IDE-Unterstützung für graduelle und individuelle Spracherweiterung

Eine Untersuchung am Beispiel von Eclipse, Java und VJ

Diplomarbeit
von
Marek Walczak
1walczak@informatik.uni-hamburg.de

Angefertigt am
Department Informatik
der
Universität Hamburg

Betreuer:
Dr. Schmolitzky
Prof. Dr. Lamersdorf

Ich möchte allen meinen Betreuern für Ihre Unterstützung danken. Besonderer Dank geht an Dr. Axel Schmolitzky, mit dem ich zahlreiche sehr fruchtbare Diskussionen hatte. Außerdem möchte ich Niels Kausche, Torsten Schoen und Matthias Zeimer für ihren Einsatz beim Testen und ihr Feedback danken. Schließlich möchte ich Emma Nishimoto danken, ohne deren Unterstützung diese Arbeit länger gedauert hätte.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1. Einleitung

Programmiersprachen entwickeln sich fortwährend und werden an neue Anforderungen angepasst. Diese Entwicklung ist maßgeblich davon getrieben den Programmierer in seinem Aufgabenfeld besser zu unterstützen. Er soll sich viel mehr mit seiner Problemstellung selbst, als mit der Programmiersprache befassen [HJ74]. *Graduelle Erweiterungen* einer Programmiersprache zeichnen sich dadurch aus, dass sie neue Konzepte dem Programmierer zugänglich machen und ihn damit besser bei der Entwicklung unterstützen können. Die Erweiterung betrifft dabei nur einen eingeschränkten Bereich der Sprache. Die meisten anderen Konzepte bleiben davon unberührt. Die Bezeichnung *individuelle Spracherweiterung* bezieht sich auf Spracherweiterungen, die unabhängig von der Hauptversion der Sprache entwickelt werden und damit nicht in die Sprachspezifikation direkt eingehen.

Ein Beispiel für eine graduelle und individuelle Spracherweiterung zu Java ist VJ (bedeutet soviel wie *Value Java*). VJ wurde im Rahmen einer Diplomarbeit [Rat06] entworfen und spezifiziert. Java ist primär objekt-orientiert und besitzt keinen dedizierten Mechanismus für eigene Werttypen. Man kann dies über verschiedene Konstrukte implementieren. Dies ist aber oft mit großem, zusätzlichem Aufwand verbunden und die Benutzbarkeit ist schwierig. VJ soll diese Lücke schließen und dem Programmierer die Arbeit an solchen Konstrukten abnehmen.

1.1. Problembeschreibung

Um den Entwicklungs- und Wartungsprozess für Software zu beschleunigen, werden integrierte Entwicklungsumgebungen (Integrated Development Environment, kurz IDE) wie Eclipse benutzt. In einer IDE sind verschiedene Werkzeuge für die Entwicklung mit einer Programmiersprache miteinander verknüpft und aufeinander angepasst. Die Aussage, dass ein Programmierer sich mit seiner Problemstellung und weniger mit der Programmiersprache befassen soll, kann auch auf die IDEs ausgeweitet werden. IDEs können den Programmierer erheblich bei der Entwicklung unterstützen. Er muss sich weniger mit einzelnen Entwicklungswerkzeugen und deren Zusammenspiel befassen. Im Gegensatz zu den 90er Jahren, als IDEs weniger verbreitet waren, ist die Unterstützung seitens einer IDE heutzutage für eine breitere Akzeptanz einer Spracherweiterung sehr wichtig. Die IDE-Unterstützung für eine Programmiersprache reift über mehrere Jahre. Nach Einführung von Java 5 beispielsweise musste Eclipse ca. ein Jahr an die Veränderungen von Java angepasst werden. Eine graduelle und individuelle Spracherweiterung ist mit diesem Problem genauso konfrontiert. Im Gegensatz zu einem generellen Versionswechsel einer Programmiersprache, für den die IDE in der Regel von den IDE-Entwicklern angepasst wird, muss die IDE für eine individuelle Spra-

cherweiterung eigenständig angepasst werden. Solche Spracherweiterungen sind zum Beispiel VJ oder das Modern Jass [Rie07]. Als Schwerpunkt werden in dieser Arbeit die Programmiersprache Java, die IDE Eclipse und die Spracherweiterung VJ gewählt. Die Wahl von Eclipse für das Fallbeispiel liegt darin, dass Eclipse weit verbreitet und frei verfügbar ist, und dadurch der breite Einsatz von VJ ermöglicht wird. VJ bietet im Vergleich zu Java in bestimmten Anwendungsgebieten einfacheren und leichter wartbareren Code. Die Vorteile von VJ werden durch das Fehlen einer IDE-Unterstützung im großen Maße aufgehoben. Eine vollständige Unterstützung von VJ als unabhängige Sprache durch Eclipse ist nicht sinnvoll, da es sich bei VJ lediglich um eine graduelle Erweiterung zu Java handelt. Eine vollständige Integration würde den Nachbau sämtlicher Werkzeuge, die bereits für Java existieren, erfordern.

1.2. Anforderungen

Es soll untersucht werden, wie gut die Unterstützung in Eclipse für graduelle und individuelle Spracherweiterungen ist. Dabei soll ein Plugin für Eclipse entwickelt werden, das VJ als Erweiterung zu Java in die IDE integriert. Beim Anwenden des Plugins soll der Programmierer ein Projekt weiterhin in Java entwickeln und nur an den Stellen entsprechende VJ-Anpassungen von Eclipse benutzen, wenn er Werttypen einsetzen will. Bei der Integration in Eclipse sollen unter anderem folgende Ziele berücksichtigt und Fragen geklärt werden:

- Der Entwickler soll sowohl Java-Quellcode als auch VJ-Quellcode weiterhin in der Java-Perspektive programmieren.
- Der VJ-Quellcode sollte in einem eigenen Editor bearbeitet werden, sodass "Highlighting"von Schlüsselwörtern möglich ist. Wie viel kann vom bereits existierenden Java-Editor übernommen werden?
- Die Übersetzung des VJ-Quellcodes soll innerhalb der IDE durchgeführt werden können und für den Benutzer möglichst transparent sein. Wie kann dabei der VJ-Compiler eingebunden werden?

Weiterhin sollte angesprochen werden, wie Refactoring-Möglichkeiten und Code Completion Methoden umgesetzt werden könnten, sodass die Handlichkeit einer IDE auch für VJ Entwickler zur Verfügung steht.

1.3. Aufbau der Arbeit

Da der Begriff der Werttypen unterschiedlich verstanden werden kann, wird in Abschnitt 2 eine genaue Definition für diese Arbeit festgelegt. Bereits exisitierende Ansätze zur Umsetzung von Werttypen bauen auf reinem Java auf. Kein Ansatz kann jedoch das Konzept von Werttypen vollständig umsetzen. VJ ist ein neuer Ansatz, der das Konzept als Spracherweiterung umsetzt.

In Abschnitt 3 werden verschiedene Spracherweiterungen mit einander verglichen. Hier werden die Begriffe individuelle und graduelle Spracherweiterung näher erläutert. Es wird auch die Unterstützung durch die IDEs für diese Spracherweiterungen besprochen.

Abschnitt 4 befasst sich wiederum direkt mit den IDEs und den Möglichkeit diese zu erweitern. Es werden die verschiedenen Plugin-Architekturen kurz vorgestellt und miteinander verglichen. Da Eclipse in dieser Arbeit als Fallbeispiel dient, wird die Architektur von Eclipse genauer untersucht.

Die Architektur von Eclipse bietet eine Großzahl an möglichen Ansätzen für eine Erweiterung dieser IDE um die Unterstützung für eine Spracherweiterung. Diese möglichen Ansätze werden in Abschnitt 5 vorgestellt und verglichen und eine Auswahl getroffen. Die Umsetzung anhand dieser Auswahl wird in Abschnitt 6 ausführlich dargestellt. Das während dieser Arbeit entstandene Plugin wurde in regelmäßigen Abständen von einer begrenzten Anzahl von Testpersonen benutzt. Die Ergebnisse und das Feedback sind in Abschnitt 7 zusammengetragen.

In Abschnitt 8 wird ein Ausblick auf mögliche andere Ansätze zur Umsetzung des Werttypkonzepts gegeben. Eine Zusammenfassung dieser Arbeit wird schliesslich in Abschnitt 9 gegeben.

2. Benutzerdefinierte Werttypen

In diesem Kapitel wird der Begriff der *benutzerdefinierten Werttypen* definiert und für diese Arbeit festgelegt. Benutzerdefinierte Werttypen sind grundlegend für die Sprache VJ, die im Abschnitt 3.4 näher vorgestellt wird. Sie sind die Motivation für die Entstehung von VJ. Zuerst wird im Abschnitt 2.1 der Bedarf nach benutzerdefinierten Werttypen motiviert. Im nachfolgendem Abschnitt 2.2 werden dann die Begriffe Werttyp und benutzerdefinierter Werttyp anhand von Eigenschaften genauer definiert. Dabei wird ein Vergleich mit *Objekttypen* durchgeführt, um eine Abgrenzung der beiden Begriffe zu schaffen. Es werden die Vorteile gezeigt, die sich aus der Benutzung von benutzerdefinierten Werttypen ergeben. Wegen dieser Vorteile wurden verschiedene Ansätze zur Umsetzung dieses Konzeptes bei der Benutzung heutiger objektorientierter Sprachen entwickelt. Diese Ansätze werden im Abschnitt 2.3 vorgestellt und diskutiert.

2.1. Motivation

Die am meisten genutzten Programmiersprachen unserer Zeit sind objektorientierte Sprachen wie C++ oder Java (s. [Sof08]). Die objektorientierte Sicht ist allerdings nicht immer ausreichend, um die Welt geeignet abzubilden [Rit03]. Die meisten objektorientierten Programmiersprachen verfügen deshalb auch über primitive Datentypen. Primitive Datentypen können aber nur einen beschränkten Bereich für Werttypen abdecken. So können damit keine wertähnlichen Typen wie Datum, komplexe Zahlen oder Geldbetrag (Betrag und Währung) umgesetzt werden.

Einige Programmiersprachen unterstützen zusätzlich zu den primitiven Datentypen auch aufzählbare Werttypen, sogenannte *Aufzählungstypen*. Diese können im Gegensatz zu primitiven Datentypen an ein Anwendungsbereich angepasst werden. Beispielsweise kann *Wochentag* damit modelliert werden. Sie bilden aber nur ein kleines und recht einfaches Teilgebiet der Werttypen ab.

Eine explizite Erweiterung einer Programmiersprache um weitere Datentypen wird keine Abhilfe schaffen. Es gibt viele unterschiedliche Anwendungsfelder, die eigene Werttypen benötigen. Für das Bankenwesen können als benötigte Werttypen *Geldbetrag* und *Kontonummer* aufgezählt werden. Aus dem Umfeld der Computernetzwerke ist z.B. die *IP-Adresse* zu nennen. Es ist nicht möglich mit vordefinierten Werttypen und den Aufzählungstypen alle Anwendungsfelder abzudecken. Deshalb sollten Werttypen ähnlich beliebig definierbar sein, wie es für Objekttypen der Fall ist. Da Werttypen sich zum Teil erheblich von Objekttypen unterscheiden, werden in Abschnitt 2.2 die Eigenschaften von Werttypen vorgestellt und mit Objekttypen verglichen.

2.2. Eigenschaften von Werttypen

Werttypen lassen sich anhand folgender Eigenschaften von Objekttypen unterscheiden. In dieser Auflistung werden Exemplare von Werttypen als *Wert* bezeichnet und Exemplare von Objekttypen als *Objekt*.

- Unveränderbarkeit - Werte können nicht geändert werden. Bei der Rechenoperation $2+3$ wird weder die 2 noch die 3 verändert sondern ein anderer Wert, die 5, zurück gegeben. Objekte hingegen können von Operationen verändert werden. Ein Exemplar vom Objekttyp `Konto` kann verschiedene Kontostände haben, wird aber das selbe Konto bleiben.

- Kein Erzeugen / Kein Vernichten - Werte werden weder erzeugt noch können sie vernichtet werden. Sie existieren von vornherein in ihrem eigenen *Werte-Universum*. Dort ist jeder Wert genau einmal enthalten, wie in einer mathematischen Menge. Der Zugriff auf einen Wert erfolgt mit Hilfe eines *Wert-Selektors*, der den gewünschten Wert liefert. Objekte müssen hingegen zur Laufzeit erzeugt werden und besitzen eine Lebensdauer nach der sie wieder vernichtet werden.

- Seiteneffektfrei - Werte verhalten sich seiteneffektfrei. Einerseits können sie nicht verändert werden, sodass mehrere Klienten-Klassen einen Wert benutzen können ohne sich durch diesen gegenseitig zu beeinflussen. Andererseits verhalten sich Werte referentiell transparent. Sie verändern mit ihren eigenen Operationen nicht den Zustand von Objekten. Dies bedeutet, dass in Werttypen keine Objekttypen benutzt werden können, da ein Wert sonst abhängig vom Zustand eines Objektes sein könnte. Auch Operationen auf Objekten sind nicht erlaubt, da Veränderungen von Objekten die referentielle Transparenz verletzen.

Werte und Objekte haben neben den beschriebenen Unterschieden auch Gemeinsamkeiten, die folgend aufgelistet sind:

- Typen - Werte und Objekte können beide als Exemplare von Typen angesehen werden. Der Typ beschreibt dabei die Gemeinsamkeiten, die die Exemplare inne haben. Beide Typarten sollten eine Schnittstelle nach außen besitzen, sodass die Interna gekapselt werden.

- Frei definierbar - Die für Objekttypen bekannte Freiheit bei der Definition ist auch für Werttypen möglich. Ausser den von einer Programmiersprache vordefinierten Typen sind auch benutzerdefinierte Typen, wie `Kontonummer` oder `Wochentag` möglich.

- Zerlegbar - Werttypen können sich genauso wie Objekttypen intern aus mehreren weiteren Werttypen aufbauen. Solche Werttypen heissen *strukturierte Werttypen*. Geldbetrag ist zusammengesetzt aus Betrag und Währung.

In der Tabelle 2.2 werden die oben aufgeführten Punkte zusammengefasst.

Vorallem wegen der Freiheit von Seiteneffekten können Werttypen ähnlich den primitiven Datentypen behandelt werden, was Klarheit bei der Benutzung mit sich bringt.

	Werttyp	Objekttyp
Veränderbar	-	+
Erzeugung	-	+
Seiteneffektfrei	+	-
Exemplare von Typen	+	+
Frei definierbar	+	+
Zerlegbar	+	+

Es stehen sowohl Werttypen (benutzerdefinierte Werttypen und primitive Datentypen) als auch Objekttypen dem Programmierer zur Verfügung. Dies gibt Möglichkeit sicherere und stabilere Programme zu entwickeln. Im folgendem Abschnitt 2.3 werden deshalb einige Ansätze vorgestellt und diskutiert, die das Konzept von benutzerdefinierten Werttypen in objektorientierten Programmiersprachen unterstützen.

2.3. Umsetzung von Werttypen in objektorientierten Programmiersprachen

Objektorientierte Sprachen besitzen keine dedizierte Möglichkeit eigene Werttypen zu definieren (s. [RRS07, Lange Version]). Deshalb müssen die Werttypen für diese Sprachen mit Hilfe von *Objektklassen* aufgebaut werden. Dazu gibt es verschiedene Ansätze: unveränderliche Objekte und Rahmenwerke, welche nachfolgend vorgestellt und diskutiert werden.

2.3.1. Unveränderliche Objekte

Der Ansatz der unveränderlichen Objekte ist eine simple und effektive Methode, um Werttypen zu realisieren. Wie der Name bereits sagt, dürfen die Objekte nicht verändert werden, d.h. sie besitzen keine manipulierenden Operationen oder die Operationen, welche versuchen das Objekt zu ändern, enden mit einem Laufzeitfehler [Blo01, item 13]. Auch muss darauf geachtet werden, dass keine Objekttypen in der Schnittstelle eines Werttyps benutzt werden. Die Exemplarvariablen müssen primitive Datentypen oder selbst unveränderliche Objekttypen sein. Der Konstruktor einer Klasse muss privat sein, und der Zugriff über Fabrikmethoden bzw. Fabriken gekapselt werden. Der Vergleich von zwei Werten kann nicht über den Operator `==` sondern muss über die `equals`-Methode durchgeführt werden. Diese Methode und dazu gehörend auch die `hashCode`-Methode [Blo01, item 8] muß vom Programmierer implementiert werden.

Der Vorteil dieses Ansatzes ist die Freiheit von Seiteneffekten. Zum Beispiel wurde die Klasse `java.lang.String` so entwickelt.

Leider erfüllt dieser Ansatz nicht alle Kriterien für Werttypen. Es können weiterhin mehrere Exemplare existieren, die den gleichen Wert darstellen. Es ist nicht möglich über die Referenz zwei Werte miteinander richtig zu vergleichen. Ein tiefer Vergleich

mit Hilfe der Methode `equals` ist nötig. Dieses Verhalten ist zwar von der `String`-Klasse bekannt, es ist aber nicht im Einklang mit den primitiven Datentypen. Ein weiterer Nachteil ist die mangelnde Unterstützung des Programmierers bei der Einhaltung der Konventionen. Der Compiler wird keine Warnung ausgeben, wenn der Programmierer die Konventionen für Werttypen bricht. Der Programmierer muss die Konventionen selbst einhalten. Das kann dazu führen, dass der Programmierer sich weniger mit dem Anwendungsfeld sondern wieder mehr mit der Umsetzung einer Lösung und der Sprache befasst. Die im folgenden Abschnitt 2.3.2 vorgestellten Rahmenwerke wurden entwickelt um genau diesem Problem zu begegnen. Sie sollen den Programmierer bei der Benutzung von benutzerdefinierten Werttypen unterstützen.

2.3.2. Rahmenwerke

Die Benutzung von Rahmenwerken ist ein weiterer Ansatz zur Unterstützung des Programmierer beim Einsatz von Werttypen. Wie im vorherigen Abschnitt dagestellt, soll der Einsatz von Rahmenwerken die Benutzung von Werttypen vereinfachen. Der Programmier soll nicht mehr das ganze Werttyp-Konzept manuell programmieren. Hier wird das JWAM-Rahmenwerk vorgestellt. JWAM ist ein Rahmenwerk, welches in zwei Versionen, JWAM1 (beschrieben in [Mül99]) und JWAM2 vorliegt. In JWAM1 wird eine Fabrik für die Kontrolle von Werten benutzt. Sie hält alle Referenzen vor, sodass Kopien von Exemplaren vermieden werden können.

JWAM1 ist eine sehr große Bibliothek, die aufwändig zu handhaben ist und eine große Fülle an Konzepten enthält. Somit ist eine Anwendung sehr schnell und unnötig aufgebläht. Eine Werttypklasse muss von einer Rahmenwerksklasse erben. Diese Rahmenwerksklassen sind sehr groß und enthalten viel Funktionalität, die bei vielen Werttypen nicht benötigt wird. So ist zum Beispiel die Fabrikklasse als innere Klasse in der Werttypklasse enthalten. Der Programmier muss diese Klassen sehr gut kennen, um sie korrekt zu beerben.

JWAM1 bietet einige Vorteile im Gegensatz zur manuellen Umsetzung. So kann innerhalb der Klienten-Klassen mit jedem Wert so umgegangen werden, als ob es kein weiteres Exemplar mehr gäbe. Der Vergleich über die Referenz wird i.d.R. korrekt funktionieren. Bei De-/Serialisierung muss aber aufgepasst werden, dass kein zweites Exemplar des Wertes erzeugt wird.

Das beerben einer Rahmwerksklasse sichert aber nicht zu, dass keine verändernden Operationen in der Werttypklasse enthalten sind. Es kann auch nicht verhindert werden, dass Exemplarvariablen Objekttypen sind. JWAM1 setzt das Konzept von Werttypen nicht vollständig um. Vorallem die Seiteneffektfreiheit ist nicht zugesichert. Zusätzlich ist es sehr groß und schwierig zu handhaben.

JWAM2 sollte ein möglichst einfaches Konzept bieten und nicht so schwergewichtig sein. Es beinhaltet hauptsächlich Konventionen, wie eine Fachwertklasse auszusehen hat. Dafür gibt es ein Markerinterface `DomainValue` und auch weiterführende Konzepte, wie z.B Factory-Interfaces [Hei05]. Das erben von Rahmenwerksklassen ist nicht mehr vorgesehen, sodass die Werttypklassen viel kleiner und handhabbarer sind.

Bei JWAM2 wird nicht mehr sichergestellt, dass es nur noch ein Exemplar eines Wertes gibt. Zwar gibt es die Möglichkeit Werteuniversen auf Prototypen-Basis zu be-

nutzen, diese sind allerdings optional. Zusätzlich kommt dazu, dass bei diesen Universen die Felder eines Wertes veränderbar sein müssen und damit die Unveränderbarkeit eines Wertes viel aufwendiger sicherzustellen ist. JWAM2 ist viel einfacher und leichter in der Anwendung als JWAM1. Allerdings kann auch dieses Rahmenwerk das Konzept der Werttypen nicht vollständig umsetzen. Es wird auch hier der Zugriff auf Objekte nicht überprüft. Veränderbare Operationen sind laut Konvention nicht erlaubt. Es liegt aber am Programmierer, ob diese eingehalten wird. Genauso muss der Programmierer die `equals`-Methode selbst schreiben, da ein Vergleich über die Referenz nicht möglich ist.

2.4. Zusammenfassung

In diesem Kapitel wurde der Werttyp-Begriff eingeführt. Bei sinnvoller Benutzung bieten Werttypen erhöhte Sicherheit, Klarheit und Stabilität bei der Anwendungsentwicklung. Objektorientierte Programmiersprachen bieten nur primitive Datentypen, aber keine Möglichkeit benutzerdefinierte Werttypen zu definieren. Aus diesem Grund wurden verschiedene Ansätze entwickelt, um das Konzept der Werttypen umzusetzen. Keiner der Ansätze kann das Konzept der Werttypen vollständig umsetzen. Rahmenwerke, wie JWAM, unterstützen den Programmierer nur teilweise. Die vielen Unzulänglichkeiten der bestehenden Ansätze stammen hauptsächlich davon, dass Werttypen über Objekttypen emuliert werden. Ein für Werttypen dedizierter Klassentyp kann diesem Problemen entgegenkommen. Doch für die Unterstützung eines neuen Klassentyps muss eine Programmiersprache um dieses Konzept erweitert werden. VJ geht diesen weg. Für eine genauere Einordnung von VJ werden in Abschnitt 3 deshalb verschiedene Spracherweiterungen vorgestellt und miteinander verglichen.

3. Spracherweiterungen

Die Entwicklung von Programmiersprachen wird oft davon getrieben den Programmierer bei bestimmten Anwendungsfeldern besser und einfacher zu unterstützen. Deshalb werden Programmiersprachen immer wieder um neue Konzepte erweitert. Sprachdesigner zielen darauf dem Programmierer besseres Werkzeug zur Verfügung zu stellen, sodass dieser sich mehr mit dem Anwendungsfeld als mit der Programmiersprache selbst befasst [HJ74].

Ein solches Konzept sind die benutzerdefinierten Werttypen, die in Abschnitt 2 vorgestellt wurden. Die verschiedenen Ansätze zur Umsetzung von Werttypen beschränkten sich auf die bereits vorhandenen Konzepte der Sprache. In diesem Abschnitt werden im Gegensatz dazu verschiedene Spracherweiterungen vorgestellt, die neue Konzepte für eine Programmiersprache einführen. VJ, eine dieser Spracherweiterungen, ist explizit für die Unterstützung von Werttypen entworfen worden. Weitere, hier vorgestellte Erweiterungen sind nicht mit dieser Zielsetzung entwickelt worden. Dadurch wird ein breiterer und allgemeinerer Blickwinkel beibehalten, der sich nicht nur auf das Gebiet der Werttypen beschränkt.

Die Spracherweiterungen können unter anderem anhand von zwei Eigenschaften unterschieden werden. Anhand dieser Unterscheidung kann VJ besser innerhalb der weiteren Spracherweiterungen eingeordnet werden. So kann eine Erweiterung daraufhin untersucht werden, ob sie individuell ist. Individuelle Erweiterungen werden nur von einer begrenzten Gruppe von Entwicklern benutzt. Oftmals sind es Erweiterungen, die wissenschaftlichen Zwecken dienen. Diese Erweiterungen sind nicht Teil der offiziellen Spezifikation der Programmiersprache und damit nicht in der Hauptversion enthalten. Die zweite Eigenschaft, die untersucht werden kann, ist die Gradualität einer Erweiterung. Diese Eigenschaft ist schwieriger festzustellen. Es muss untersucht werden, wie tiefgreifend die Erweiterung die ursprüngliche Sprache verändert. Graduelle Erweiterungen zeichnen sich dadurch aus, dass sie die Sprache nicht grundsätzlich durchziehen und verändern, sondern um ein neues Konzept erweitern. Die Benutzung der Konzepte der Erweiterung ist nicht notwendig für das korrekte Ablaufen eines Programmes. Die neuen Konzepte können allerdings explizit genutzt werden, um z.B. ein besser wartbareres Programm zu entwickeln. Bei nicht graduellen Erweiterungen ist dies nicht möglich. Diese Erweiterungen sind so tiefgreifend, dass auf die Benutzung der Erweiterung nicht verzichtet werden kann.

3.1. Modern Jass

Java besitzt nur ein schwaches Konzept zur Unterstützung des Vertragmodells. *Modern Jass* [Rie07] versucht diese Lücke mit Java-eigenen Mitteln zu schliessen. *Modern Jass*

benutzt die Java 5 *Annotations* und *Instrumentation API*, sowie die Java 6 *Compiler API*, um das Vertragsmodell zu unterstützen.

Zur Festsetzung der Verträge werden *Annotations* benutzt. Die einzelnen Annotationen werden als *Modifier* für Methoden, die durch das Vertragsmodell gesichert werden sollen, platziert. Die Überprüfung der Verträge kann sowohl beim Eintritt in die Methode als auch beim Verlassen der Methode stattfinden. Auf diese Weise können sowohl Vorbedingungen als auch Nachbedingungen überprüft werden.

Bei der Übersetzung einer Klasse werden *Annotation Processor*en aufgerufen und ihnen die Klasse zur Auswertung übergeben. Dieser Vorgang ist nur lesend, deshalb erstellt *Modern Jass* ein externes und angereichertes Modell der Klasse. Die Anreicherungen beinhalten Methoden, die aus den Annotationen generiert wurden. Diese Methoden dienen zur Überprüfung der Verträge und werden mit der *Java Compilation API* als reines Java erzeugt.

Die generierten Prüfmethoden sind noch nicht mit den entsprechenden Klassen assoziert. Dies geschieht zur Ladezeit. Dabei wird die *Instrumentation API* benutzt, mit der Binärdaten in Methoden eingefügt werden. Die Methoden, die mit Annotationen versehen sind, werden mit Aufrufen der Prüfmethoden angereichert. Zur Laufzeit stellen die Prüfmethoden sicher, dass die Verträge eingehalten werden.

Modern Jass ist einfach in den Übersetzungsprozess einzubinden. Es muss sich auf dem Klassenpfad befinden. Einträge in der Manifest-Datei (s. Abschnitt 4.3.2) der Bibliothek melden den eigenen *Annotation Processor* beim Java Compiler als Plugin an. Dies ermöglicht eine breites Einsatzgebiet und schränkt diese Lösung nicht auf eine Entwicklungsumgebung ein.

Diese Erweiterung zu Java ist nicht weit verbreitet und nur auf eine wissenschaftliche Gruppe eingeschränkt. Wegen dieser Einschränkung kann *Modern Jass* als eine individuelle Erweiterung kategorisiert werden. Weiterhin kann diese Erweiterung als graduell eingestuft werden. Die Konzepte greifen nicht in die originallen Konzepte von Java ein und sind damit nicht grundlegend.

3.2. Generics

Mit Java 5 wurden viele neue Konzepte eingeführt. Eines davon sind die *Generics*. Die ersten Ansätze für *Generics* wurden bereits kurze Zeit nach der Erstveröffentlichung von Java vorgestellt. *Pizza* [OW97] ist der bekannteste Ansatz, der über *GJ* [BOSW98] sieben Jahre später in der Sprachspezifikation von Java [GJSB05] mündete. *Generics* führen Parametrisierung für Typen in Java ein. Operationen an Behältern, wie `Collection` oder `List`, können auf bestimmte Element-Typen beschränkt werden. Bereits zur Übersetzungszeit kann die Typsicherheit überprüft werden und für mehr Sicherheit sorgen. Unsichere *Type-Casts* sind nicht mehr nötig. Gleichzeitig ist der Quelltext aussagekräftiger.

Intern werden *Generics* vom Java Compiler auf die Java Version 1.2 übersetzt. Diese Zurückstufung ist dadurch bedingt, dass Generics bereits in Java 1.3 eingebaut waren, aber erst mit Java 5 freigegeben wurden. Der Java Übersetzer führt einen internen zusätzlichen Übersetzungsschritt dafür durch. Nach dem die Überprüfung der Typen

stattgefunden hat werden die Typparameter wieder entfernt. Weiter wird der nicht parametrisierte Standard-Typ benutzt.

Diese Erweiterung ist eine der umfangreichsten, die an Java durchgeführt wurde. Viele Werkzeuge, u.a. Eclipse mit dem eigenen Java Übersetzer, konnten erst Monate später nach erscheinen der neuen Spezifikation die volle Kompatibilität vorweisen. Die fehlende Unterstützung für die gesamte Sprache würde für ein Werkzeug die Abkehr der Anwender bedeuten. *Generics* sind eine sehr grundlegende Erweiterung der Java Sprache. Ursprüngliche Konzepte werden verändert. Dies zieht Veränderungen an der ganzen Sprache mit sich, sodass diese Erweiterung nicht als eine graduelle angesehen werden kann. Die ursprünglichen Ansätze, *Pizza* und *GJ*, waren noch individuell. Die Vorteile von parametrisierten Typen haben *Generics* zum Einzug in die offizielle Spezifikation verholfen. Der Einsatz ist nicht mehr auf eine kleine Gruppe beschränkt, sodass *Generics* keine individuelle Erweiterung ist.

3.3. AspectJ

AspectJ erweitert Java um das Konzept der Aspektorientierten Programmierung (AOP). AOP wurde ursprünglich am Xerox PARC entwickelt und ist unabhängig von der Programmiersprache. AspectJ war eine der ersten Implementierungen und wurde von dem gleichen Team am Xerox PARC entwickelt. Da AspectJ eine Obermenge zu Java ist [Lad03, s. 33], sind alle korrekten Java Programme auch korrekte AspectJ Programme.

Mittlerweile ist AspectJ als eigenständiges Projekt in der Eclipse-Organisation aufgenommen worden. Damit stehen für die Weiterentwicklung und Wartung dieser Programmiersprache Fördergelder und Entwickler zur Verfügung. Gleichzeitig kann davon ausgegangen werden, dass die Integration von AspectJ in Eclipse deshalb besser durchgeführt wird als für andere Entwicklungsumgebungen [Böh06, s. 34]. Damit ist AspectJ keine individuelle Spracherweiterung. Wegen der starken Änderung des Programmierparadigmas, kann von AspectJ als einer eigenen Sprache gesprochen. Es ist auf jedenfall keine graduelle Erweiterung zu Java.

AspectJ ist eine Spracherweiterung zu Java, die eine neue Syntax einführt. Für die Unterstützung von AspectJ in Eclipse wurden die *Java Development Tools* (JDT) als Muster gewählt. Dieser Schritt bietet die Konformität mit den JDT, sodass AspectJ in Eclipse ähnlich zu handhaben ist wie Java.

Intern bauen die *AspectJ Development Tools* (AJDT) für Eclipse viel Funktionalität von den JDT nach. Die JDT bieten nur wenig und hauptsächlich abstrakte Funktionalität offen an. Die Klassen, die Java-spezifische Implementierungen enthalten, sind verborgen. Aus diesem Grund werden oftmals die Schnittstellen von den AJDT nicht eingehalten. Dies ist riskant, da die Implementierung bzw. die interne Schnittstelle sich mit neuen Versionen des JDT verändern und damit das Plugin nicht benutzbar machen kann.

Auf der anderen Seite wird dem Plugin sehr viel Funktionalität bereits geliefert, ohne dass diese zusätzlich implementiert werden muss. Dies sichert eine schnelle und mit den JDT übereinstimmende Einbindung in Eclipse.

	Individuell	Graduell
Modern Jass	*	*
Generics	-	-
AspectJ	-	-
VJ	*	*

Tabelle 3.1.: Zuordnung der Spracherweiterungen zu den Begriffen: individuell und graduell

3.4. VJ

Wie bereits in Kapitel 2 beschrieben, können Werttypen vorteilhaft bei der Anwendungsentwicklung sein. Die verschiