedanken der Modulariät und Wiederverwendung von Code Rechnung getragen und eine differenzierte Granularität in der Entwicklung von Debuggern ermöglicht. Diese Granularität ist vergleichbar mit derjenigen zwischen den Programmierebenen von C, Assembler

und Maschinencode.

2.6 Extension-Points, Extensions, Adapter und AdapterFactories

Eclipse kann man sich als ein Puzzle vorstellen, deren einzelne Funktionen auf die Puzzle-Teile verteilt sind und sich gegenseitig zu einem Gesamtkonzept verbinden [8]. Dabei entspricht einem Puzzle-Teil eine abgegrenzte Menge von Funktionalität, die über einen eigenen Klassenpfad und Sichtbarkeitsbereich verfügt. Um aber als Puzzle-Teil gebrauch von Funktionalität seitens anderer Puzzle-Teile (Plug-Ins) machen zu können und seinerseits anderen Plug-Ins die eigene Funktionalität zur Verfügung zu stellen, gibt es sogenannte Extension-Points.

2.6.1 Extension-Points

Einem Extension-Point eines Plug-Ins entspricht bildlich eine Steckdosenleiste mit festgelegter spezifischer Buchsenform; so zum Beispiel Flachstecker-Buchse oder Rund-Buchse [5, S. 5]. Dieser bildlichen Umschreibung eines Extension-Points entspricht softwaretechnisch ein Interface für den jeweiligen Extension-Point. Ein Plug-in stellt über Extension-Points eine API bereit, mit der es Benutzern der Extension-Points mitteilt, welche Funktionserweiterungen eingebunden werden können. So kann ein Plug-in z.B. über einen Extension-Point anderen Plug-Ins ermöglichen sich als Event-Listener zu registrieren oder aber Menüeinträge und Symbolleisten zu erweitern, indem Attribute wie Beschriftung, Symboldatei usw. als zu setzende Parameter definiert und evtl. ein Interface von Seiten des Extension-Point-Anbieters vorgegeben wird.

Die Parameter des Extension-Points definiert man in einer sogenannten *.exsd Datei, welche als Atrribut vom `<extension-point>` Element in der Datei `plugin.xml` verlinkt wird. Diese `*.exsd` Datei wird ausgelesen, wenn man sich die Beschreibung eines Extension-Points im „Plug-in Manifest Editor“ der Eclipse PDE Perspektive anzeigen läßt.

2.6.2 Extensions

Jede Benutzung eines Extension-Points stellt eine Extension dar. In der `plugin.xml` kommt dies durch das xml Element extension zum Ausdruck. Im Unterschied zur Definition eines Extension-Points wird bei der Erstellung einer Extension `<extension point="...">` statt `<extension-point id="...">` geschrieben. Dabei enthält das Attribut `point` den Wert des Attributes `id` der Extension-Point Definition, allerdings in vollqualifizierter Schreibweise mit ID des den Extension-point definerenden Plug-Ins.

Eine Extension wird also vom Extension-Point definerenden Plug-In fremdaufgerufen [5, S. 49]. Dies entspricht dem Hollywood Prinzip: Don’t call us, we call you!

[8] Eigener Blogartikel

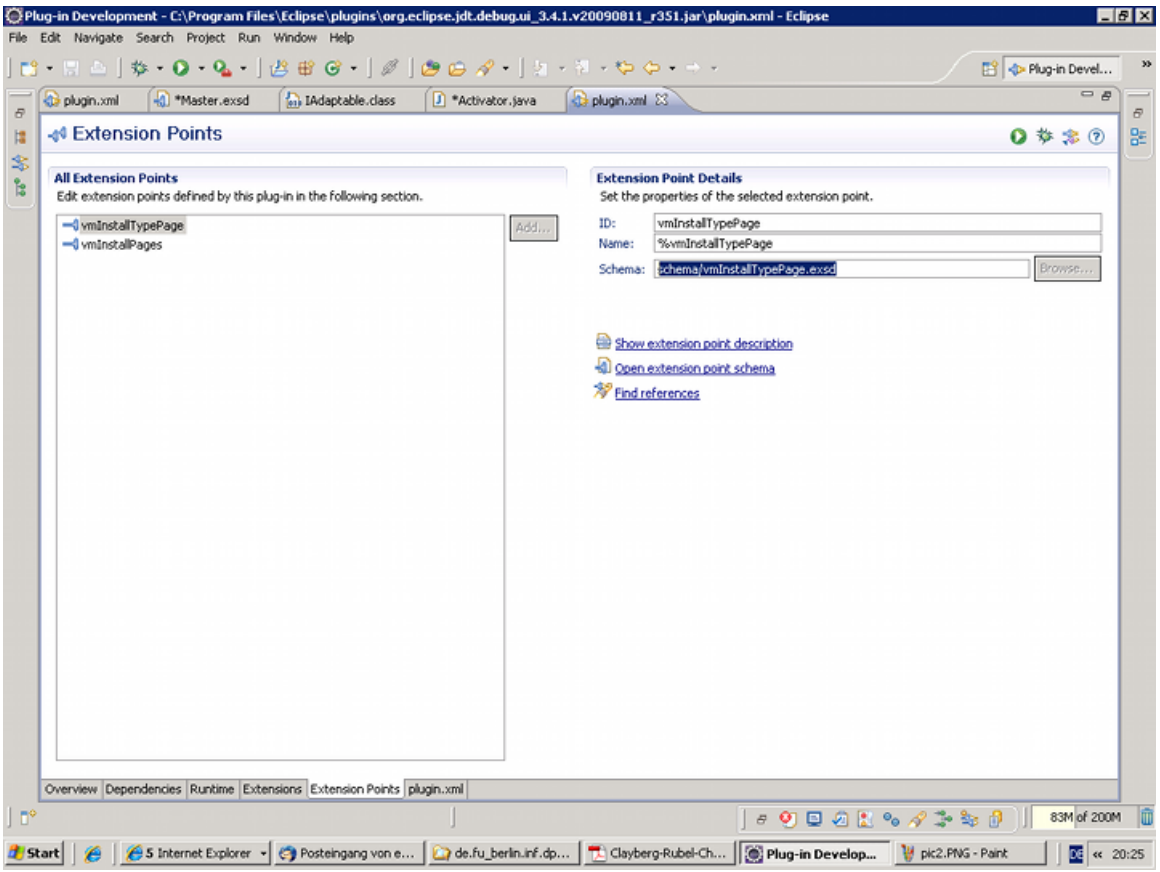

Abbildung 2.4: Ein Extension-Point im PDE Editor

2.6.3 Adapter

Die zweite Möglichkeit eigenen Code in das bestehende Plug-In-Geflecht von Eclipse einzuschleusen ist in der Definition von Adaptern gegeben [5, S. 283-294]. Ein Adapter bietet die Möglichkeit eine bestehende Klasse im Nachhinein um Interface Implementierungen zu erweitern. Dabei wird die Signatur der Klasse nicht verändert und es treten keine Konflikte mit davon abhängenden anderen Klassen auf. Was passiert ist vielmehr ein Eingriff zur Laufzeit. Nahezu jede öffentliche Klasse eines Plug-Ins implementiert hierzu das Interface IAdaptable. Wenn man nun später weitere Interfaces unterstützen möchte ohne die Signatur der Klasse ändern zu müssen, überschreibt man einfach die einzige Methode aus `IAdaptable`, `getAdapter(Class c)`.
Darin schreibt man dann sowas wie:

```
class B extends A
{
    private NewInterface delegate = new NewInterfaceImpl();
    class NewInterfaceImpl implements NewInterface
    {
        //Innere Klasse hat vollen Zugriff auf A mittels A.this
        public f1()
```

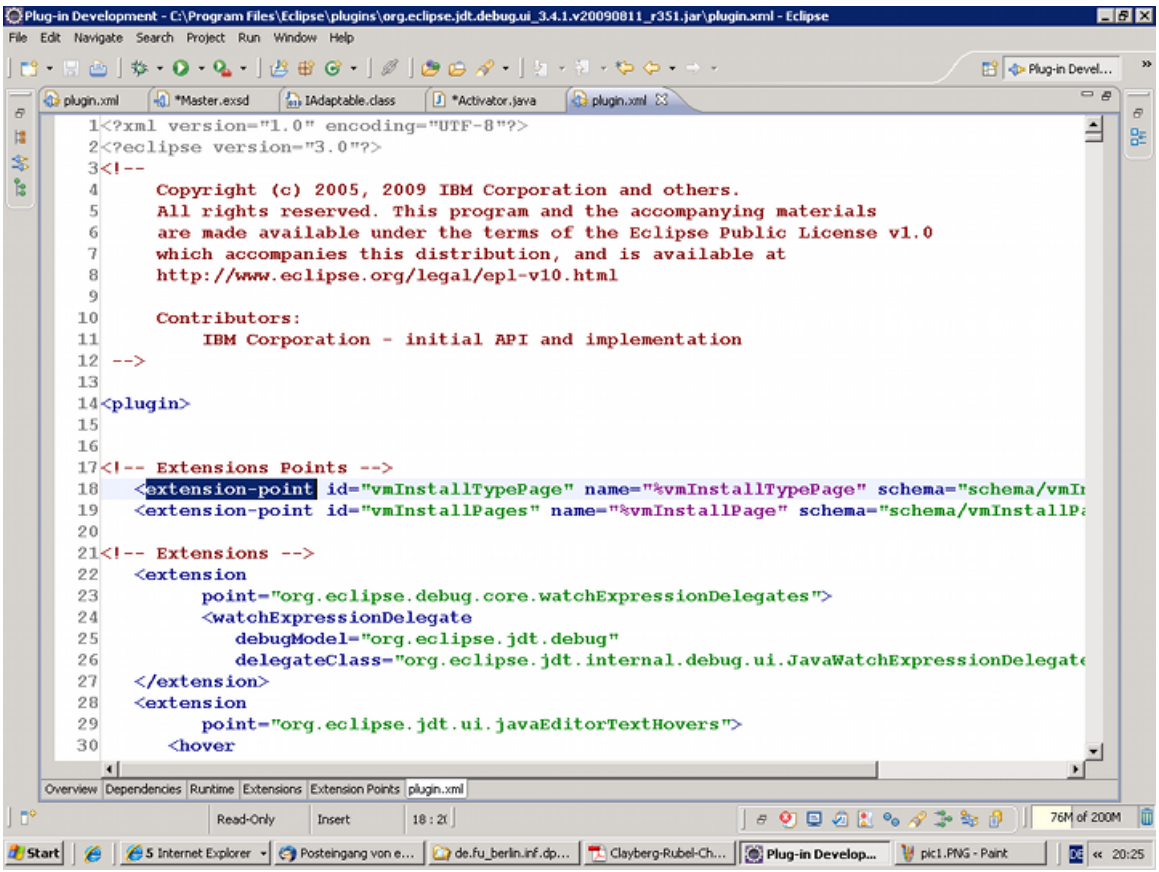

Abbildung 2.5: Extension-Point in XML Quelltext Ansicht

```
            {
                //benutze Methoden von A so, dass f1 sinnvoll
                   realisiert wird.
            }
            ...
        }
        public Object getAdapter(Class c)
        {
            if (c == NewInterface.class)
            return delegate;
            else
            super.getAdapter(c);
        }
    }
```

Möchte man das neue Interface nutzen so kann mit folgender Abfrage geprüft werden, ob das Interface unterstützt wird:

```
A a = ...;
Object ni = null;
```

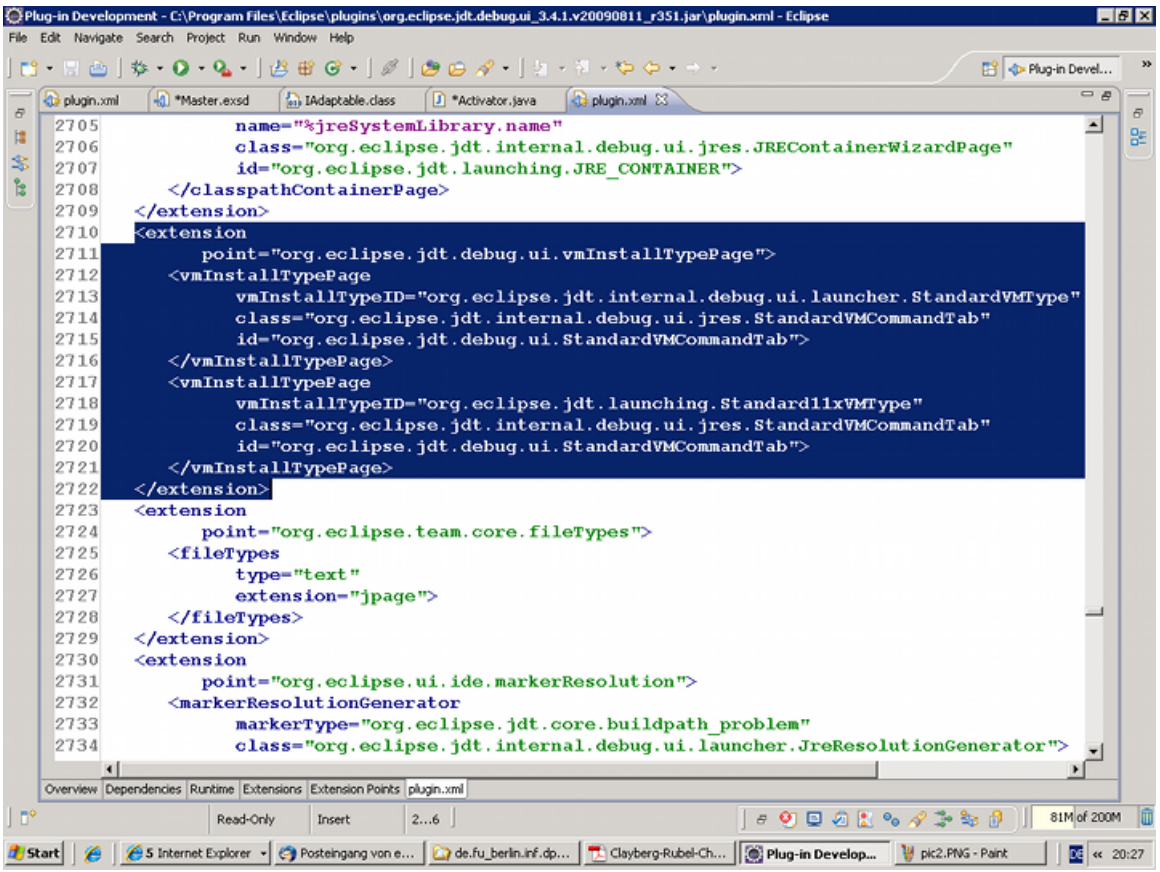

Abbildung 2.6: Extension implementiert Extension-Point

```
if((ni=a.getAdapter(NewInterface.class))!=null) ((NewInterface)
    ni).f1();
```

Auf diese Weise kann man also die dynamische Signatur einer Klasse ändern.
Was aber hat das mit dem Einbringen von eigenem Code in das bestehende Plug-In-Geflecht zu tun? Wir können ja nicht davon ausgehen, dass wir die jeweilige Klasse überschreiben und dann statt der Originalklasse unsere instantiiert wird. Daher gibt es noch eine zweite Möglichkeit der Nutzung des Adapter-Prinzips.

2.6.4 AdapterFactories

Eine AdapterFactory ist eine Klasse, welche das Interface IAdapterFactory implementiert. Dieses Interface hat nur die beiden Methoden

```
//Gibt Liste aller unterstützten Adapter zurück
public Class[] getAdapterList();
/*adaptableObject teilt der Fabrik mit, welches Objekt um einen
    entsprechenden Adapter vom Typ adapterType gefragt wurde.
*/
```

```
public Object getAdapter(Object adaptableObject, Class
   adapterType);
```

Eine Fabrik verwaltet eine Menge von Adaptern im Auftrag eines anderen Objektes. Die Verknüfung von Fabrik und Klient, welcher wie nachfolgend begründet vom Typ IAdaptable ist, erfolgt über einen systemweiten Adaptermanager. Dieser Adaptermanager implementiert das Interface IAdapterManager mit u.a. folgender Methode

`public void registerAdapters(IAdapterFactory factory, Class client);`
Auf den systemweiten Adaptermanager erhält man Zugriff über `Platform.getAdapterManager()` und kann nun (den Objekten) einer Klasse eine entsprechende Fabrik zuordnen. Dies geschieht durch Nennung des entsprechenden Grundtyps, auf den hin die Fabrik aktiv werden soll.
Was jetzt geschieht ist folgendes:
Ein Objekt, welches eine Fabrik damit beauftragen will, entsprechende Adapter für ihn zu produzieren, implementiert die Methode aus `IAdaptable` auf folgende Weise:

```
public Object getAdapter(Class adapterType){
    return Platform.getAdapterManager().getAdapter(this,
       adapterType);
}
```

Der Adaptermanager durchläuft nun die Vererbungshierarchie des Aufrufers von unten nach oben bis zur Klasse Object und fragt in jeder Stufe mit dem enstprechenden Klassenobjekt (Typ Class) als Schlüssel die Hashtabelle nach registrierten Fabriken. Ob der gesuchte Adapter von der Fabrik unterstützt wird erkennt der Adaptermanager durch Aufruf der Fabrikfunktion getAdapterList(). Im Erfolgsfall wird die Suche frühzeitig beendet und die entsprechende Fabrik folgendermaßen aufgerufen.

```
//innerhalb von getAdapter(Object adaptableObject,Class
   adapterType) vom IAdapterManager
...
//Nachfolgend Pseudo-Code
IAdapterFactory fab;
fab = findFactory(adaptableObject,adapterType);
return fab.getAdapter(adaptableObject,adapterType);
...
```

Der Grundgedanke dazu ist folgender:
Eine Klasse hat eine Vererbungshierarchie, welche aus Klassen und Interfaces bestehen kann. Gehen wir von folgender Vererbungshierarchie aus:

```
public class C extends B implements IC_1, IC_2 {...}

public class B extends A implements IF {...}

public class A {...}
```

Wenn wir nun ein Objekt
`Object obj = new C();` erzeugen, dann ist folgendes wahr:

```
(obj instanceof C) \&\& (obj instanceof B) \&\& (obj instanceof
    A) \&\&
(obj instanceof Object) \&\& (obj instanceof IC_1) \&\& (obj
   instanceof IC_2) \&\& (obj instanceof IF)
== true
```

Wir haben also für die Registrierung der Adapterfabrik für die Klasse C folgende Möglichkeiten:

```
//Konkrete Fabrik muss je nach gewählter Alternative
   implementiert werden.
IAdapterFactory fab = ...;

IAdapterManager man = Platform.getAdapterManager();

1. man.registerAdapters(fab,C.class);
2. man.registerAdapters(fab,B.class);
3. man.registerAdapters(fab,A.class);
4. man.registerAdapters(fab,Object.class);
5. man.registerAdapters(fab,IC_1.class);
6. man.registerAdapters(fab,IC_2.class);
7. man.registerAdapters(fab,IF.class);
```

Bei Variante 1 sieht die Fabrik so aus:

```
class C_casting_Factory implements IAdapterFactory
{
    ...
    Object getAdapter(Object client,
    Class adapterType){
        C casted = (C) client;
        if(adapterType==Adapter1.class){
            //Konstruktor: Adapter1_impl(C client);
            return new Adapter1_impl(casted);
        }
        elseif(adapterType==Adapter2.class){
            //Konstruktor: Adapter2_impl(C client);
            return new Adapter2_impl(casted);
        }
        ...
    }
}
```

und bei Variante 3 so:

```
class A_casting_Factory implements IAdapterFactory
{
    ...
    Object getAdapter(Object client,
    Class adapterType){
        A casted = (A) client;
        if(adapterType==Adapter1.class){
            //Konstruktor: Adapter1_impl(A client);
            return new Adapter1_impl(casted);
        }
        elseif(adapterType==Adapter2.class){
            //Konstruktor: Adapter2_impl(A client);
            return new Adapter2_impl(casted);
        }
        ...
    }
}
```

Dadurch kann es vorkommen, dass für eine Klasse gleich mehrere Fabriken in Frage kommen, welche den angefragten Adapter zurückgeben können. In diesem Fall gilt die Regel. Die Fabrik, welche mit dem spezifischsten Klassenobjekt regsitriert wurde erhält den Zuschlag und darf den entsprechenden Adapter zurückgeben. Kommen wir nun zurück zu unserem Ausgangsgedanken: Wir wollen von externer Seite neue Adapter für eine Klasse hinzufügen oder bestehende Adapter überschreiben. Dazu kann man den Extension-Point

`org.eclipse.core.runtime.adapters`

verwenden oder aber zur Laufzeit folgenden Code

```
Platform.getAdapterManager().registerAdapters(fab,client)
```

schreiben. Eine mögliche Extension könnte folgendermaßen aussehen:

```
<extension point="org.eclipse.core.runtime.adapters">
  <factory adaptabletype="pfad.zum.klienten.der.fabrik"
      class="pfad.zur.fabrik.klasse">
    <adapter type="pfad.klasse.adapter1"/>
    <adapter type="pfad.klasse.adapter2"/>
  </factory>
</extension>
```

Dabei ist die Aufzählung der Adapter nur als deklarative Wiederholung von der Fabrik-Methode `getAdapterList()` zu verstehen. Die obige Extension Variante der Fabrikregistrierung ersetzt die Notwendigkeit bei Plug-In Aktivierung folgenden Code schreiben zu müssen:

```
Platform.getAdapterManager().registerAdapters(pfad.zur.fabrik.
   klasse,pfad.zum.klienten.der.fabrik);
```

Die registrierte Fabrik wird aber nur dann bei der Adaptersuche miteinbezogen, wenn das die Fabrik definierende Plug-In geladen ist. Das bloße definieren der Extension hat nicht die Aktivierung des zugehörigen Plug-Ins zur Folge.

2.7 Eclipse Standard Debug Model (ESDM)

Eclipse liefert ein Standardgerüst für die Implementierung von Debuggern an, welches aus einem Datenmodell und einem auf diesem basierenden GUI-Framework besteht. Das Datenmodell entspricht dabei dem transistenten Zustand der virtuellen Maschine einer imperativen Programmiersprache. In diesem Zustandsmodell sind die gemeinsamen Elemente von imperativen Programmiersprachen wie Threads, Stackframes, Variablen u.a. als Abstraktionen in Form von Interfaces definert, welche vom generischen GUI-Framework angesprochen werden. Dieses Datenmodell entspricht einer abstrakten Form eines Debuggees und stellt nicht den eigentlichen Debugger dar. Daher ist im Diagramm auch die Tatsache zu erklären, dass einem IThread-Element eine Aggregation von IBreakpoint-Elementen zugeordnet ist. Dies macht nur für einen Debuggee bzw. dessen Zustandsmodell Sinn und ist damit zu erklären, dass ein IThread in seinem suspendierten Zustand durch eine Menge ihn in diesen Zustand versetzender IBreakpoint-Elemente gebracht worden sein kann.

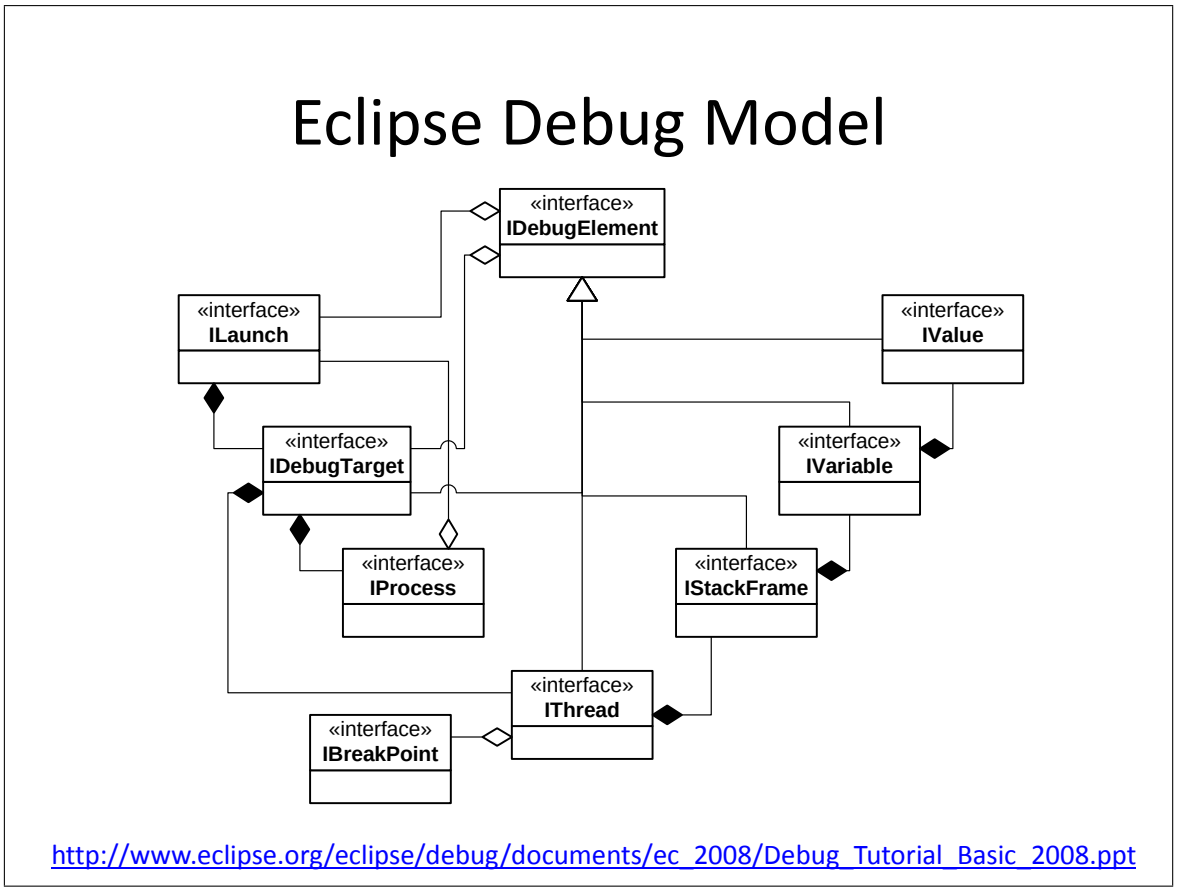

Abbildung 2.7: Eclipse Standard Debug Model

Wir können in diesem Zustandsmodell folgende Eigenschaften ablesen:

1. Wir können die Elemente aus dem UML Diagramm auch in einer Baumstruktur darstellen, wenn wir als Eltern-Kind-Beziehung „instantiiert durch“ benutzen.
2. Als Baum betrachtet stellt ILaunch die Wurzel des Baumes dar. Es ist folgerichtig ein Kompositionsobjekt der restlichen Elemente.
3. IProcess enthält einen Verweis auf das ILaunch-Objekt in einer Aggregationsassoziation. Dies ist aber für das Verständnis des Zustandsmodells irrelevant.
4. IDebugTarget kann man als zweite eigentliche Wurzel des Baumes betrachten, da es das einzige Kindelement von ILaunch ist und alle weiteren IDebugElement-Objekte auf ihn verweisen.
5. Es gibt eine zentrale Vererbungsbeziehung in Form eines gemeinsamen IDebugElement-Objektes, das über eine Aggregationsassoziation einen Verweis zum IDebugTarget herstellt. Jedes der Elemente außer ILaunch, IProcess und IBreakpoint ist ein IDebugElement. Die Assoziation mit dem ILaunch-Objekt ist irrelevant fürs Verständnis und drückt eine semantische Beziehung aus.
6. IStackFrame, IVariable und IValue sind Komponentenobjekte ihrer zugehörigen Kompositionsobjekte. Nach der Diagrammdarstellung kann man keine Rückreferenzen auf die sie beinhaltenden Kompositionsobjekte erwarten. Jedoch enthalten bis auf IVariable alle Komponentenobjekte Rückreferenzen auf Ihre Komponenten (Eltern), wenn man sich die zugehörigen Interface-Klassen im Plug-In `org.eclipse.debug.core` anschaut.

Das Framwork kommuniziert eingehende Benutzeraktionen an die Debuggeranwendungslogik. Diese wird ebenfalls durch die Interfaces des Debugmodells spezifiziert und ist Ausdruck der Capabilities des Debuggees. Das implementierte Debugmodel steuert seinerseits den tatsächlichen Debuggee (virtuelle Machine oder back-end) und legt erhaltene Antworten und Events im Debugmodell ab, welches nach dem Observerpattern vom Debugger-Userinterface abgefragt wird. Hier kommt das Prinzip der Trennung von Funktionalität nach dem Muster von Model-View-Controller (MVC) zum Einsatz. Dem Model entspricht der Baum mit der Wurzel IDebugTarget - in einer „instantiiert durch“ Beziehung - als zeitlich veränderliches Zustandsmodell des Debuggees (virtuellen Maschine), der View entspricht die grafisch darstellende GUI-Schicht des Debuggers und dem Controller entspricht die Debuggeranwendungslogik, welche die GUI-Events verarbeitet und in Befehle an den eigentlichen Debuggee umsetzt. Aufgrund dieser Trennung kann ein neuer Debugger relativ einfach in Eclipse integriert werden, indem es die spezifizierten Interfaces des Debuggee-Zustandmodells implementiert und im Gegenzug von Eclipse das vorhandene GUI-Framework nutzen kann. Was in diesem Zusammenhang noch zu erwähnen ist, dass es noch eine weitere Abstraktion zwischen GUI-Eventhandling und Debuggeranwendungslogik gibt. Diese entsteht durch das Vorhandensein eines BreakpointManagers, welches vom DebugPlugin-Objekt verwaltet wird und seinerseits nach dem Observer-Pattern von der Debuggeranwendungslogik überwacht werden kann. Die vom Framework bereitgestellten GUI-Eventhandler manipulieren hauptsächlich den vom BreakpointManager verwalteten Datenbestand. Wenn es Funktionalität gibt, die nicht durch das Standardmodell abgedeckt ist, so kann der konkrete Debugger das User-Interface (UI) so erweitern, dass es sich direkt auf das speziellere Datenmodell bezieht und somit die graphische Bedienbarkeit der zusätzlichen Merkmale ermöglichen.

2.8 Java Debugger als Implementierung des ESDM

Der in Eclipse integrierte Java Debugger stellt eine Implementierung des Datenmodells dar und baut dieses über JDI-Aufrufe auf. Konkret bedeutet dies, dass es eine Klasse gibt, welche Objekte von Klassen instantiiert, die die Interfaces des ESDMs implementieren, und diese Objekte mit Inhalt gefüllt werden, die durch JDI-Aufrufe an das JDI-back-endgeneriert worden sind. Bei der Betrachtung der Einbindung des Java Debuggers in Eclipse müssen wir uns klar machen, wo die Trennung von spezifischen JDI-Klassen und von das ESDM implementierenden Klassen stattfindet und wie diese beiden Welten miteinander agieren. Dazu betrachten wir die JDI-Spezifikation mal etwas genauer:

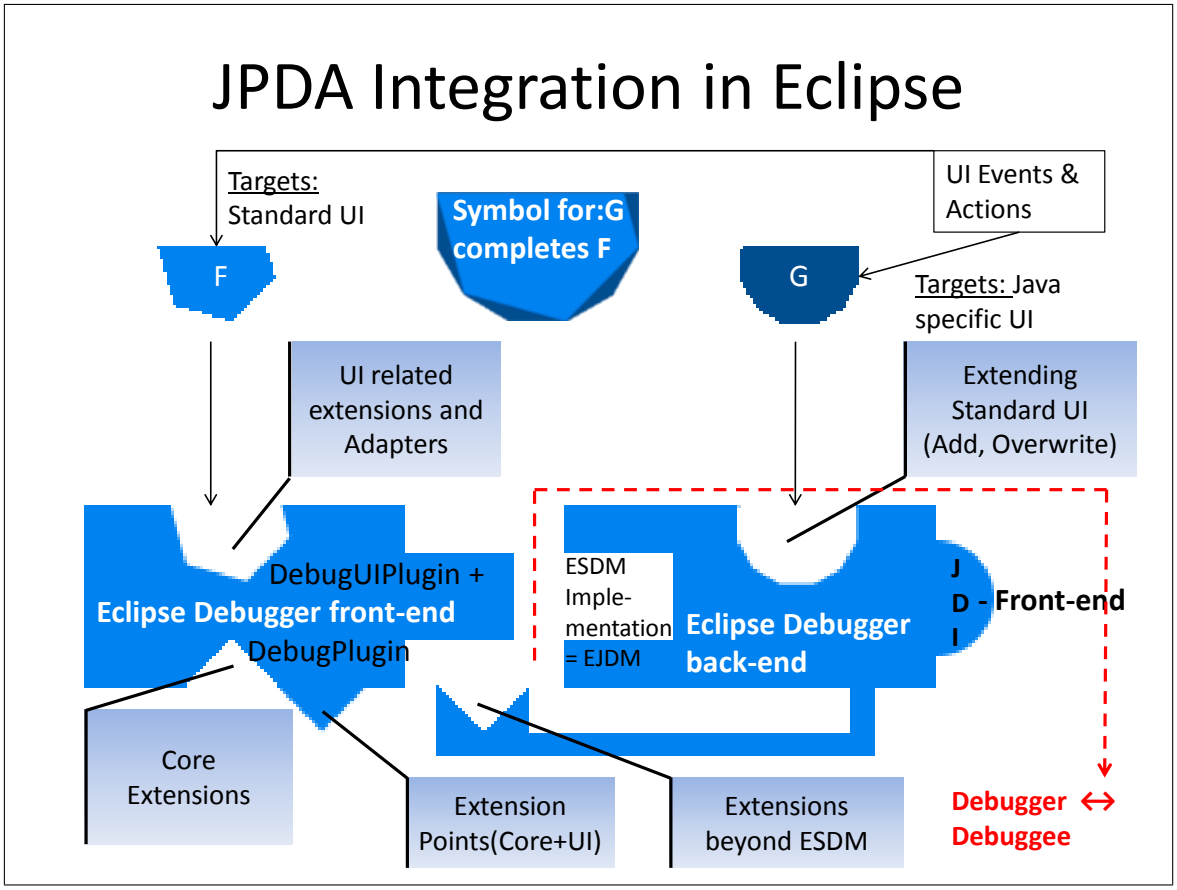

Abbildung 2.8: JPDA Integration in Eclipse

2.8.1 JDI Spezifikation

Im Paket `com.sun.jdi.connect` gibt es drei Interfaces, welche eine virtuelle Maschine instantiieren:

1. LaunchingConnector: `VirtualMachine launch(...)`, startet die VM mit dem zu de-

buggenden Programm, verbindet sich mit dieser und liefert ein Handle darauf zurück.

2. AttachingConnector: `VirtualMachine attach(...)`, verbindet sich mit der VM eines bereits gestarten Programmes und liefert ein Handle zurück.
3. ListeningConnector: `String startListening(...)` und `VirtualMachine accept(...)`, startet zunächst einen Server und gibt dessen Verbindungsinformationen aus und wartet anschließend blockierend mittels `accept(...)` auf Verbindungsherstellung durch die VM des zu debuggenden Programmes.

Die `VirtualMachine` erbt vom Interface `com.sun.jdi.Mirror`, welches Elemente einer fremden virtuellen Maschine kennzeichnet, die durch einen Debugger ausgelesen oder verändert werden können. Daher kommt auch der naheligende Name `Mirror`, welches das lokale Widerspiegeln von externen Information andeutet. Die beiden in `Mirror` defnierten Methoden sind:

- `String toString()`, gibt einen beschreibenden Namen für das jeweilge Element aus. Dies entspricht normalerweise dem Laufzeitinterfacenamem, z.B. `LocalVariable`.
- `VirtualMachine virtualMachine()`, gibt die `VirtualMachine`-Instanz zurück, aus dem das jeweilige `Mirror`-Objekt enstammt.

Aus diesem letzten Punkt erkennt man, dass alle anderen Objekte, die ebenfalls das Interface Mirror implementieren, über eine VirtualMachine Instanz erzeugt werden. Sie sind Nachfahren in einem Baum mit der VirtualMachine als ihre Wurzel.

Da alle das Interface Mirror implementierenden Klassen nur über Methodenaufrufe der VirtualMachine-Instanz indirekt instanziiert werden, kann man die VirtualMachine implementierende Klasse als den JDI-Debugger klassifizieren. Diese Festlegung ist eigentlich nicht ganz sauber, da sich die Funktionalität des Debuggers auf alle Klassen verteilt, welche eines der von Mirror erbenden Interfaces implementieren. Es ist daher nur eine semantische willkürliche Festlegung. Ein anderes Objekt, das sich dieser Klassen über Methodenaufrufe auf dem VirtualMachine-Objekt bedient, nennen wir einen JDI-front-end.

Einen solchen JDI-front-endkönnen wir nun beim eclipse.jdt.debug.core Plug-In in einer Instanz der Klasse org.eclipse.jdt.internal.debug.core.model.JDIDebugTarget identifizieren.

Eclipse Java Debug Model als Eclipse Java Debugger back-end und JDI-front-end

Die Klasse JDIDebugTarget implementiert das Interface IDebugTarget (über IJavaDebugTarget) aus dem Eclipse Standard Debug Model (ESDM) und stellt somit die Wurzel des Objektbaumes dar, welches sich aus der Beziehung „instanziiert durch“ ableiten lässt. Das heißt, dass alle im ESDM vorhandenen Elemente, durch die JDIDebugTarget-Instanz erzeugt werden. Das Eclipse Java Debug Model (EJDM) erweitert zum Teil die Interfaces des ESDM zwecks vorhandener erweiterter Debugging Capabilities. Diese zusätzlichen Interfaces müssen teilweise durch neue GUI-Elemente und Actionhandler angesprochen werden und finden in weiteren Plug-Ins des JDT-Features Einzug in Eclipse. Die in den Interfaces spezifizierten Funktionen werden nun dadurch realisiert, dass man diese über eine Serie von JDI-Aufrufen und ein entsprechendes JDI-Eventhandling realisiert.

Es handelt sich also beim EJDM um einen Debugger-back-endaus der Perspektive des Eclipse Debug Frameworks als auch um einen Debugger-front-endaus der Perspektive des JDI-Frameworks, letzteres in der Form der JDI implementierenden Klassen.

Der Weg geht hierbei immer über die VirtualMachine Instanz, welche ihrerseit ausschließlich von

der JDIDebugTarget-Instanz verwaltet wird [9]. JDIDebugTarget delegiert nun die Verarbeitung der EventQueue des JDI-back-ends an ein EventDispatcher Objekt, welches über eine addJDIEvent-Listener() Registrierungsmethode JDI-EventRequest-Erzeugern JDI-Events mitteilt. Dabei werden die JDI-EventRequests von den implementierenden Klassen des Java Debug Models über die Fabrikmethoden des JDI EventRequestManagers erzeugt, der wiederum aus der im JDIDebugTarget gespeicherten VirtualMachine Referenz über den Aufruf eventRequestManager() erhalten wird.

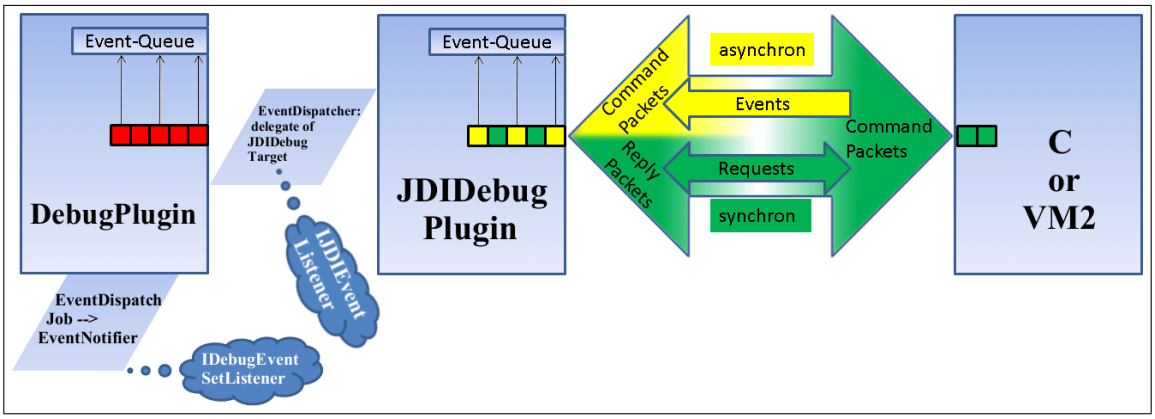

Abbildung 2.9: Zusammenspiel von Eclipse und JDI

2.8.2 JDI Request- und Eventhandling

Am Beispiel einer Breakpoint-Haltesituation eines Threads möchte ich die Benutzung des JDI-back-enddurch das Java Debug Model back-end demonstrieren.

Threadsuspendierung in einem JDI-front-endDebugger

1. JDIDebugTarget wird instantiiert und erhält im Konstruktor die VirtualMachine Instanz.
2. JDIDebugTarget besitzt EventDispatcher, an den er die JDIEvent-Dispatch-Aufgabe delegiert. Dazu startet JDIDebugTarget einen Daemon Thread in dem der EventDispatcher als Runnable ausgeführt wird. Dieser liest darin die EventQueue der Virtualmachine blockierend aus und leitet die JDI Events an die registrierten JDIEventListener weiter.
3. JavaBreakpoint registriert sich im JDIDebugTarget als JDIEventListener für ein erstelltes ClassPrepareRequest.
4. Die VirtualMachine startet und kommt zu dem Punkt, wo es die Klasse in den Speicher lädt, welche den Breakpoint enthalten soll.
5. Die JavaBreakpoint.handleEvent() Methode wird vom EventDispatcher aufgerufen und installiert mittels JavaBreakpoint.handleClassPrepareEvent() den eigentlichen JDI BreakpointRequest im JDIDebugTarget. Diese Installation bedingt wiederum die Registrierung des JavaBreakpoints als dessen JDIEventListener.

[9] erkennt man z.B. daran, dass getVM() in JDIDebugElement durch Delegation an getVM() vom JDIDebugTarget implementiert ist

6. Ein Thread kommt zu der Codestelle, an welcher der JavaBreakpoint liegt und suspendiert. Die VirtualMachine liefert diesbzgl. ein EventSet zurück, welches alle Events enthält, die der Grund für die Suspendierung waren.
7. Der EventDispatcher leitet alle Events aus der EventSet-Menge an die zugehörigen JDIEvent-Listener, welche in einer einstimmigen Wahl das Forsetzen des Threads bestimmen können.
8. Das JavaBreakpoint Objekt stimmt mit einem „Nein“ ab.
9. In EventDispatcher wird das Abstimmungsergebnis allen JDIEventListenern durch Aufruf von JDIEventListener.eventSetComplete() bekanntgegeben.
10. JavaBreakpoint ermittelt darin über Abfrage der JDI ThreadReference aus dem JDI Event Objekt das zugehörige JDIThread Model und ruft JDIThread.completeBreakpointHandling() auf.
11. JDIThread erzeugt ein Event Objekt auf der Abstraktionsebene des ESDM mit seiner eigenen Referenz als Quelle und legt dieses org.eclipse.debug.core.DebugEvent Objekt entweder in einer Queue des EventDispatcher Objektes ab oder feuert dieses direkt ab.
12. In beiden Fällen wird das DebugEvent Objekt im letzten Schritt mit DebugPlugin.fireDebugEventSet() der zentralen Instanz des debug.core Plug-Ins zugeleitet, welches schließlich seinen eigenen DebugEvent-Dispatcher besitzt. Dementsprechend sind die zugehörigen Listener hierfür vom Typ IDebugEventSetListener. Es findet also in diesem letzten Schritt ein Wechsel der Abstraktionsschicht in der Eventverarbeitung statt.

2.9 Anforderungen an einen Multi-User Debugger

An ein durch mehrere Benutzer gleichzeitig bedienbares Anwendungsprogramm werden bestimmte Anforderungen gestellt, welche in gängiger Literatur zu „computer-supported-cooperative work“ beschrieben werden und sich auf die Präsentation und die Steuerung des Zugriffes auf den gemeinsamen Datenbestand beziehen [7, p. 83]. Folgende Kriterien sollten beim Verteilten Debuggen angesetzt werden:

- Individuelle Sichten (views) [7, S. 83]. Jeder Benutzer sollte unabhängig von anderen Benutzern seinen Debugger front-endgestalten können. Dies trifft z.B. für die zentrale Intention des verteilten Debuggings zu, dass sich einzelne Benutzer um die getrennte Steuerung von parallelen Threads kümmern. Diese Forderung impliziert die Trennung von GUI- und Applikationslogik.
- Verbindungsauf und -abbau mit dem Programm [3, S. 4]. Diese Aufgabe wird durch die Sessionfunktion von Saros übernommen.
- Multiplexing von Benutzereingaben [3, p. 4]. Aktivitäten aus der Bedienung der für jeden Benutzer getrennt vorhandenen Front-ends des Debuggers müßen vom Multi-User Debugger back-end gleichzeitig verarbeitet werden können.
- Demultiplexing von Programmausgaben an die einzelnen Benutzer [3, p. 4]. Dies geschieht einerseits in der Form von Unicast-Weiterleitung bei Variablenwertmitteilung und andererseits in der Form von Broadcast-Weiterleitung von Stdout und Stderr Streams und asynchronem Eventdispatching.
- Parallelitätssteuerung: verhindern von in Konflikt stehenden Eingabeszenarien [3, p. 4]. Es

muss verhindert werden, dass ein Benutzer einen Breakpoint löscht während ein anderer diese disabled, beide ein Step-Kommando auf dem selben Thread initiieren oder zwei Benutzer den Wert derselben Variablen zu ändern versuchen.

- Zugriffssteuerung - einzelne Funktionen des Programmes sollen nur bestimmten Benutzern zugänglich sein [3, p. 4]. In [7, p.83] als Flußkontrolle (floor control) bezeichnet. Dies ist beim gemeinsamen Debuggen z.B. für das Starten des Debuggers angezeigt. Der Grund liegt in der notwendigen Konfiguration der Startparameter, die nur vom zentralen Benutzer des Debuggers bestimmt werden können. Alle anderen Funktionen eines Single-User Debuggers sind wiederum gleichberechtigt benutzbar.
- Kopplung - andere Benutzer über Eingabevorgänge informieren [3, S. 4]. Alle Benutzer verfügen über eine konsistente Menge von Breakpoints und besitzen einen konsistenten Informationsstand während des Debuggingvorgangs.
- Unterstützen von Nachzüglern (latecomers). Neue Teilnehmer müssen schnell in eine laufende Sitzung aufgenommen und die Sitzung mit einem für alle Teilnehmer konsistentem Zustand fortgesetzt werden können [7, S. 83]. In unserem Fall soll das nur für die Pre-Launch Phase des Debuggers gelten, da in der Launch-Phase bereits viele Debug-Events vom zentralen Benutzer ausgesandt worden und damit verworfen sind und nicht ohne großen Aufwand eine Konsistenz mit dem neuen Teilnehmer erreicht werden kann.
- Awareness: Um Zusammenarbeit zu unterstützen, muß es Verfahren geben, mit dem Benutzer über Aktivitäten anderer Benutzer informiert gehalten werden. Nur so kann man eigene Aktionen davon ableiten [7, S. 83]. Dies kann man im deutschen etwa als „im Klaren sein“-Eigenschaft übersetzen. Beim Verteilten Debugger sind das z.B. in der Launch-Phase vorhandene Informationen darüber, auf welcher Codezeile ein Remoteuser ein Codestepping ausführt bzw. man kann sich auch überlegen, ob man in erstellten Breakpoints noch die Information über den ersteller mitführen möchte.

2.10 Lösungsmöglichkeiten zur Implementierung eines Multi-User Java-Debuggers in Eclipse

Betrachtet man das Zusammenspiel der verschiedenen Abstraktionsebenen im Eclipse Debugger für Java, so erkennen wir folgende Eingriffsmöglichkeiten, um diesen Single-User-Debugger multiuser fähig zu machen.

1. Wir greifen in die JDWP-Kommunikation des JDI back-ends ein.

   Die Idee hierbei ist, dass man die Socketverbindung zum Debuggee eingabemultiplext und ausgabedemultiplext. Ein zentraler Koordinator, in der Rolle des Hosts von Saros, besitzt hierbei eine modifizierte Socketverbindung mit erweiterter Intelligenz. Diese besteht darin Eingaben in den Socket so zu verarbeiten, dass es die JDWP-Kommandos protokolliert und nur dann an die Debuggee-Gegenstelle weiterleitet, wenn es im Kontext eines single-user Debuggings steht. Dieser Kontext gewährleistet, dass JDWP-Kommandos, die nicht wiederholt ausgeführt werden dürfen – wie z.B. das Erstellen eines EventRequests in der VirtualMachine, auf das man eine Antwort in Form der erstellten requestID bekommt – auch nur einmal weitergeleitet werden. Nachfolgende gleiche JDWP-Kommandos werden dann verworfen und der Rückgabewert auf das Kommando aus dem Cache bedient. Andere Kommandos, wie z.B. die

Abfrage eines Variablenwertes, können hingegen wiederholt durchgeführt werden und sollten deswegen gecached werden, um eine bessere Skalierung der Benutzerzahl zu erhalten. Die Aufgabe beim Ausgabedemultiplexen besteht darin, JDWP-Antwortpakete nur an den jeweiligen Anfrager und JDWP-Eventpakete als Broadcast an alle Benutzer weiterzuleiten. Im ersten Fall muss sich der intelligente Host-Socket also merken, wer das JDWP-Kommando in den Socket geschrieben hat, damit er ein mögliches Antwortpacket an diesen durchleiten kann.

2. Wir greifen in die JDI back-end Kommunikation ein.

   Dabei kommen die gleichen Kriterien für einmal- oder mehrfach-wiederholbare Befehle zum Tragen wie beim JDWP-Eingriff. Das JDI-back-endmuss auf Hostseite ein Caching für auf ihr aufgerufene Methoden betreiben, damit jeder Benutzer das gleiche JDI EventRequest generiert. Dies ist nötig, weil das JDI EventRequest in einem nachfolgenden Zeitpunkt zu einem JDI Event führt, das die Information des ihn verursachenden JDI EventRequests über die Methode Event.request() mitführt. Diese Information wird dabei von der Klasse EventDispatcher benutzt, um die eingehenden JDI Events den JDI EventRequest-Erzeugern zwecks Verarbeitung zuzustellen. Die Klasse EventDispatcher ist dabei ein Delegate von der Klasse JDIDebugTarget, welches als der Repräsentant des Eclipse Java Debug Models aufgefasst werden kann. Mit dieser Information im Hinterkopf wird auch der Grund für das nötige Caching von JDI EventRequests deutlich. Wenn ein EventRequest für einen Remote-User vom Host-JDIback-enderzeugt wird, dann wird dieses im Anschluß serialisert und an den Remote-User übertragen. Dieser empfängt ihn und registriert das remoteanfrage-verursachende Objekt - welches die JDIEventListener Schnittstelle implementieren muss - als JDI Eventhandler für das deserialisierte JDI EventRequest Objekt. Der EventDispatcher vom Host muss also nur noch die Events bzw. die EventSet-Menge an die EventDispatcher der weiteren Debugteilnehmer senden, so dass das Dispatching des Hosts bei den anderen Teilnehmern repliziert stattfinden kann.

3. Wir greifen in den Eclipse Java Debugger front-endein.

   JDI-Events werden in der Klasse EventDispatcher den zugehörigen JDIEventlistenern zugestellt und diese erzeugen unter bestimmten Kriterien daraus neue Events auf einer höheren Abstraktionsebene. Diese Ebene bezieht sich auf Objekte des Eclipse Java Standard Debug Models als Quelle ihrer Entstehung und werden an den Activator des eclipse debug.core Plug-Ins übergeben. Dieser Activator ist ein Singleton-Objekt der Klasse DebugPlugin und besitzt einen eigenen Dispatcher für Events vom Typ DebugEvent. An diesem Objekt registrieren sich viele Objekte, die für das grafische Eclipse Debugger front-endverantwortlich sind, aber auch das JDIDebugTarget Objekt. Sie sind somit die Observer des DebugPlugins und der mit ihr assoziierten Datenstrukturen, wie z.B. den BreakpointManager, welcher die vorhandenen Breakpoints verwaltet. Der BreakpointManager ist unter anderem auch der Grund, warum JDIDebugTarget ein Observer des DebugPlugins ist. Die GUI des Eclipse Debugger front-ends aktualisiert sich also durch das Eventdispatching im DebugPlugin, wobei die Events vom Typ DebugEvent sind und diese als Quellen die EJDM Objekte enthalten. Die Basisklasse von DebugEvent ist zwar serialiserbar, jedoch wird die Quelle als transient gekennzeichnet und würde bei einer Serialisierung nicht berücksichtigt. Somit ist es nicht möglich den Weg zu gehen, dass man Events auf Ebene des Eclipse Debugger front-ends benutzt und diese unter den Benutzern des Multi-User Debuggers austauscht. Auch wäre bei diesem Ansatz jedes empfangene EJDM-Objekt auf Empfängerseite ein remote zu benutzendes Objekt,

weil diese Elemente von der GUI-Schicht in der DebugEvent Verarbeitung hierarchisch aufgerufen werden. Somit müsste eine Ersetzung der Objekte des EJDM durch entsprechende Proxyobjekte stattfinden bevor der Host diese in der Dispatchingroutine seines DebugPlugins an die anderen Teilnehmer aussendet. Bei diesem Ansatz ergibt sich damit auch das Problem, dass die Methoden der EJDM-Objekte für den parallen Zugriff angepasst werden müssten und sich dies bei Remotezugriff äußerst ungünstig auf die Reagibilität des Eclipse Debugger Frontends auswirken kann.

2.11 Auswahlentscheidung

Der Vorteil des Eingriffs auf JDWP-Ebene ist, dass es relativ einfach zu installieren ist. Installation meint hier die Einbindung in das bestehende Codegesrüst von Eclipse. Das Prinzip ähnelt einem Netzwerksniffer oder Webproxy, welcher auch auf lokalen PCs eingesetzt werden kann, um z.B. Werbung auszufiltern oder Webseiten für Minderjährige zu sperren. Hier leitet sich der Gedanke auch von der Idee der Ein-/ Ausgabeumleitung in Unixshells ab, bei denen man auch ein sogenanntes `tee`-Kommando in eine Pipeline zwischenschalten kann. Der Nachteil an diesem Ansatz ist allerdings die komplizierte Implementierung, die sich dadurch ergibt, dass man das JDWP-Protokoll lernen muss, welches auf einer niedrigen Abstraktionsebene arbeitet und viel Spezialwissen über die Java Virtual Machine erfordert. Außerdem sind die Grundgedanken über Ein- und Mehrfachausführbarkeit von Debuggerbefehlen nicht verschieden zum Lösungsansatz auf JDI-Ebene.

Der Vorteil bei Eingriff in die JDI-Backendkommunikation ist die reine Betrachtung von Java-Methoden, welche hinsichtlich Caching-Gedanken zu untersuchen sind. Hier müssen wir uns überlegen wo und wie wir den JDI-Backend vom EJDM unterscheiden und trennen können. Letzteres ist damit zu erklären, dass zum einen der Host seinen JDI-Backend multi-frontend fähig machen muss und zum anderen die Peers einen Proxy JDI-back-endbenutzen müßen, der mit dem JDI-back-enddes Hosts kommuniziert. In diesem Sinne müssen wir das Verhalten von RPC-Funktionen [10] mit der Einhaltung des Kontexts eines Single-User Zugriffs auf das Host JDI-back-endkombinieren.

Ein Eingriff in den Eclipse Debugger Frontend bietet, wie oben geschildert, nur Nachteile und kann damit als Kanditat ausgeschlossen werden.

Nachfolgend behandelt die vorliegende Arbeit daher den Eingriff in die JDI-Backendkommunikation.

[10] RPC = Remote Procedure Call

KAPITEL 3

Implementierung des Multi-User Debuggers auf JDI back-end Ebene

3.1 Lastenheft

1. Debugging besteht aus den zwei Zuständen Pre-Launch und Launch. Der Pre-Launch Zustand besteht aus dem Zeitraum einer Sarossitzung, in der der Debugger nicht gestartet ist. In diesem Zeitraum können nur Break- und Watchpoints manipuliert werden. Im Launched Zustand werden diese Möglichkeiten um Debugger Steuerungsfunktionen, wie z.B. Step, Resume usw., ergänzt.
2. Aktivitäten auf Break- und Watchpoints sollen unter allen Debug-Teilnehmern ausgetauscht und synchronisiert werden.
3. Neue Debuggingteilnehmer können nur im Pre-Launch Zustand hinzugefügt werden.
4. Es findet nur eine Unterscheidung in Host und Peer fürs Debuggen statt. Die Rollen Driver und Observer von Saros werden ignoriert.
5. Der Host in einer Saros-Session ist allein berechtigt einen neuen Teilnehmer einzuladen.
6. Einladung eines neuen Teilnehmers geht einher mit dem Stoppen aller Sarosaktivitäten der bisherigen Teilnehmer. In diesem Zustand erfolgt das Übertragen aller Breakpoints vom Host an den Peer. Der Host gibt nach Einladungsbeendigung alle Aktivitäten wieder frei.
7. Konsistenzwiederherstellung mit dem BreakpointManager des Hosts bedeutet das Löschen aller im lokalen BreakpointManager vorhandenen Breakpoints und anschließendes Einfügen der vom Host erhaltenen Breakpoints.
8. Der Host ist der alleinige Initiator eines Debug-Launches, aufgrund der hierfür erforderlichen Informationen bzgl. der Systemumgebung und der Startparamter. Hierbei kommt es zu einer sichtbaren Trennung von privaten Informationen, die geschützt werden müssen und von öffentlichen Informationen, die kommuniziert werden sollen (Awareness, Kollaboration) [7, p. 83].
9. Die Standardeingabe des Debuggee Programms soll vorläufig nicht mit Debuggingteilneh-

mern geteilt werden und bleibt beim Host.

10. Die Standardausgabe und die Fehlerausgabe sollen allen Teilnehmern zur Verfügung stehen.
11. Alle weiteren Aktivitäten bzgl. der Benutzbarkeit des Debuggers sollen allen Teilnehmern gleichberechtigt zur Verfügung stehen.
12. Die Erstellung und Manipulation von Break- und Watchpoints soll ohne Sperren auskommen.
13. Wenn ein Teilnehmer eine Session verläßt, behält er seine Daten vom lokalen BreakpointManager. So kann dieser ein lokales Debugging mit diesen durchführen.

3.2 Assoziation von Breakpoints zwischen Host und Peers

Bei einer replizierten Verwaltung von Daten muss man sich Gedanken um eine entsprechende Assoziation des zentralen Datenbestands mit denen des replizierten Datenbestandes machen. In [3, S. 7] werden replizierte Daten als „interaction variable“ bezeichnet, mit der Eigenschaft, dass diese sich automatisch mit den zentralen „active variable“ synchronisieren. Wir müssen hingegen eine manuelle Synchronisation durchführen und schicken dazu über einen SharingBreakpointListener Eventereignisse vom lokalen BreakpointManager an exterene Teilnehmer ab. Wenn wir nun bei der Erstellung eines Breakpoints diesen serialisieren, dann kann der Empfänger dieses Breakpoints diesen nicht einfach deserialisieren und in seinen BreakpointManager einfügen. Der Grund liegt an der 1:1 Zuordnung eines Breakpointobjekts zu einem Markerobjekt, welches Metainformationen für eine Datei im Dateisystem speichert. Das Markerkobjekt besitzt eine ID, welche in der assoziierten Datei des Dateisystems eindeutig ist. Wenn wir nun das deserialisierte Markerobjekt auf der Empfangsseite einem erstellten Breakpoint zuweisen [1], dann ist die ID vielleicht schon von einem anderen Marker in Benutzung und dies führt zu einem Systemfehler. Die ID aus dem Marker kann also nur zur lokalen Identifizierung eines Breakpoints benutzt werden und das nur in Kombination mit der Information der zugehörigen Quellcodedatei. Eine externe Assoziation von Breakpoints des Host und derjenigen von Peers – Peers kommunizieren nie untereinander, sondern immer nur Host zu Peer – geschieht auf Seiten der Peers, wenn diese eine ADD-Nachricht vom Host erhalten.

```
1   /**
2    * Peer-Code: Process command to add breakpoint. If peer was
       requestor,
3    * this will be handled as acknowledgment. Always process
       messages from
4    * host regardless of beeing blocked or not by StopManager
5    */
6   @Override
7   public boolean consume(
8       BreakpointResponseActivity breakpointResponseActivity) {
9       ....
10          if (breakpointResponseActivity.getSource().equals(
                current)
11          && echoBkpNmbr != NIL) {
12          // the following is already done in
```

[1] Abschnitt 3.6

```
13          // SharingBreakpointListener.breakpointAdded(
                IBreakpoint):
14          // id2Bkp.put(bp);
15          boolean removed = wait4confirm.remove(file,
16              BreakpointRequest.ADD) != null;
17          if (removed) {
18              log.warn("got confirm from host");
19              createLocal2HostMapping_4_newLocalBreakpoint(file,
20                  echoBkpNmbr, breakpointResponseActivity.
                        hostBkpNmbr);
21              updateRevisionTupel(newHostRevision, revisionTupel)
                    ;
22          } else {
23              log.fatal("Acknowledged Breakpoint was not in set
                    of waiting ones.");
24          }
25          ....
```

Quellcode 3.1: Assoziation von Host und Peer Breakpoints

3.3 Konfliktsituationen im BreakpointManager

Der Host als zentrale Kontrollinstanz im verteilten Debuggingframework übernimmt eine Koordinierungs- und Verteilungsfunktion hinsichtlich des Datenbestandes an gemeinsamen Breakpoints. Der Host verwaltet ihn in seinem lokalen BreakpointManager aus dem DebugPlugin. Jede Benutzeraktion wie

- Erstellung
- Löschung
- und Veränderung

von Breakpoints landet im jeweiligen lokalen BreakpointManager und wird anschließend an den Host zwecks Bestätigung geschickt. Dieser verarbeitet die Ereignisse im Kontext einer verteilten Applikation und teilt das Resultat allen Teilnehmern per Broadcast mit. Das Ergebnis der Verarbeitung der Ereignisse muss ein konsistenter Endzustand bei allen Debuggingteilnehmern sein.

In Tabelle 3.1 werden die möglichen Konfliktsituationen mit ihren Lösungsstrategien dargestellt.

In der konkreten Implementierung ist ein Modify Ereignis nur in der Art „toggle Breakpoint“ implementiert worden und daher sieht die Konflikttabelle für die spezielle Implementierung wie in Tabelle 3.2 abgebildet aus.

Einer positiven Bestätigung durch den Host folgt das Löschen aus der „vorläufig“ Menge der Aktivitäten des Peers und einer negativen Bestätigung die Revidierung der Aktivität durch eine Umkehroperation.

Vorereignis	Ereignis		
Gemeinsame Konfliktvorbedingung: Gleicher Typ und gleiche Zeile bzgl. des Breakpoints der Ereignisse			
	ADD	REMOVE	MODIFY
ADD	Ereignis ablehnen.	Kann nicht vorkommen bei Konsistenz der Teilnehmer	Kann nicht vorkommen bei Konsistenz der Teilnehmer
REMOVE	Kann nicht vorkommen bei Konsistenz der Teilnehmer	Ereignis ignorieren.	Ereignis ignorieren.
MODIFY	Kann nicht vorkommen bei Konsistenz der Teilnehmer	Ereignis ablehnen.	Ereignis ablehnen.

Tabelle 3.1: Konfliktsituationen bei parallel erzeugten Breakpointaktivitäten

Vorereignis	Ereignis		
Gemeinsame Konfliktvorbedingung: Gleicher Typ und gleiche Zeile bzgl. des Breakpoints der Ereignisse			
	ADD	REMOVE	ToggleEnable
ADD	Ereignis ablehnen.	Kann nicht vorkommen bei Konsistenz der Teilnehmer	Kann nicht vorkommen bei Konsistenz der Teilnehmer
REMOVE	Kann nicht vorkommen bei Konsistenz der Teilnehmer	Ereignis ignorieren.	Ereignis ignorieren.
ToggleEnable	Kann nicht vorkommen bei Konsistenz der Teilnehmer	Ereignis ablehnen.	Ereignis ignorieren.

Tabelle 3.2: Konfliktsituationen bei MODIFY = ToggleEnable

Mit einem Anteil von 4/9 aller Paarungen sind Konstellationen belegt, welche nur bei einer Inkonsistenz der Teilnehmer entstehen können. Sie sind daher bei einer zuverlässigen Kommunikationsverbindung unerwartet. Nichtsdestotrotz kann man darauf mit einer Inkonsistenzmitteilung an den Peer reagieren, der daraufhin eine Resynchronisation durchführen kann.

Ein Drittel [2] aller Konfliktsituationen sind vom Charakter „obsolete“, was bedeuten soll, dass eine Bestätigung der Aktivität nicht mehr erforderlich ist und das Ereignis vom Host ignoriert werden kann. Als Beispiel nehmen wir die Konstellation von zwei parallel ausgeführten Löschoperationen auf demgleichen Breakpoint durch die Benutzer Alice und Bob. Wenn Alice zuerst mit seiner Nachricht vom Host verarbeitet wird, dann sendet dieser per Broadcast die Löschaufforderung an alle Teilnehmer. Alice hatte die Aktion bereits durchgeführt und entfernt dementsprechend nur noch ihren Eintrag aus der „vorläufig“-Liste. Bob empfängt dieselbe Nachricht vom Host und betrachtet diese ebenso wie Alice als Antwort auf seine Anfrage. Er handelt dementsprechend genauso wie Alice. Alle anderen Teilnehmer führen den Löschbefehl real durch. Im Ergebnis dieses Verhaltens

[2] bei implementiertem MODIFY=ToggleEnable

ist die Anfrage von Bob beim Host zum Zeitpunkt ihrer Verarbeitung obsolete (nichtig) geworden.

Mit einem Anteil von 2/9 muss ein Ereignis abgelehnt werden. Im Fall von ADD fragt man sich vielleicht, warum das nicht auch einen „obsolete“-Charakter hat. Dies liegt darin begründet, wie Breakpoints auf Host- und Peer-Seite miteinander assoziiert werden. Wenn ein Peer noch keine Bestätigung für seine ADD-Aktion bekommen hat, dann gibt es noch keine Assoziation für den betreffenden Breakpoint. Also verarbeitet der Peer die Nachricht vom Host eines ADD-Vorgangs unabhängig davon, ob es mit einem lokal erstellten Breakpoint kollidiert. In dieser Zeit besitzt der Peer – der Host kommt niemals in diese Situation – zwei Breakpoints die sich überdecken. Man kann die Breakpoints aber in der BreakpointView von Eclipse als Einträge auf derselben Zeile ablesen. Wenn dann im Anschluss die Ablehnungsantwort vom Host beim Peer eintrifft, dann entfernt der Peer seinen „doppelten“ Eintrag aus dem BreakpointManager und ist wieder Konsistent mit dem Host. Beim REMOVE-Ereignis besteht die Assoziation mit dem Host bereits und man könnte fordern, dass ein REMOVE nicht abgelehnt werden müsste, da es im Anschluss an ein MODIFY genau den gleichen Effekt hat wie ohne. Der Grund für die Dominanz des MODIFY gegenüber dem REMOVE-Ereignis liegt in der strikten Anwendung des Prinzips, dass jedes Ereignis von den Benutzern „gesehen“ werden soll. Damit wird das Prinzip der Intentionserhaltung verfolgt. Intentionserhaltung bedeutet an diesem konkreten Beispiel, dass der löschende Benutzer seine Entscheidung aus dem veralteten Wissenstand über den Breakpoint gefällt hat und seine Entscheidung bei Kenntnis des aktuellen Zustands evtl. anders ausfallen würde. Zum Beispiel könnte sich Bob nicht zum Löschen des Breakpoints x entscheiden, wenn er wüsste, dass Alice Breakpoint x gerade deaktiviert hat. Der Aspekt der Intentionserhaltung ist eine Komponente von Konsistenzkriterien die von Ziller [18] beschrieben werden.

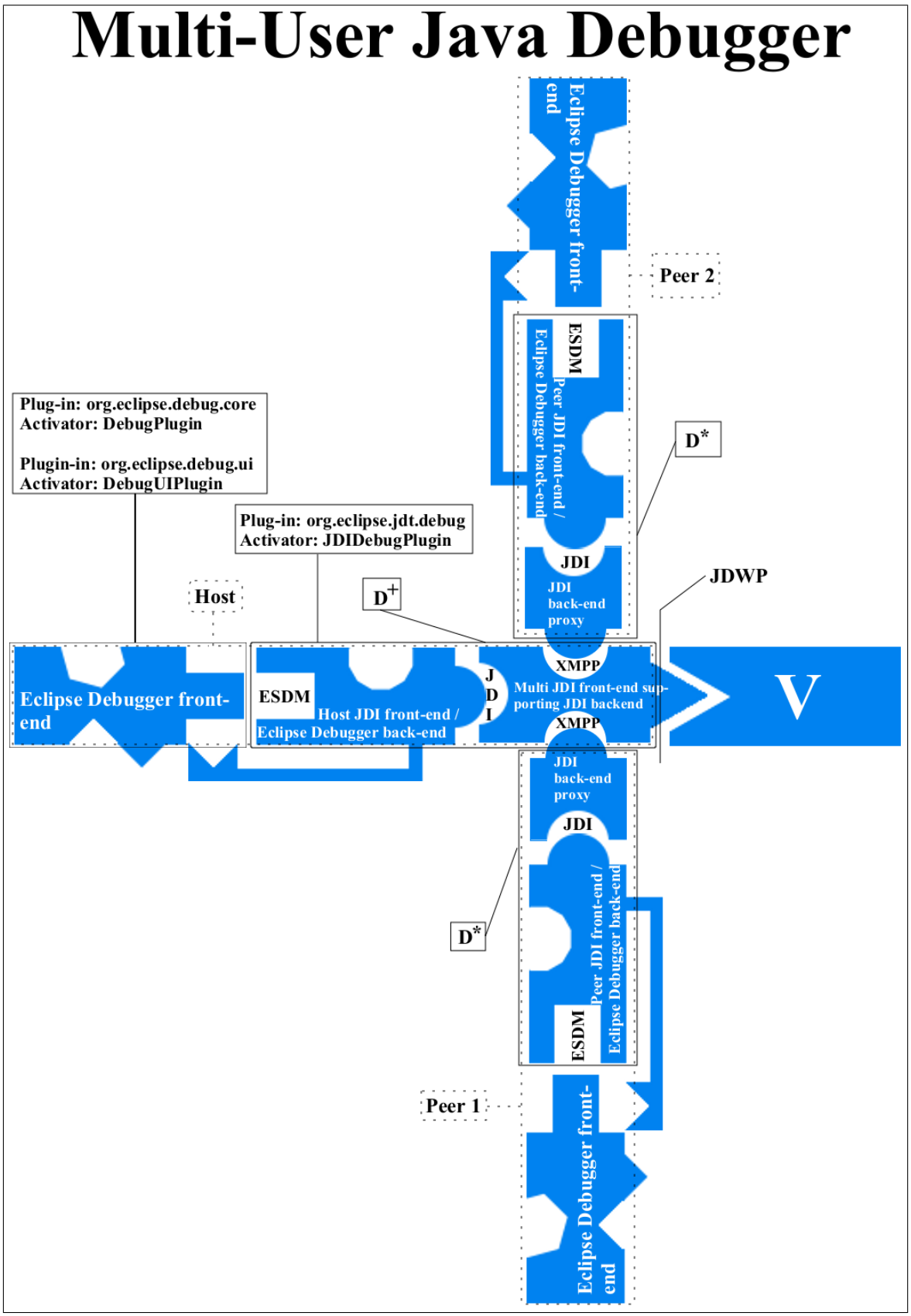

Abbildung 3.1: Eingriff über die JDI Schnittstelle

3.4 Koordinierung der Verarbeitung von lokalen und remote Ereignissen

Sowohl der Host als auch die Peers erzeugen durch ihre GUIs Breakpointereignisse und teilen sich diese über einen Nachrichtenkanal mit. Im Nachfolgenden wird die Verarbeitungsreihenfolge in einem Szenario ohne Konflikte gezeigt.

3.4.1 Peer initiiert Breakpointereignis

1. Peer führt über seine GUI eine Aktion durch, die zu einer Veränderung im Datenbestand des lokalen BreakpointManagers führt.
2. BreakpointManager generiert daraufhin ein Breakpointereignis und teilt dieses an alle registrierten IBreakpointListener mit.
3. `SharingBreakpointListener` des Peers sendet Ereignis an Host damit dieser es bestätigt.
4. Host empfängt Ereignis vom Peer und führt dieses auf lokalem BreakpointManager aus.
5. Host führt Broadcast des Ereignisses durch.
6. Anfragender Peer empfängt Ereignis vom Host und betrachtet dies als Bestätigung. Peer entfernt daraufhin den Breakpoint des Ereignisses aus seiner Warteliste.
7. Andere Peers empfangen Broadcast vom Host und führen Ereignis auf lokalem BreakpointManager aus.

3.4.2 Host initiiert Breakpointereignis

1. Host führt über seine GUI eine Aktion durch, die zu einer Veränderung im Datenbestand des lokalen BreakpointManagers führt.
2. BreakpointManager generiert daraufhin ein Breakpointereignis und teilt dieses an alle registrierten IBreakpointListener mit.
3. `SharingBreakpointListener` des Host sendet Broadcast mit Ereignis an alle Peers.
4. Peers empfangen Broadcast vom Host und führen Ereignis auf lokalem BreakpointManager aus.

An diesen Abläufen erkennt man, dass der `SharingBreakpointListener` als Nachrichtensender für lokale Breakpointereignisse genutzt wird. Er wird vom BreakpointManager bei entsprechenden Änderungen an durch ihn verwalteten Breakpoints aktiv. Wir müssen daher verhindern, dass die Verarbeitung von empfangenen Breakpointereignissen zu einer erneuten Aussendung von Ereignissen führt und damit eine Kettenreaktion entsteht. Um diese Situation zu verhindern nutze ich die Tatsache aus, dass der BreakpointManager die Eventerzeugung und das anschließende Dispatching im gleichen Thread ausführt, wenn es sich um ADD oder REMOVE Aktionen handelt. Bei einer MODIFY Aktion ist dies nicht der Fall. Das ist allerdings ganz einfach zu erklären, wenn man sich vor Augen führt, dass empfangene ADD- und REMOVE-Events in Aktionen umgesetzt werden, die als Aufrufe von Methoden des BreakpointManagers in Erscheinung treten. Bei einem MODIFY Ereignis geschieht die Umsetzung in eine Aktion über einen Methodenaufruf auf dem Breakpointobjekt. Der BreakpointManager kriegt hierbei die Aktion also nicht unmittelbar mit. Für

die ersten beiden Arten von Events kann man daher folgende Strategie wählen.

```
...
processingBP = bp;
breakpointManager.addBreakpoint(bp);
processingBP = null;
...

public void breakpointAdded(IBreakpoint breakpoint) {
  if (feature_disabled)
    return;
  setLocalComputingFlag();
  if (violates_pre_cond(breakpoint)) {
      return;
  }
  ...
}

/**
 * Check if the reason of listener notification is the
    processing of a
 * receiving message from outside (not UI triggered).
 *
 * @param breakpoint
 *            The breakpoint which is affected.
 * @return True if listener should be skipped for the
    breakpoint, to
 *         prevent cycles in outsending messages.
 */
protected boolean violates_pre_cond(IBreakpoint breakpoint) {
    if (breakpoint == processingBP
        || sarosSession == null
        || !sarosSession.isShared(breakpoint.getMarker().
           getResource()
            .getProject())) {
        clearLocalComputingFlag();
        return true;
    } else {
        return false;
    }

}
```

Quellcode 3.2: Zyklusvermeidung bei Nachrichtenempfang ADD REMOVE

Für die Situation eines MODIFY Events habe ich diese Strategie gewählt.

```
...
```

```
protected void setEnableStat(IBreakpoint breakpoint, boolean
    doEnable) {
  processingBP = breakpoint;
  try {
    breakpoint.setEnabled(doEnable);
  } catch (CoreException e) {}
}
...
public void breakpointChanged(IBreakpoint breakpoint,
    IMarkerDelta delta) {
  if (feature_disabled)
    return;
    setLocalComputingFlag();
    if (violates_pre_cond(breakpoint)) {
      // this is special for breakpointChange
      processingBP = null;
      return;
    }
    ...
```

Quellcode 3.3: Zyklusvermeidung bei Nachrichtenempfang MODIFY

3.5 Behandlung von Nebenläufigkeit

Dem Prinzip von Saros folgend sollen lokale Aktionen erst ausgeführt und dann ausgesendet werden. Auf diese Art wird Echtzeitverhalten der Anwendung erreicht und die Nutzerzufriendheit positiv beeinflusst. Wenn man nun dieses Verhalten bei jedem Teilnehmer voraussetzen kann, dann ist es möglich das Konflikte wie in 3.1 enstehen. Aufgrund der sofortigen lokalen Ausführung und dann erst erfolgenden Mitteilung der Aktivität können wir keine Sperrmechanismen anwenden, um eine Konvergenz in dieser replizierten Architektur zu gewährleisten.

3.5.1 Strategie orientiert an Versionskontrollsystemen

Beim Empfang einer Peer Nachricht prüft der Host zunächst, ob es bereits auf derselben Zeile und derselben Datei ein verarbeitetes Ereignis gibt, von dem der Peer zum Zeitpunkt seiner Ereignismitteilung noch nichts wusste. Dazu schickt jeder Peer eine dateibasierte Versionsnummer mit der Nachricht mit, die dem Host mitteilt auf welchem Informationsstand der Peer bzgl. der Datei war, als er eine Aktion auf dieser durchführte. Dieses Vorgehen bedient sich dem Prinzip von gängigen Versionskontrollsystemen, wie z.B. CVS. Wenn der Host nun die erhaltene Versionsnummer mit seiner Versionsnummer vergleicht, dann ist die Aktion genau dann durchzuführen, wenn diese beiden identisch sind. Im anderen Fall ist die empfangene Versionsnummer kleiner als die lokale und damit veraltet. Um nicht nun sofort eine Aktion ablehnen zu müssen, bloß weil der Peer in einem veralteten Wissenstand agierte, verwaltet der Host eine sogenannte Historie von Ereignissen, die alle auf einer bestimmten Dateiversion erzeugt wurden. Denn es kann parallele Aktionen auf derselben Datei geben, solange sich diese auf eine unterschiedliche Zeilennummer beziehen. So kann es z.B. drei gleichzeitige Breakpointerzeugungen und drei Löschungen geben, ohne dass

ein Teilnehmer von den parallel stattfinden Aktionen erfahren haben muss, solange sich diese auf verschiedene Zeilen und damit Breakpoints beziehen. Diese verwaltete Historie ist ein Hash, welcher über ein Tupel – bestehend aus der jeweiligen Datei und der Versionsnummer – geschlüsselt ist und einen Vektor mit den Breakpoints enthält, auf den sich die parallelen Aktivitäten beziehen. Der Host überprüft nun den Typ des empfangenen Breakpoints mit den Typen aller Breakpoints in der zugehörigen Historie der Dateiversion. Ein Konflikt kann nur enstehen wenn Typengleichheit mit einem Breakpoint aus der Historie besteht. Sollte es sich um JavaLineBreakpoint handeln wird zusätzlich auf gleiche Zeilennummern geprüft. Andernfalls wird nicht weiter differenzierend ein Konflikt angenommen und wie bei gleicher Zeilennummer eine Ablehnungsmeldung an den Absender der Aktion zurückgeschickt, auf den dieser mit einer Rücksetzung seiner lokal durchgeführten Aktion reagieren muss. Der Peer agiert auf diese Weise nach dem Muster von Optimistischer Synchronisation [9] aus dem Bereich der Datenbanktheorie, wonach angenommen wird, das Konflikte eher selten auftauchen und man Transaktionen zunächst auf einer lokalen Arbeitskopie durchführen kann und dann in einer Commit-Phase durch den zentralen Datenbankserver überprüft werden muss, ob es Konflikte mit parallellaufenden Transaktionen gab. Im Konfliktfall muss dann die konflikthafte Transaktion zurückgesetzt werden. Um einen solchen Konflikt auflösen zu können, muss man sich eines Prioritätskriteriums bedienen, anhand dessen die konfliktschuldige Transaktion selektiert werden kann. Eine Variante ist die Vergabe von Zeitstempeln an die Transaktion zum Zeitpunkt ihres Commit-Wunsches durch den Datenbankserver. Die Vergabe des Zeitstempels sowie die anschließende Validierungs- und Commitphase stellen atomare Vorgänge dar und sind damit strikt sequentielle Vorgänge ohne Nebenläufigkeit. In der Validierungsphase wird die Menge der Transaktionen ermittelt, die aufgrund ihres Zeitstempels als Konfliktkandidaten mit der aktuellen Transaktion überhaupt in Frage kommen und dann anschließend weitere Kriterien, wie z.B. die Ermittlung der Schnittemengen von gelesenen und geschriebenen Variablen untereinander, gebildet anhand derer ein potentieller Konflikt erkannt wird.

In unserem Fall entspricht die Verarbeitungsreihenfolge von empfangenen Nachrichten durch den Host einer impliziten Vergabe von Zeitstempeln. Die Atomarität ist durch die streng sequentielle Verarbeitung des sequentiell arbeitenden Activity Dispatch Threads gegeben. Der Validierungsphase entspricht der Vergleich mit den parallelen Aktivitäten aus der Host-verwalteten Historie. Dem Commit entspricht das lokale replizieren der empfangegenen Aktivität und Aussenden einer Bestätigungsmitteilung an den Absender. Dem Rollback entspricht eine Ablehnungsmitteilung an den Absender der Aktion zwecks Revidierung dessen lokaler Aktion. Was bei diesem Vergleich vielleicht nicht ganz einleuchtend ist, ist die fehlende Einbindung des als implizit klassifizierten Zeitstempels in unserer Validierungsphase. Diese tritt nur dadurch in Erscheinung, dass es sich bei der Historie um eine alternde Historie handelt, welche ab einer festgelegten Zeitspanne als abgelaufen markiert wird und dadurch als generelles Ablehnungsorgan für weitere eigentlich bereits veraltete Breakpoint-Nachrichten dient. In unserem Fall stellt die Vergabe der impliziten Zeitstempel zwar einen atomaren Vorgang dar, jedoch wird der jeweilige implizite Zeitstempel mehrfach vergeben bis die zugehörige Historie abgelaufen ist und dann keine neuen Elemente mehr aufgenommen werden.

```
1     private static final AgingVector<IBreakpoint> EXPIRED;
2
3     static {
4         EXPIRED = new AgingVector<IBreakpoint>();
5         EXPIRED.timestamp = System.currentTimeMillis() +
              AgingVector.MAX_AGE
6              + 1;
```

```
7       }
8
9       protected AutoHashMap<MyTupel, AgingVector<IBreakpoint>>
            revision2paralBkpActivities = new AutoHashMap<MyTupel,
            AgingVector<IBreakpoint>>(
10           new Function<MyTupel, AgingVector<IBreakpoint>>() {
11               public AgingVector<IBreakpoint> apply(MyTupel u) {
12                   return new AgingVector<IBreakpoint>();
13               }
14           });
```

Quellcode 3.4: BreakpointActivityProvider

3.6 Austausch von Breakpoints

Wenn man sich die Methoden der `IBreakpointListener` Schnittstelle anschaut, dann ist dort stets ein IBreakpoint Argument enthalten. Wenn wir nun beim verteilten Debugging einen „SharedBreakpointManager“ einführen, welcher von lokalen BreakpointManagern repliziert werden soll, dann müssen wir einen Mechanismus entwickeln, der die Änderungen an den jeweiligen lokalen BreakpointManagern an den zentralen „SharedBreakpointManager“ weiterleitet. Dieses Ziel können wir dadurch erreichen, dass wir einen SharingBreakpointListener vom Typ `IBreakpointListener` definieren, welcher auf Mitteilungen des lokalen BreakpointManagers mit einer Anfrage auf Bestätigung durch den Host reagiert. In dieser Anfrage muss zumindest für ein Erstellungsereignis der Breakpoint mit seinen relevanten Informationen über einen Nachrichtenkanal übertragen werden. Bei diesem Vorgang benutzen wir die Kommunikationsinfrastruktur von Saros, welche XMPP als Protokoll verwendet [4]. Um Objekte von Java übertragen zu können, bedient man sich der XStream Bibliothek. Eine Übertragung von Java-Objekten geht einher mit ihrer Serialiserung. Bei dieser Serialisierung muss eine geeignete Kodierung der Zustandsparameter des Laufzeitobjektes stattfinden. Diese Kodierung erfolgt bei XStream in der Gestalt von XML-Strukturen. Der XML-Text dieser Serialisierung kann nun vom XMPP-Protokoll als Nutzdaten an den jeweiligen Empfänger übertragen werden. Auf der Empfängerseite deserialisiert XStream die Nutzdaten wieder und es entsteht ein neues Java-Laufzeitobjekt in der JVM des Empfängers. Der Effekt dieses Vorgangs ist also ein „Remote-Cloning“. Im Gegensatz zu Java orientiert sich XStream nicht an einem Markerinterface wie Serializable, um Objekte als serialisierbar zu klassifizieren. Vielmehr gibt es das Konzept von Convertern – welche bereits für viele Klassen vordefiniert sind – die dazu benutzt werden Objekte von zugehörigen Klassen serialisieren zu lassen. Man kann eigene Converter definieren und dabei von bestehenden Convertern erben. Die jeweiligen Converter müssen sich beim `XStream`-Objekt registrieren, um aktiv zu werden: `xstream.registerConverter(aConverter)`. In der im Interface `ConverterMatcher` definierten `canConvert(Class)` Methode wird der Converter dann bzgl. seiner Fähigkeit zur Serialisierung eines Objektes abgefragt. Wenn dieser Suchmechanismus von XStream nicht zum Erfolg führt, dann springt ein auf Java-Reflection basierender Converter ein, der über Introspection geeignete Converter für die Feldelemente des diesbzgl. Objektes ermittelt. Für die Breakpointserialisierung habe ich einen Converter geschrieben, welcher vom `SerializableConverter` erbt. Dieser `SerializableConverter` emuliert den Serialisierungmechanismus von Java in XStream. Auf diese Weise kann man den gewohnten Mechanismus der Serialisierungssteuerung in Java anwenden, um eine Klasse serialisierbar zu machen, ohne

sich um XStream Prinzipien Gedanken machen zu müssen. Der Converter sieht folgendermaßen aus:

```
1  public class BreakpointActivityDataObjectConverter extends
2      SerializableConverter {
3
4      public BreakpointActivityDataObjectConverter(Mapper mapper,
5          ReflectionProvider reflectionProvider) {
6          super(mapper, reflectionProvider);
7      }
8
9      @Override
10     public boolean canConvert(Class type) {
11         return BreakpointRequestActivityDataObject.class.
               isAssignableFrom(type);
12     }
13
14 }
```

Quellcode 3.5: BreakpointActivityDataObjectConverter

Wenn wir nun einen Breakpoint serialiseren wollen, dann stellt sich die Frage wie wir das für die vielen Untertypen von IBreakpoint effizient erreichen können.

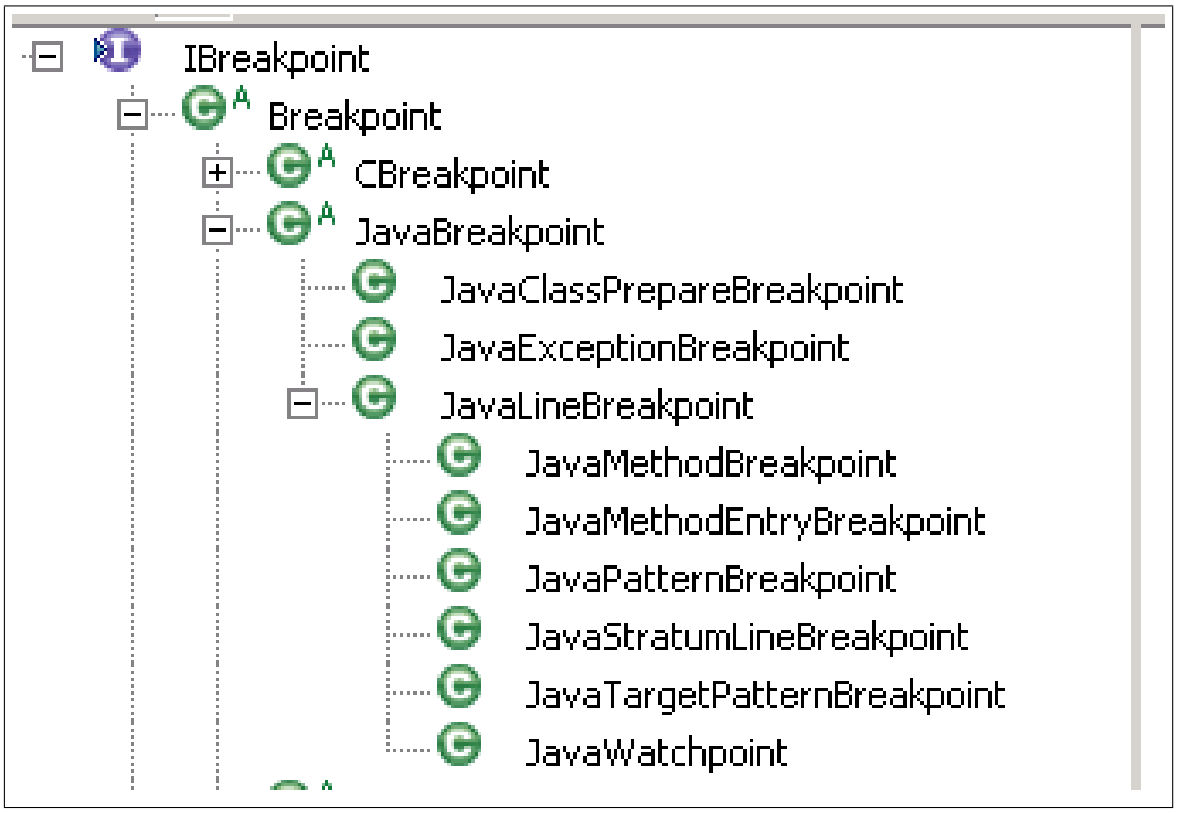

Abbildung 3.2: Hierarchie von IBreakpoint

Als Hinweis für eine Lösung dieses Problems betrachten wir den internen Mechanismus zur persistenten Speicherung von Breakpoints durch Eclipse. Breakpoints gehen beim Beenden von Eclipse nicht einfach verloren, sondern werden im Dateisystem gespeichert. Diese Speicherung erfolgt nicht etwa in den Quellcode-Dateien auf die sich die jeweiligen Breakpoints beziehen, sondern im sogenannten Configuration Area von Eclipse.

Genauso wie Projektinformationen als Metainformationen im Workspace von Eclipse gespeichert werden, werden auch die Breakpoints als Metadaten im Configuration Area von Eclipse gespeichert. Es muss also einen eclipse-internen Mechanismus geben, der diese Serialiserung in den Konfigurationsbereich und deren Deserialiserung aus diesem vollzieht. Wenn wir diesen Mechanismus kopieren, können wir die Suche nach relevanten Informationen für eine Serialisierung von Breakpoints beenden und sind damit fähig alle Unterypen von IBreakpoint zu serialisieren. Als Ergebnis dieser Suche habe ich die Funktion createBreakpoint(IMarker) 4.1 von Eclipse ausfindig gemacht. Die Eclipse Funktion fragt zunächst seinen Cache ab, ob ein Breakpoint für diesen Marker bereits existiert. Ist das nicht der Fall, dann muss ein „leeres“ Breakpoint-Objekt zunächst instantiiert und anschließend mit seinem persistenten Datenbestand – in Form des Marker-Objekts – gefüllt werden. Zur Instantiierung des korrekten Breakpoint-Typs nutzt die Funktion die Tatsache aus, daß einem Breakpoint-Typ [3] auch immer ein Marker-Typ zugeordnet 4.2 ist, in welchem die Eigenschaft von persistenter Speicherbarkeit des Breakpoints notiert ist. Außerdem sind einige Variablennamen von persitenten Daten dort eingetragen. Über diese 1:1 Zuordnung von Marker-Typ zu Breakpoint-Typ

[3] id Attribut der breakpoints extension

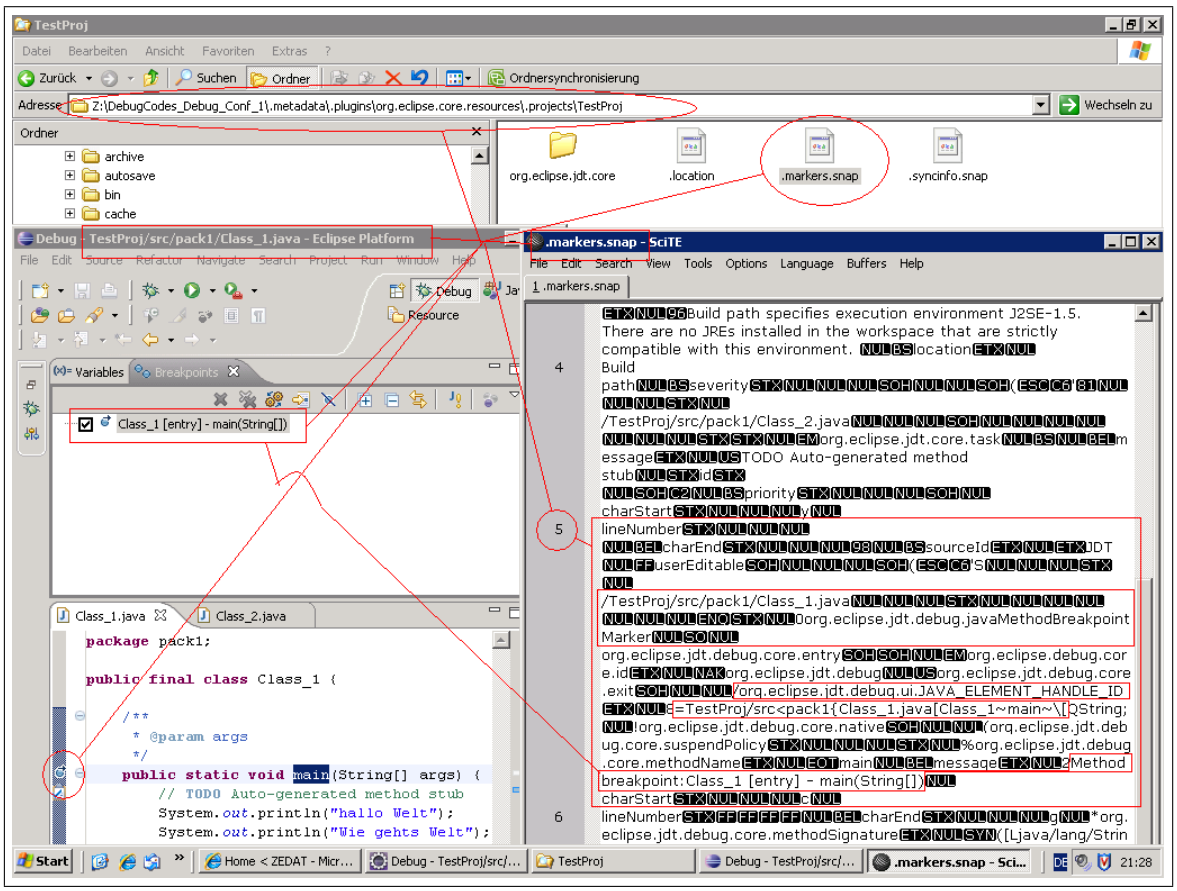

Abbildung 3.3: Persistente Speicherung von Breakpoitns in Eclipse

ist es nun möglich die Extension des Breakpoint-Typs zu bestimmen und darüber den parameterlosen Konstruktor des Breakpoints aufzurufen. Das Problem bei dieser Sache ist nun folgendes. Die persistenten Daten sind nicht direkt über die IMarker Instanz auslesbar. Der Grund liegt darin, dass IMarker nur ein Handle-Objekt ist, das unter anderem im `MarkerManager` dazu benutzt wird das zugehörige `MarkerInfo`-Objekt zu finden. Das MarkerInfo-Objekt verwaltet nun die persistenten Attribute eines Breakpoints in einer Map, welche Tupel der Form „<Name,Wert>“ enthält. Im Ergebnis dieser Vorüberlegungen ist nun folgende Verfahrensweise zur Serialisierung der Breakpoints enstanden.

```
1  public BreakpointRequestActivityDataObject(JID creatorOfbkpt,
2      IBreakpoint sendersLocalBreakpoint, ISarosSession
         sarosSession,
3      long bkpNmbrOfPeerSource, int[] activityLeads_ToRevFromRev)
         {
4      ...
5      Workspace workspace = (Workspace) ResourcesPlugin.
         getWorkspace();
6      IMarker marker = sendersLocalBreakpoint.getMarker();
7      SPath sPath = new SPath(marker.getResource());
8      fileOfBreakpoint = sPath.toSPathDataObject(sarosSession);
```

```
9       MarkerManager markerManager = workspace.getMarkerManager();
10      breakpointValueObject = markerManager.findMarkerInfo(marker
11          .getResource(), marker.getId());
12
13  }
```

Quellcode 3.6: Breakpoints werden serialisierbar gemacht

Das `breakpointValueObjekt` ist ein Feldelement vom Typ `MarkerInfo` und wird nicht als `transient` gesetzt. Somit wird es bei der Serialisierung der Klasse BreakpointRequestActivityDataObject [4] berücksicht. Die Klasse `BreakpointRequestActivityDataObject` delegiert dazu unseren `BreakpointActivityDataObjectConverter` über die folgende Java-Serialisierungs-Hook-Methode

```
private void writeObject(ObjectOutputStream out) throws
    IOException {
        out.defaultWriteObject();
    }
```

sich für seine Aufgabe an seine Basisklasse zu wenden. Die wiederum führt ein Lookup nach Convertern für alle Feldelemente des zu serialisierenden Objektes durch, welche nicht als transient markiert sind. Da es keinen registrierten Converter für `MarkerInfo` gibt, wird der Default-Converter `ReflectionConverter` mit der Serialiserung von MarkerInfo beauftragt. Diese etwas komplizierte und abenteuerlich anmutende Vorgehensweise ist nötig, damit wir bei der eigentlich entscheidenderen Deserialiserung Kontrolle über den Deserialisierungsvorgang bekommen. Aufgrund der Tatsache, dass der SerializableConverter das normale Verhalten des Java Serialsierungsframeworks emuliert, können wir über die Hook-Methode `readObject(ObjectInputStream)` das Breakpoint-Objekt wieder aus dem MarkerInfo-Objekt generieren. Dieser Prozess entspricht weitgehend dem Vorgang aus `BreakpointManager.createBreakpoint(IMarker)`. Allerdings sind einige Besonderheiten zu berücksichtigen bzw. zu korrigieren, die mit den verwendeten Datentypen in MarkerInfo zusammenhängen. Auf diese Details sei hier aber nur verwiesen.

3.7 Starten des Multi-User Debuggers

Wie wir in 3.1 festgelegt haben, wird das Starten des Debuggers ausschließlich dem Host erlaubt. Um nun eine Idee dafür zu bekommen, welche Strategie angebracht ist um diesen Start bei anderen Teilnehmern replizieren zu lassen, schauen wir uns einige Startvarianten von Eclipse beim Debuggen an. Nachfolgende Aufzählung soll als „textuelles Sequenzdiagramm“ aufgefasst werden und ist daher nicht erschöpfend kommentiert. Weiter gehts in Unterabschnitt 3.7.3.

3.7.1 Java Application Launch

Actionhandler startet Vorgang

1. LaunchAction.run() [5]

[4] auch hier: eigentlich muß es natürlich „Serialisierung des Objektes der Klasse“ heißen
[5] org.eclipse.debug.ui.actions

2. DebugUITools.launch(fCfg, fMode) [6]
3. DebugUIPlugin.launchInBackground(fCfg, fmode) [7]
4. DebugUIPlugin.buildAndLaunch(fCfg, fmode, new_monitor) [7]
5. fCfg.launch(fmode, new_monitor2, bBL [8], true) [9]
   a) Set modes = getModes();
      modes.add(fmode);
      ILaunchDelegate[] delegates = getType().getDelegates(modes);
      ILaunchConfigurationDelegate delegate = delegates[0].getDelegate(); [10]
   b) launch = new Launch(this, fmode, null) [11]
   c) launch.setAttribute(DebugPlugin.ATTR_CAPTURE_OUTPUT, null)
   d) launch.setAttribute(DebugPlugin.ATTR_CONSOLE_ENCODING, getLaunchManager().getEncoding(this))
   e) getLaunchManager().addLaunch(launch)
   f) initializeSourceLocator(launch)
   g) delegate.launch(this, fmode, launch, new_monitor3)

Extension 2.6.2 auf org.eclipse.debug.core.launchDelegates basiert gehts weiter

Folgendes passiert nun in delegate.launch(fCfg, fmode, launch, new_monitor3):

1. IVMRunner runner = getVMRunner(fCfg, mode) [12]
2. VMRunnerConfiguration runConfig = new VMRunnerConfiguration(mainTypeName, classpath)
3. runner.run(runConfig, launch, new_monitor3)

Weiter gehts mit `StandardVMDebugger.run(runConfig, launch, new_monitor3)`:

1. int port= SocketUtil.findFreePort()
2. String program= constructProgramString(runConfig);
   List arguments= new ArrayList(12);
   arguments.add(program);
3. ListeningConnector connector= getConnector()
4. connector.startListening(map)
5. Process p = exec(cmdLine, workingDir, envp)
6. IProcess process= newProcess(launch, p, renderProcessLabel(cmdLine), getDefaultProcessMap())

[6] org.eclipse.debug.ui.DebugUITools.launch(ILaunchConfiguration, String)
[7] org.eclipse.debug.internal.ui
[8] buildBeforeLaunch boolean Argument
[9] fCfg.getClass()==org.eclipse.debug.internal.core.LaunchConfiguration.class
[10] delegate = org.eclipse.jdt.launching.JavaLaunchDelegate
[11] org.eclipse.debug.core
[12] runner ist StandardVMDebugger

7. ConnectRunnable runnable = new ConnectRunnable(connector, map)
8. VirtualMachine vm= runnable.getVirtualMachine()
9. createDebugTarget(runConfig, launch, port, process, vm)
   - JDIDebugModel.newDebugTarget(launch, vm, renderDebugTarget(runConfig.getClassToLaunch(), port), process, true, false, runConfig.isResumeOnStartup())

3.7.2 Remote Java Application Launch

Die Schritte 1 bis 5 der ersten Aufzählung aus Java Application Launch gelten auch hier. Der ermittelte `delegate` ist hier aber vom Typ `JavaRemoteApplicationLaunchConfigurationDelegate` [13]. Folgendes passiert nun in delegate.launch(fCfg, fmode, launch, new_monitor3):

1. String connectorId [14] = getVMConnectorId(fCfg)
2. IVMConnector connector = JavaRuntime.getVMConnector(connectorId)
3. Map argMap = configuration.getAttribute(IJavaLaunchConfigurationConstants.ATTR_CONNECT_MAP, (Map)null)
4. connector.connect(argMap, new_monitor3, launch)

Für den IVMConnector gibt es zwei implementierende Klassen: SocketListenConnector und SocketAttachConnector, beide definiert im Plug-In `org.eclipse.jdt.launching`. Die Schnittstelle `IVMConnector` ist das Analogon von der `Connector`-Schnittstelle [15] aus der JDI Spezifikation. Dementsprechend ist `SocketListenConnector` analog zu `ListeningConnector` und `SocketAttachConnector` analog zu `SocketAttachConnector` aus der JDI Spezifikation. Diese Tatsache erkennt man auch daran, dass die beiden Klassen die eigentliche Verbindungsherstellung an Objekte deligieren, welche die entsprechenden JDI Schnittstellen implementieren. Zusammengefasst kann man sagen, dass `SocketListenConnector` nach Punkt 1 und `SocketAttachConnector` nach Punkt 2 aus 2.4 verfährt.

3.7.3 Idee der Umsetzung

Diese mitunter aufs wesentliche gekürzte Abfolge der Schritte endet in der statischen Methode `JDIDebugModel.newDebugTarget()`, welche die entscheidende Aufgabe hat, die Debuggingabstraktion von Eclipse mit derjenigen von JDI zu verbinden. Hier wird also das „Wurzelelement“ des Eclipse Java Debug Models `JDIDebugTarget` instanziiert und dabei so initialisiert, dass es sich aus Events und Requests der mit ihr assoziierten JDI Abstraktionsebene „speist“. Das bedeutet, dass weitere von JDIDebugElement abstammende Objekte in Folge der Verarbeitung von JDI-Aufrufen instanziiert oder manipuliert werden. Wie wir hier auch an der Reihenfolge sehen, wurde das VirtualMachine Objekt vor dem Aufruf von `newDebugTarget` erzeugt. Aufgrund des nun ermittelten Startablaufs wird auch die Einstiegsstelle deutlich, an der wir ansetzen müssen, um Multi-User Betrieb zu realisieren.

Die Lösung wie man bei einem Start des Debuggers durch den Host die Peers dazu bringen kann ebenfalls ihren Debugger zu starten ist, dass wir deren Startvorgang nach dem Muster von Re-

[13] org.eclipse.jdt.internal.launching
[14] Id Wert von Extension auf org.eclipse.jdt.launching.vmConnectors
[15] com.sun.jdi.connect

mote Java Application Launch implementieren. Dieser Launch ist deshalb so gut geeignet, weil er beim Aufruf von `JDIDebugModel.newDebugTarget` ohne IProcess Argument auskommt, falls der Verbindungsconnector der SocketAttachConnector ist [16]. Der Host sendet also nach seinem launch(...) eine Nachricht `MessageTypes.HOST_LAUNCHES` auf die ein Peer damit reagiert, eine Launchdatei zu generieren, welche aus einem früheren lokalen Debuggerstart existiert, indem diese kopiert wird und als Endung „_proxy“ bekommt. Diese Launchdatei benutzt er schließlich, um den Startprozess mit Aufzählungspunkt 2 der ersten Aufzählung zu initiieren.

Die Idee bei diesem Vorhaben ist es, die VirtualMachine – als „Wurzelement“ der JDI Spezifikation – vor der Übergabe an die newDebugTargetMethode in ein Wrapperobjekt zu packen und dann dieses Wrapperobjekt anstelle des „Originalobjektes“ zu übergeben. Das Wrapperobjekt muß hierbei natürlich das VirtualMachine Interface implementieren und dient dabei als delegierender Proxy. Somit ist es über den Proxy möglich die Kommunikation zwischen der ESDM Implementierung für Java [17] und der JDI-Implementierung zu überwachen und dazu zu nutzen verteiltes Debugging zu realisieren.

[16] Zeile in connect(...)-Methode: IDebugTarget debugTarget= JDIDebugModel.newDebugTarget(launch, vm, vmLabel, null, allowTerminate, true);

[17] in Form des JDI DebugModels

3.8 Umsetzung

1. Wir definieren eine neuen Debugmode „Saros.debug“ mittels Extension für `org.eclipse.debug.core.launchModes`.
2. Assoziieren über diesen neuen Debugmode den launchConfigurationType org.eclipse.jdt.launching.localJavaApplication auf DistributedJavaDebugLaunchDelegate statt JavaLaunchDelegate und erben dabei von diesem.
3. Erstellen eine Klasse, die von StandardVMDebugger erbt: ProxyStandardVMDebugger
4. Überschreiben getVMRunner() 1 in DistributedJavaDebugLaunchDelegate so, daß ein neues ProxyStandardVMDebugger Objekt zurückgegeben wird.
5. In ProxyStandardVMDebugger wird schließlich createDebugTarget wie folgt definiert:

```
@Override
protected IDebugTarget createDebugTarget(
    VMRunnerConfiguration config,
     ILaunch launch, int port, IProcess process,
        VirtualMachine vm) {
     LoggingVM loggingVM = new LoggingVM(vm);
     VirtualMachine proxy = (VirtualMachine) Proxy.
        newProxyInstance(
         VirtualMachine.class.getClassLoader(), new Class[]
            {
             VirtualMachine.class, org.eclipse.jdi.
                VirtualMachine.class,
             org.eclipse.jdi.hcr.VirtualMachine.class },
                loggingVM);
     SharedBreakpointManager.setSharedVM(proxy);
     return super
         .createDebugTarget(config, launch, port, process,
            proxy);
}
```

Quellcode 3.7: Proxy

6. Erstellen eine Extension auf org.eclipse.debug.ui.launchConfigurationTabs mit der wir die Tabgroup für den Java Launch, org.eclipse.jdt.debug.ui.launchConfigurationTabGroup.localJavaApplication, um ein weiteres Tab erweitern. In diesem neuen Tab definieren wir die Auswahlmöglichkeit für unseren neuen Debugmodus „Saros.debug“.

Über diese Maßnahmen können wir schließlich erreichen, dass der Host über den Eclipse Launcheditor einen neuen Launch erzeugt, den er über den zusätzlichen Reiter als verteilten Debuglaunch klassifiziert.

Dadurch das der Launch nun zwei Modes gleichzeitig aufweist stellt dieser einen sogenannten mixed-mode Launch dar. Über die oben durchgeführte Zuweisung unseres DistributedJavaDebugLaunchDelegate an diesen mixed-launch Mode wählt nun das Eclipse Launch Framework unsere Klasse

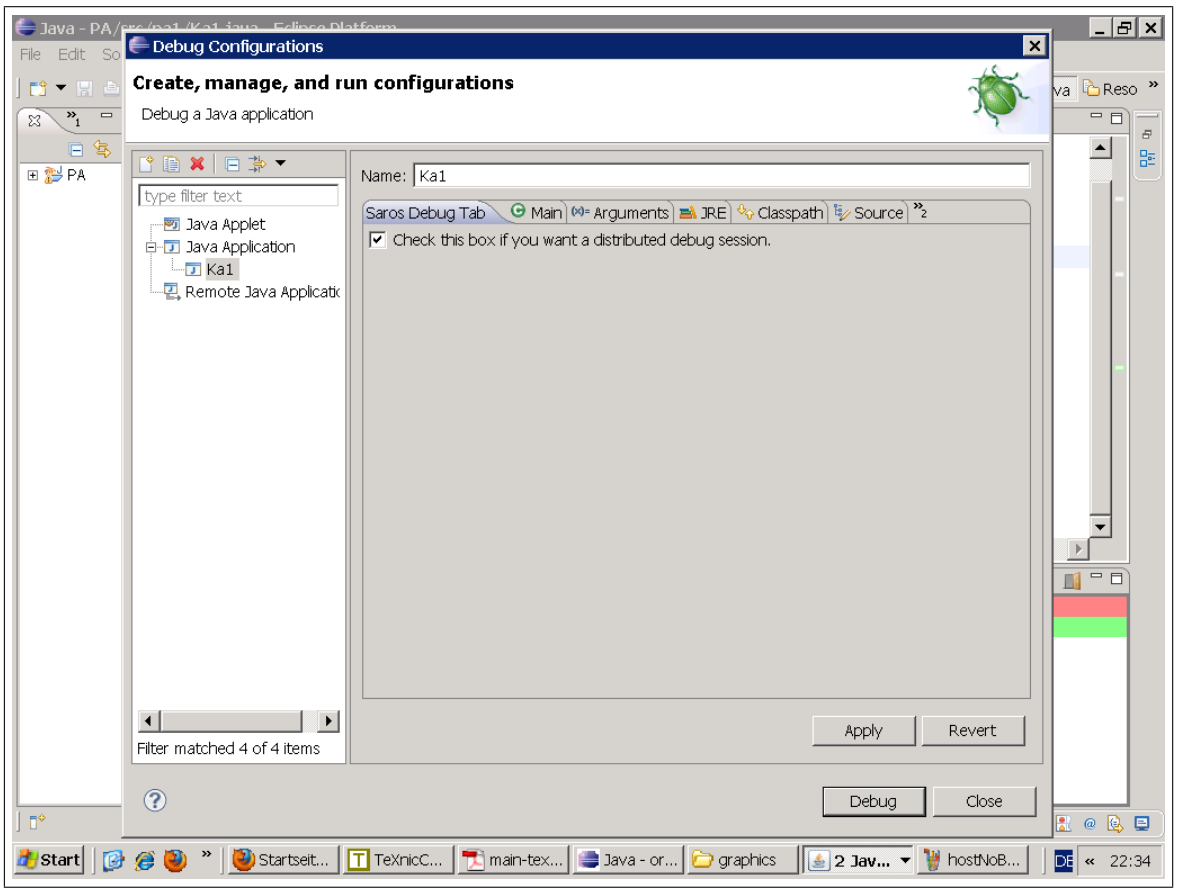

Abbildung 3.4: Host startet verteiltes Debugging

aus, so daß unsere ProxyVM [18] Einzug in den Startvorgang findet. Die nachfolgenden Schritte orientieren sich an dem Ziel, dass das JDI- Backend des Host, welches durch die VirtualMachine Instanz „verwurzelt“ ist, geeignet protokolliert und analysiert werden muß, damit daraus ein mehrere-JDI-front-ends-unterstützendes Backend wird. Dieses Ziel kann erreicht werden, indem JDI-Aufrufe in die Kategorien „nur einmal ausführbar“ und „mehrfach ausführbar“ eingeteilt werden und bei Ersterem wiederholte Anfragen seitens Proxy-Backends vom Cache bedient werden. Die zweite Kategorie, zu welcher zum Beispiel Variablenabfragen gehören, können direkt stellvertretend vom Host JDI-Backend ausgeführt werden und müssen anschließend dem Anfrager geeignet zugesandt werden. In diesem Zusammenhang ergibt sich natürlich das generelle Problem der adäquaten Serialisierung.

3.9 Remote Procedure Calls im Kontext von Objektserialisierung

Remote Procedure Calls sind Methodenaufrufe, die nicht lokal auf dem Rechner des Aufrufers ausgeführt werden, sondern auf einem entfernten Rechner, aber deren Ergebnis dem lokalen Aufrufer zur Verfügung gestellt werden soll [15, S. 93-103]. Ein solcher Aufruf soll für den Aufrufer transparent erfolgen, also den gleichen Effekt haben wie ein analoger lokaler Aufruf. Die Implentierung

[18] Dynamische Proxys in Java

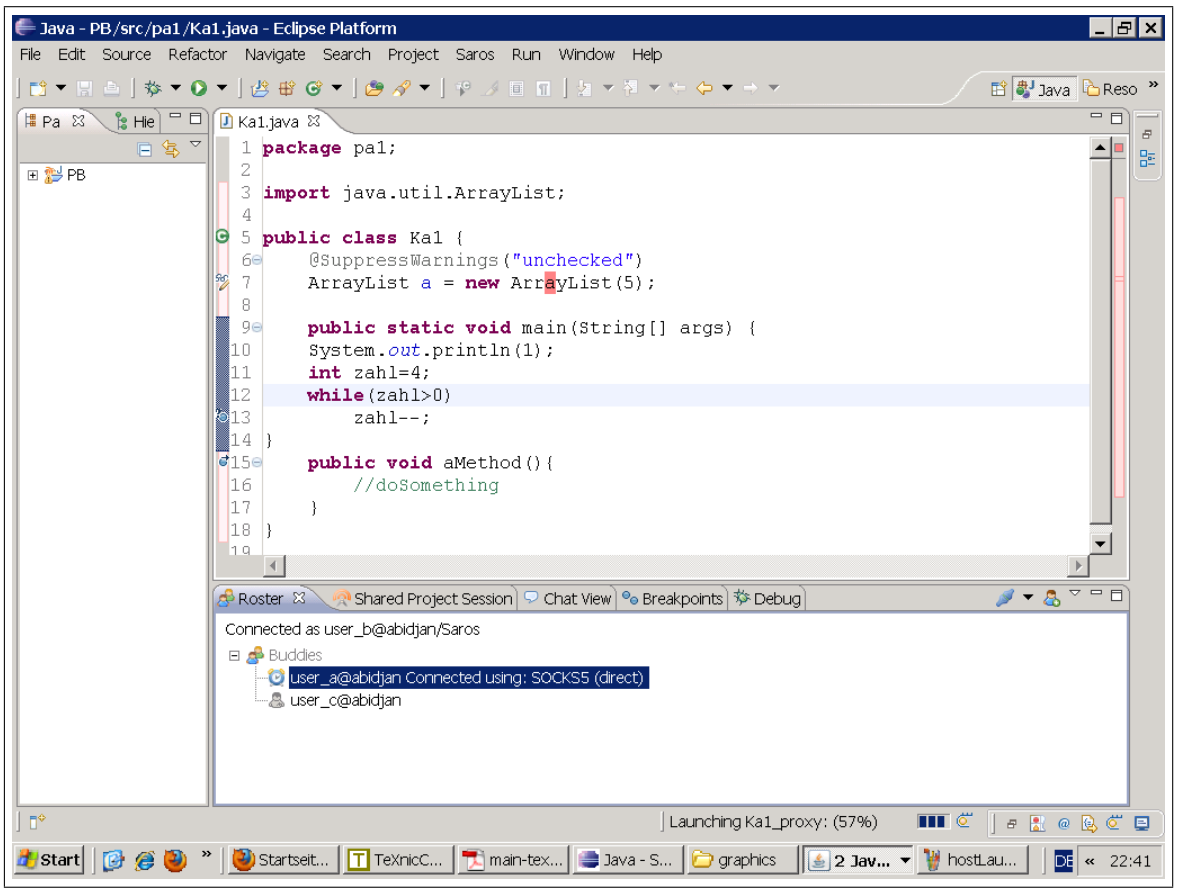

Abbildung 3.5: Zu sehen ist der Progress Balken mit Beschriftung Launching Ka1_proxy

von solchen RPC genannten Aufrufen ist für skalare oder immutable (unveränderliche) Argumente und Rückgabewerte vergleichsweise einfach zu realiseren. Dazu müssen sämtliche zu kommunizierenden Daten serialisert und über einen Nachrichtenkanal übertragen werden. Dabei kommt es sowohl auf lokaler als auch auf Remoteseite zu sogenannten Stellvertreterobjekten, den sogenannten Client-Stubs [19] und Server-Stubs [20]. Der Client-Stub erscheint für den lokalen Aufrufer wie ein ganz normales Objekt bzw. Modul und verarbeitet einen Funktionsaufruf nicht wie gewöhnlich mittels Delegation an eine Serie von Bibliotheksfunktionen, sondern sendet eine Nachricht mit dem Funktionsnamen und den geeignet codierten serialiserten Funktionsargumenten an den zugeordneten Remote-Server-Stub. Anschließend blockiert der Client-Stub solange bis er das Ergebnis vom Server-Stub erhält. Der Server-Stub erhält die Nachricht und decodiert die Argumente entsprechend und ruft dann seinerseits die auf dem Server vorhandene native Funktion mit diesen nun serverlokalen Daten auf. Für das Serverbetriebssystem ist dieser Aufruf ebenso transparent wie auf lokaler Seite der Aufruf des Client-Stubs. Nachdem die Funktion ihr Ergebnis geliefert hat verpackt Sie der Server Stub als Nachricht und schickt diese an den wartenden Client-Stub zurück. Nun entsteht aber ein Problem, wenn es sich bei dem Rückgabewert um ein Objekt handelt, das seinerseits Referenzen auf andere Objekte oder Datenstrukturen enthält. In diesem Fall beziehen diese sich auf den Adressraum im Serverprozess. Bei einfachen Feldern und Datenstrukturen, welche

[19] auch: Client-Proxy
[20] auch: Server-Proxy

keine Objekte mit aufrufbaren Funktionen enthalten, kann man dazu übergehen diese Strukturen rekursiv zu dereferenzieren und als Bytesequenz mitzuschicken. Im anderen Fall kann man diese Strategie allerdings nicht fahren, da man sich nicht sicher sein kann, ob und welche Funktionen auf diesen Unterobjekten auf der Client-Seite aufgerufen werden. Ein möglicher Ansatz ist die Ersetzung jeder Objektreferenz im Rückgabewert durch entsprechende Stub-Objekte, welche dann serialisiert werden um beim Client wieder transparent Funktionen an parallel erzeugte Server-Stubs im Serverprozess weiterleiten. Dieser Strategie bedient sich auch der RMI-Ansatz von Java. In RMI wird jedes Objekt, welches als Rückgabewert einer Funktion, die mittels eines Stubs aufgerufen wurde, durch einen entsprechenden Stub ersetzt, aber nur wenn das reale Objekt im RMI eigenen Namensdient exportiert wurde. Dies geschieht mitunter automatisch, wenn die Klasse vom Typ java.rmi.server.UnicastRemoteObject ist.

Was wir nun bei der Objektserialisierung im Zusammenspiel von Host JDI-Backend (Server-Proxy) und Peer JDI-Backends (Client-Proxies) machen müssen ist der Nachbau von RMI unter Nutzung des vorhandenen XMPP Protokolls. Allerdings ist das angestrebte Verhalten keine 1:1 Umsetzung von RMI da die JDI-Schnittstelle nur für Single-User Betrieb ausgelegt ist und man daher für jede Methode in JDI prüfen muss, ob Sie zur Kategorie „einmal ausführbar“ oder „beliebig ausführbar“ zählt. Dementsprechend kann man die Umsetzung als „Caching RMI“ deklarieren.

```
1  public class LoggingEventRequestManager implements
       InvocationHandler {
2
3      protected EventRequestManager delegate;
4      protected ThreadStartRequest fThreadStartRequest;
5
6      public LoggingEventRequestManager(EventRequestManager
           delegate) {
7          this.delegate = delegate;
8      }
9
10     public Object invoke(Object delegater, Method method,
           Object[] args)
11         throws Throwable {
12         if (method.getName().equals("createThreadStartRequest")
               ) {
13             if (fThreadStartRequest == null) {
14                 fThreadStartRequest = (ThreadStartRequest)
                       method.invoke(
15                         delegate, args);
16             }
17             return fThreadStartRequest;
18         }
19         return method.invoke(delegate, args);
20     }
21
22 }
```

Quellcode 3.8: Server-Proxy: Beispiel für „einmal ausführbar“

```
1  public class ProxyThreadStartRequest extends WakeupAble
      implements
2      InvocationHandler {
3
4      // This is also a Proxy
5      protected final EventRequestManager parent;
6
7      public ProxyThreadStartRequest(EventRequestManager parent)
          {
8          this.parent = parent;
9      }
10
11     public Object invoke(Object delegater, Method method,
          Object[] args)
12         throws Throwable {
13         final String methodName = method.getName();
14         if (methodName.equals("setStorage")) {
15             synchronized (this) {
16                 method.invoke(this, args);
17                 notify();
18             }
19         } else if (methodName.equals("virtualMachine")) {
20             return parent.virtualMachine();
21         } else
22         //because the host has already made the setup for us,
              we must not repeat this calls! Just ignore.
23         if (methodName.equals("addThreadFilter")
24             || methodName.equals("enable")
25             || methodName.equals("setSuspendPolicy")) {
26             // instead of void we use null
27             return null;
28         }
29         return null;
30     }
31 }
```

Quellcode 3.9: Client-Proxy: Beispiel für „einmal ausführbar“

3.10 Zusammenfassung

In dieser Diplomarbeit ging es darum, am Beispiel eines konrekten Debuggers die Möglichkeiten zur Erweiterung in einen verteilten Multi-User Debugger zu erforschen. Am Beispiel des Java De-

buggers habe ich zunächst untersucht, welche Möglichkeiten zur Implementierung angeboten werden. Dabei habe ich die Java Platform Debug Architecture (JPDA) eingehend mit Ihren aufeinander aufbauenden Schichten erklärt und in ihrem Zusammenwirken beschrieben. Besonderen Wert habe ich dabei auf die darin verwendete Terminologie gelegt, um diese in einem nachfolgenden Schritt im Eclipse Kontext anzuwenden. Anschließend habe ich das Eclipse Standard Debug Model (ESDM) erklärt und gezeigt wie sich der Java Debugger darin integriert hat. Hierbei habe ich die Terminologie aus der JPDA konsequent fortgeführt und die Interaktion zwischen der JDI-Abstraktion einserseits und der ESDM-Abstraktion andererseits am konkreten Beispiel der Installation eines Breakpoints in einer Java Virtual Machine (JVM) gezeigt. In einer anschließenden Analysephase habe ich die Möglichkeiten des Eingriffs in das Eclipse Debugging Framework hinsichtlich Vor- und Nachteilen verglichen und die JDI Abstraktion als Kanditaten für eine Implementierung ausgewählt. Im darauffolgenden Implementierungskapitel habe ich einen Anforderungskatalog für einen multi-user Debugger aufgestellt und dabei die beiden Phasen Pre-Launch und Launch definiert. Die Aufgaben und Probleme die sich beim verteilten Debugging ergeben habe ich dann in diese beiden Phasen aufgeteilt und gezeigt welche Lösungsstrategien es gibt. Dabei hat sich gezeigt, dass sogar die Pre-Launch Phase nicht trivial in ihren Problemfeldern ist. Ich habe dazu die möglichen Konfliktpaare identifiziert und ein versionsbasiertes Konsistenzschema definiert, das Konflikte erkennen und vermeiden soll. Die Serialisierung von Breakpoints ist mir auf eine generische Weise gelungen, so dass der Austausch von Breakpoints verschiedener Programmiersprachen in Eclipse unterstützt wird. Dies ist von Vorteil, wenn man in einem nächsten Schritt einen Multi-User Debugger für andere Programmiersprachen entwickeln möchte. Die Launch-Phase erfordert ungleich mehr Zeit- und Analyseaufwand und es konnte gezeigt werden, daß es mittels Javas dynamischer Proxy Architektur möglich ist, dieses Ziel zu erreichen. Dabei kam der Analyse des Startvorgangs ein großes Augenmerk zuteil und es gelang mir in diesen einzugreifen und den Startvorgang des Debuggers beim Peer zu replizieren.

3.11 Fazit und Ausblick

Ich habe in dieser Diplomarbeit einen großteil meiner Zeit in die Analyse der Vorgänge beim Debuggen gesteckt. Dabei ist es mir gelungen die Pre-Launch Phase nahezu komplett abzuschließen. In der Diplomarbeit wurde zur Konsistenzsicherstellung das Verfahren von gängigen Versionskontrollsystemen angewandt. Den in der Implementierung meines Kontrollsystems benutzten Mechanismus zur atomaren Verarbeitung von lokalen und externen Events habe ich in dieser Diplomarbeit nicht explizit erklärt. Um die Atomarität der Verarbeitung zu gewährleisten hatte ich das in Eclipse vorhandene Framework von Scheduling Rules benutzt, welches dafür sorgt, dass auf der zugehörigen Datei des Breakpoints keine nebenläufige Verarbeitung entsteht, welche nicht durch die verwendete Regel erlaubt ist.

In der Implementierung der Breakpointverteilung bot es sich eigentlich an vom Jupiter-Algorithmus Gebrauch zu machen, da es zum Beispiel durchaus die Situation geben kann, in der ein Benutzer auf einer Zeile einen Breakpoint setzen will und ein anderer Benutzer die Zeile textuell so verändert, dass es sich um einen semantisch anderen Programmcode handelt. Leider ist der Jupiter-Algorithmus nur mit viel Spezifikations- und Arbeitsaufwand für weitere Anwendungsfälle anwendbar und daher begnügen wir uns zunächst mit der einfacheren Taktik der lokalen Ausführung und eventueller Rückgängigmachung von Aktionen. Die Pre-Launch Phase kann ich als im wesentlichen abgeschlossen betrachten, jedoch muß man noch die MODIFY Events auf Breakpoints von

zur Zeit „ToggleEnable“ auf die weiteren Arten, welche aus allen Arten von Attributänderungen des Breakpoints resultieren, implementieren. Für diese Diplomarbeit habe ich dieser Aufgabe allerdings eine niederwertige Priorität geben müssen, damit ich ein Framework für die Launch-Phase implementieren konnte. Dieses Framework funktioniert meines Erachtens und es Bedarf lediglich eines großen Arbeitsaufwandes um es zu vervollständigen. Die Launch-Phase kann man mit Javas dynamischem Proxy-Framework geeignet realisieren. In diesem Zusammenhang schlage ich vor einen generischen Proxy zu schreiben, welcher Java Reflection einsetzt. Damit wäre es möglich einen regelbasierten generischen Proxy zu entwickeln, welcher die Ein- und Ausgabeparameter der JDI-Methoden analysiert und entsprechend regelbasiert implementiert.

KAPITEL 4

Anhang

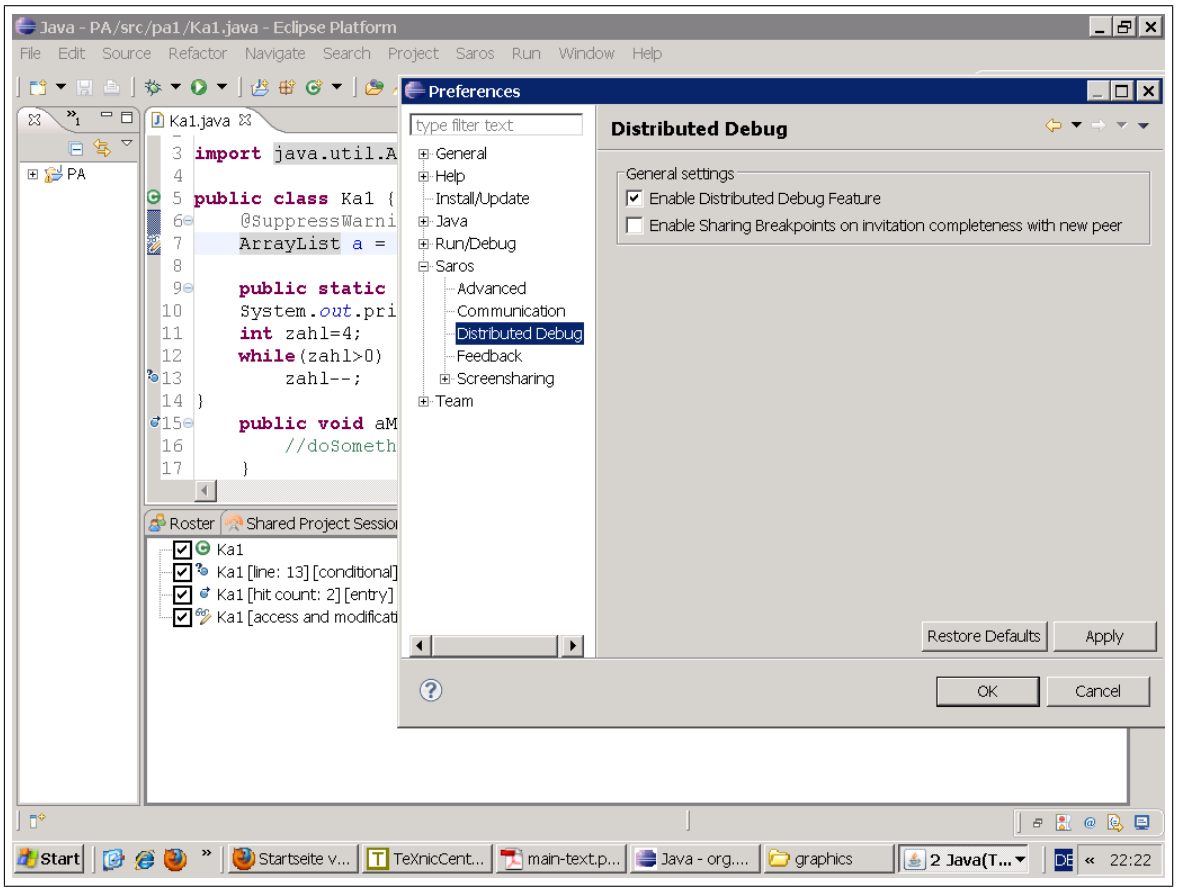

Abbildung 4.1: Zu sehen ist der Host mit seinen unterschiedlichen Typen von Breakpoints und der Konfigurationsdialog

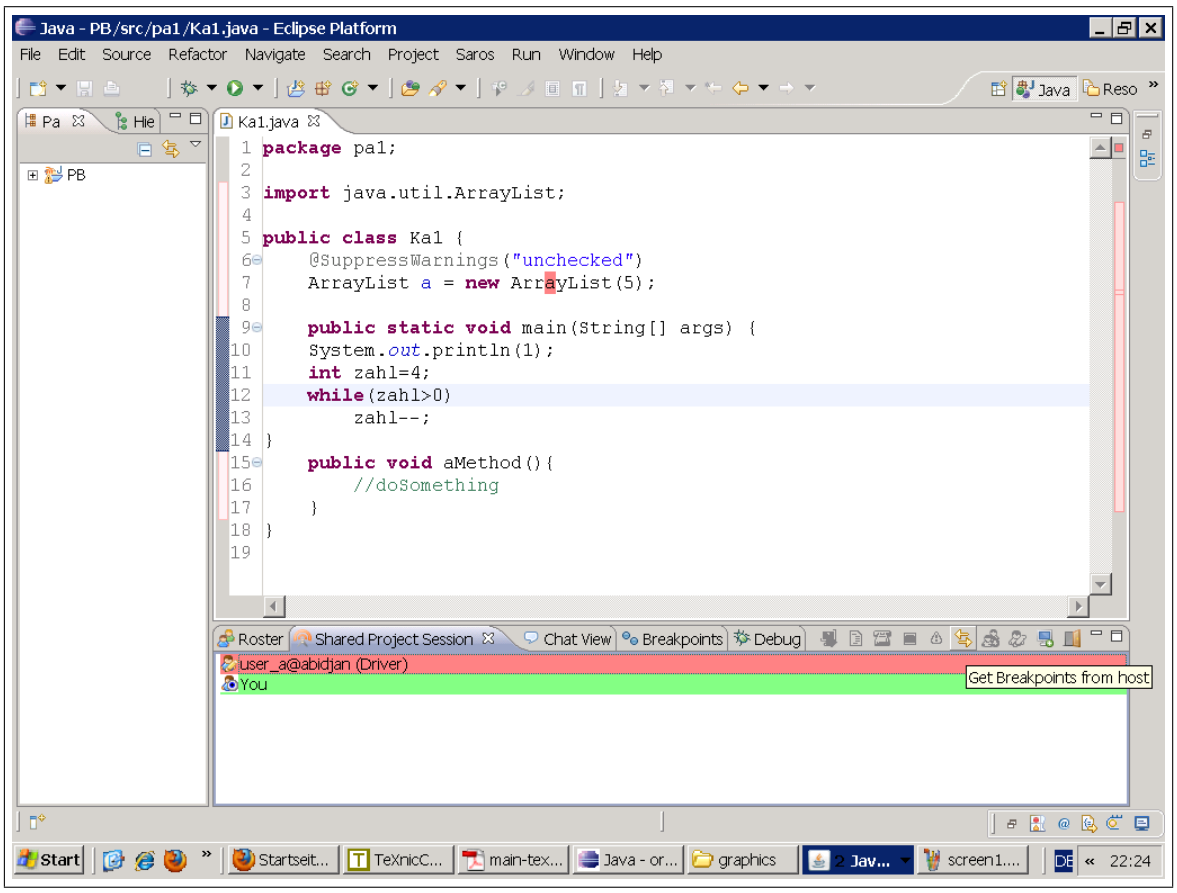

Abbildung 4.2: Szenario, indem Peer keine eigenen Breakpoints besaß und nach Einladungsvorgang noch keine Breakpoints angefordert hat

```
/**
 * Create a breakpoint for the given marker. The created
   breakpoint
 * is of the type specified in the breakpoint extension
   associated
 * with the given marker type.
 *
 * @param marker marker to create a breakpoint for
 * @return a breakpoint on this marker
 * @exception DebugException if breakpoint creation fails.
   Reasons for
 *  failure include:
 * <ol>
 * <li>The breakpoint manager cannot determine what kind of
   breakpoint
 *     to instantiate for the given marker type</li>
 * <li>A lower level exception occurred while accessing the
   given marker</li>
```

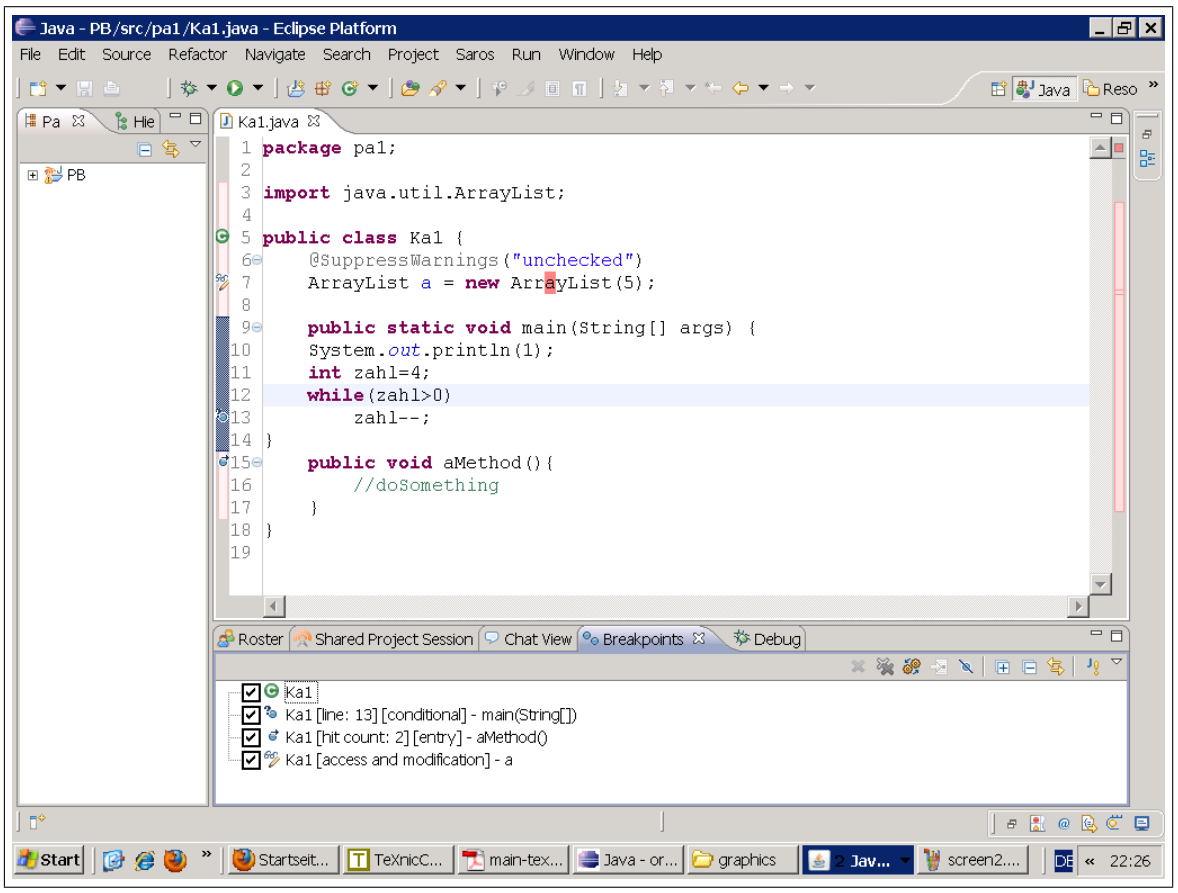

Abbildung 4.3: Wie hier zu sehen ist, sind alle Typen von Breakpoints des Host mit allen Attributen übertragen worden

```
 * </ol>
 */
public IBreakpoint createBreakpoint(IMarker marker) throws
    DebugException {
        IBreakpoint breakpoint= (IBreakpoint)
            fMarkersToBreakpoints.get(marker);
        if (breakpoint != null) {
                return breakpoint;
        }
        try {
                IConfigurationElement config = (
                    IConfigurationElement)fBreakpointExtensions.
                    get(marker.getType());
                if (config == null) {
                        throw new DebugException(new Status(
                            IStatus.ERROR, DebugPlugin.
                            getUniqueIdentifier(),
                                DebugException.
```

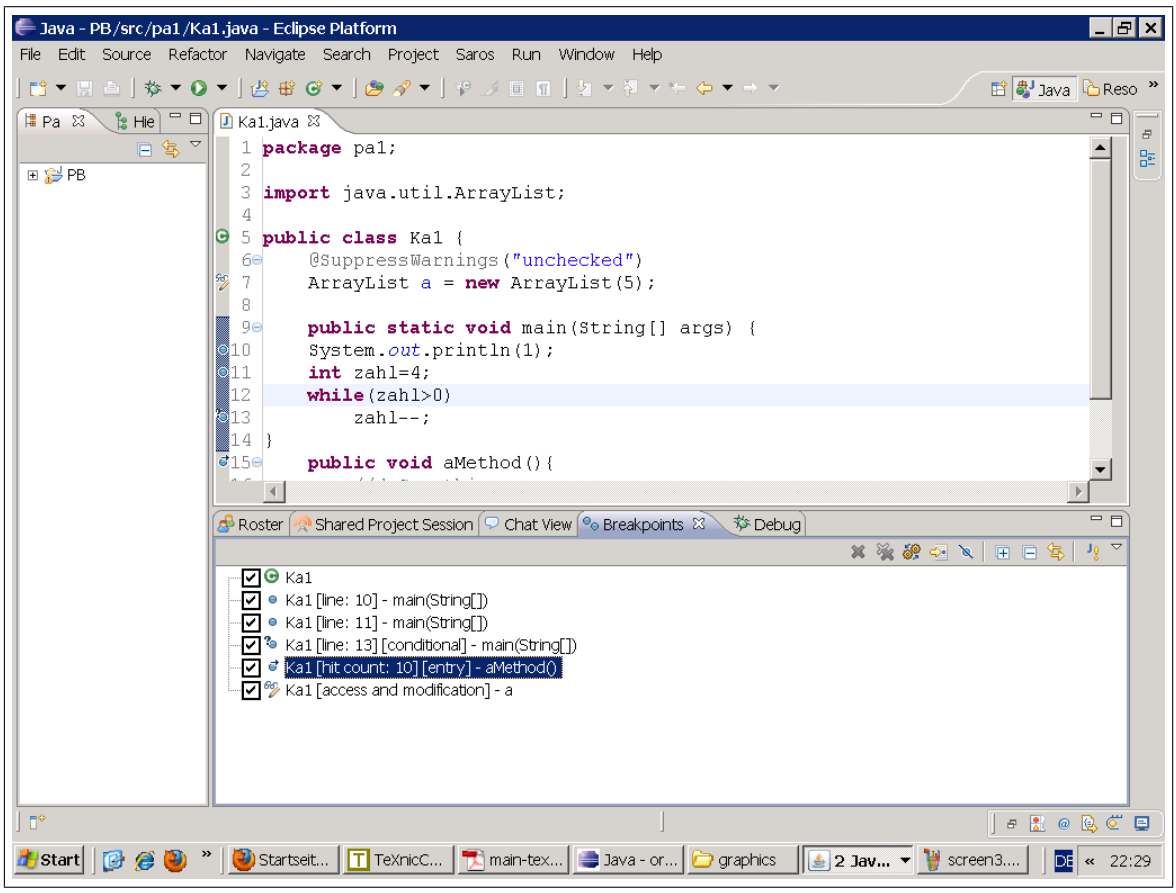

Abbildung 4.4: Szenario, indem Peer eigene Breakpoints besaß und nach Einladungsvorgang noch keine Breakpoints angefordert hat. Man beachte: hitcount hat den Wert 10 und auf Zeilen 10 und 11 sind Breakpoints

```
                                        CONFIGURATION_INVALID,
                                        MessageFormat.format(
                                        DebugCoreMessages.
                                        BreakpointManager_Missing_breakpoint_d
                                        , new String[] {marker.
                                        getType()}), null));
                }
                Object object = config.
                    createExecutableExtension(
                    IConfigurationElementConstants.CLASS);
                if (object instanceof IBreakpoint) {
                        breakpoint = (IBreakpoint)object;
                        breakpoint.setMarker(marker);
                } else {
                        DebugPlugin.log(new Status(IStatus.
                            ERROR, DebugPlugin.
                            getUniqueIdentifier(), DebugPlugin.
```

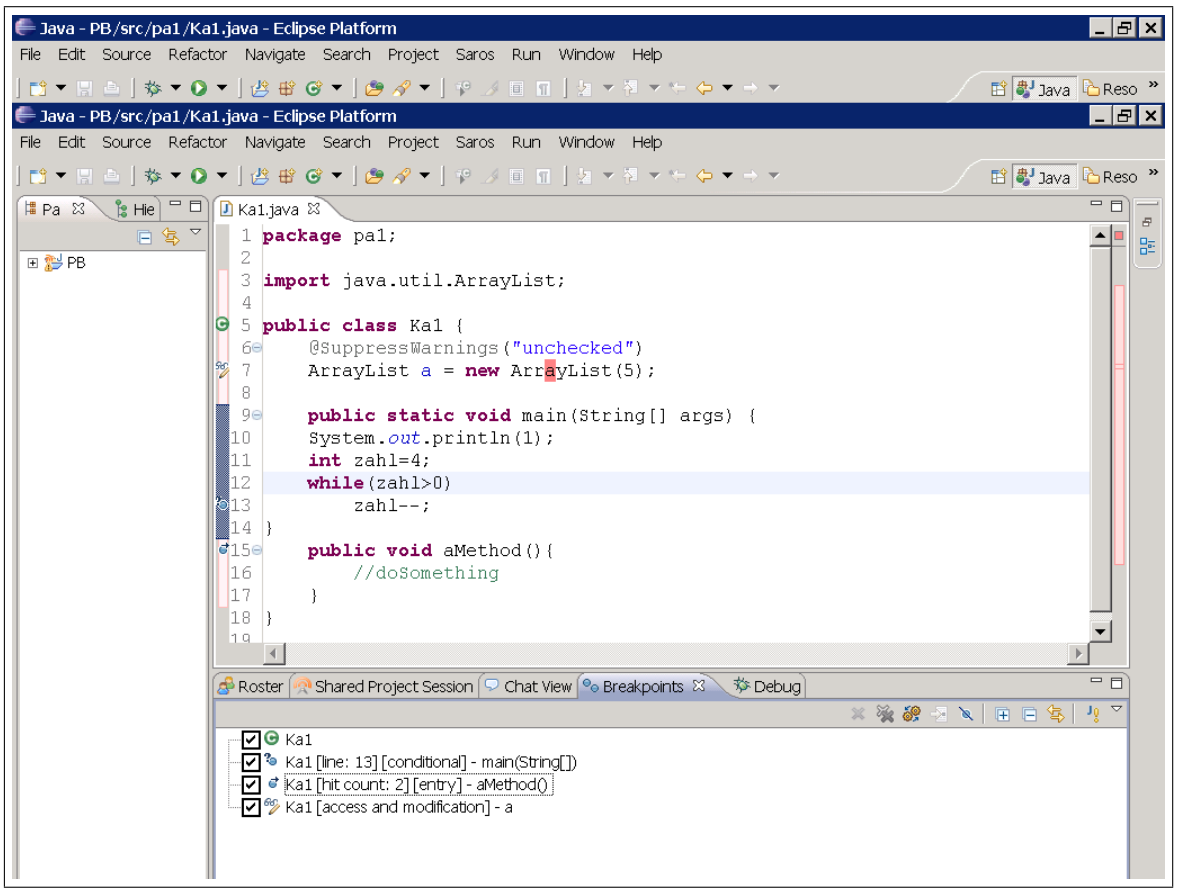

Abbildung 4.5: Folgezustand von Abbildung 4.4: Man beachte, daß Breakpoints auf Zeilen 10 und 11 gelöscht sind und hitcount den Wert 2 hat

```
                                INTERNAL_ERROR, "Breakpoint
                                extension " + config.
                                getDeclaringExtension().
                                getUniqueIdentifier() + " missing
                                required attribute: class", null));
                                //$NON-NLS-1$ //$NON-NLS-2$
                        }
                        return breakpoint;
                } catch (CoreException e) {
                        throw new DebugException(e.getStatus());
                }
        }
```

Quellcode 4.1: createBreakpoint(IMarker)

Tabelle 4.1: Vererbungslinie von IBreakpoint

Typ	**CompileTimeInheritance**	**RunTimeInheritance(getAdapter)**
IBreakpoint(I)	IAdaptable(I)	org.eclipse.ui.IPersistableElement
IJavaBreakpoint(I)	IBreakpoint	
ILineBreakpoint(I)	IBreakpoint	
IWatchpoint(I)	IBreakpoint	
Breakpoint(A)	PlatformObject(C) IBreakpoint	
JavaBreakpoint(A)	Breakpoint IDebugEventSetListener(I) IJavaBreakpoint IJDIEventListener(I)	
IJavaClassPrepareBreakpoint(I)	IJavaBreakpoint	
JavaClassPrepareBreakpoint(C)	JavaBreakpoint IJavaClassPrepareBreakpoint	
IJavaExceptionBreakpoint(I)	IJavaBreakpoint	
JavaExceptionBreakpoint(C)	JavaBreakpoint IJavaExceptionBreakpoint	
IJavaLineBreakpoint(I)	IJavaBreakpoint ILineBreakpoint	
JavaLineBreakpoint(C)	JavaBreakpoint IJavaLineBreakpoint	
IJavaMethodBreakpoint(I)	IJavaLineBreakpoint	
JavaMethodBreakpoint(C)	JavaLineBreakpoint IJavaMethodBreakpoint	
IJavaMethodEntryBreakpoint(I)	IJavaLineBreakpoint	
JavaMethodEntryBreakpoint(C)	JavaLineBreakpoint IJavaMethodEntryBreakpoint	
IJavaPatternBreakpoint(I)	IJavaLineBreakpoint	
JavaPatternBreakpoint(C)	JavaLineBreakpoint IJavaPatternBreakpoint	
IJavaStratumLineBreakpoint(I)	IJavaLineBreakpoint	
JavaStratumLineBreakpoint(C)	JavaLineBreakpoint IJavaStratumLineBreakpoint	
IJavaTargetPatternBreakpoint(I)	IJavaLineBreakpoint	
JavaTargetPatternBreakpoint(C)	JavaLineBreakpoint IJavaTargetPatternBreakpoint	
IJavaWatchpoint(I)	IJavaLineBreakpoint IWatchpoint	
JavaWatchpoint(C)	JavaLineBreakpoint IJavaWatchpoint	

Super-Marker-ID	**Marker-ID**	**Breakpoint-id**	**Plug-in ID**	**Kommentar**
org.eclipse.core-.resources.marker	breakpointMarker	-	org.eclipse.debug-.core	Break-point(A)
org.eclipse.debug-.core.breakpoint-Marker	lineBreakpoint-Marker	-	org.eclipse-.debug.core	LineBreak-point(A)
org.eclipse.core-.resources.textmarker				
org.eclipse.debug-.core.breakpoint-Marker	javaBreakpoint-Marker	-	org.eclipse-.jdt.debug	JavaBreak-point(A)
org.eclipse.jdt.debug-.javaBreakpoint-Marker	javaClassPrepare-BreakpointMarker	javaClass-Prepare-Breakpoint	org.eclipse-.jdt.debug	Klassenname = Break-point-id mit großem Anfangs-buchstaben
org.eclipse.jdt.debug-.javaBreakpoint-Marker	commonJavaLine-BreakpointMarker	-	org.eclipse-.jdt.debug	
org.eclipse.debug-.core.lineBreakpoint-Marker				
org.eclipse.jdt.debug-.commonJavaLine-BreakpointMarker	javaLineBreak-pointMarker	javaLine-Breakpoint	org.eclipse-.jdt.debug	
org.eclipse.jdt.debug-.commonJavaLine-BreakpointMarker	javaPatternBreak-pointMarker	javaPattern-Breakpoint	org.eclipse-.jdt.debug	
org.eclipse.jdt.debug-.commonJavaLine-BreakpointMarker	javaTargetPattern-BreakpointMarker	javaTarget-Pattern-Breakpoint	org.eclipse-.jdt.debug	
org.eclipse.jdt.debug-.javaBreakpoint-Marker	javaException-BreakpointMarker	java-Exception-Breakpoint	org.eclipse-.jdt.debug	
org.eclipse.jdt.debug-.javaLineBreakpoint-Marker	javaWatchpoint-Marker	java-Watchpoint	org.eclipse-.jdt.debug	
org.eclipse.jdt.debug-.javaLineBreakpoint-Marker	javaMethodBreak-pointMarker	javaMethod-Breakpoint	org.eclipse-.jdt.debug	
org.eclipse.jdt.debug-.javaLineBreakpoint-Marker	javaMethodEntry-BreakpointMarker	javaMethod-EntryBreak-point	org.eclipse-.jdt.debug	
org.eclipse.jdt.debug-.commonJavaLine-BreakpointMarker	javaStratumLine-BreakpointMarker	javaStratum-LineBreak-pointMarker	org.eclipse-.jdt.debug	

Tabelle 4.2: Assoziation von Markern zu Breakpoints. Angegeben ist das Plug-in, in welchem es definiert ist. (A) bedeutet, daß ein zugehöriger Breakpoint nicht existiert, weil dieser in Form einer Abstrakten Klasse nicht instantiiert werden kann.

Literaturverzeichnis

[1] Joshua Bloch. *Effective Java Second Edition*. Sun Microsystems, Inc., 2008.

[2] Prof. Dr. Volker Claus and Prof. Dr. Andreas Schwill. *DUDEN Informatik*. Number ISBN 3-411-05233-3. Dudenverlag, 3 edition, 2001.

[3] Prasun Dewan and Rajiv Choudhary. A high-level and flexible framework for implementing multi-user user-interfaces. *ACM Transactions on Information Systems*, 10:345–380, 1992.

[4] Riad Djemili. Entwicklung einer eclipse-erweiterung zur realisierung und protokollierung verteilter paarprogrammierung. Master's thesis, Freie Universität Berlin, 2006.

[5] Erich Gamma and Kent Beck. *Contributing to eclipse Principles, Patterns, and Plug-ins*. Addison-Wesley, 2004.

[6] Brian Goetz. *Java Concurrency in Practice*. Number ISBN-13: 978-0-321-34960-6. Addison-Wesley, 2006.

[7] Ralph D. Hill, Tom Brinck, Steven L. Rohall, John F. Patterson, and Wayne Wilner. The rendezvous architecture and language for constructing multiuser applications. *ACM Trans. Comput.-Hum. Interact.*, 1(2):81–125, 1994.

[8] Christoph Jacob. Weiterentwicklung eines werkzeuges zur verteilten, kollaborativen softwareentwicklung. Master's thesis, Freie Universität Berlin, 2009.

[9] Alfons Kemper and André Eickler. *Datenbanksysteme*, chapter Optimistische Synchronisation, pages 327–328. Oldenburg Wissenschaftsverlag GmbH, 2006.

[10] Jim D'Anjou; Scott Fairbrother; Dan Kehn; John Kellerman; Pat McCarthy. *Java Developer's Guide to Eclipse, The, 2nd Edition*. Number isbn: 3-8273-2254-5 in Java/Eclipse. Addison-Wesley Professional, 2 edition, Oct 26 2004.

[11] Oliver Rieger. Weiterentwicklung einer eclipse-erweiterung für verteilte paar-programmierung im hinblick auf kollaboration und kommunikation. Master's thesis, Freie Universität Berlin, 2008.

[12] Edna Rosen. Verteilte paarprogrammierung im industriellen umfeld etablierung und qualitative empirische untersuchung. Master's thesis, Freie Universität Berlin, 2009.

[13] Peter Saint-Andre. Streaming xml with jabber/xmpp. Technical report, IEEE Internet Computing, 2005.

[14] TAS SÓTI. Einladungsprozess in saros. Master's thesis, Freie Universität Berlin, 2009.

[15] Andrew S. Tanenbaum. *Verteilte Betriebssysteme*. Number ISBN 3-930436-23-X. Prentice Hall, 1995.

[16] Prof. Dr. Michael Weber. Synchronisation und koordination, 1999.

[17] Wikipedia. Hook (edv) — wikipedia, die freie enzyklopädie, 2010. [Online; Stand 13. Oktober 2010].

[18] Sebastian Ziller. Behandlung von nebenläufigkeitsaspekten in einem werkzeug zur verteilten paarprogrammierung. Diplomarbeit, Freie Universität Berlin, October 2009.

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis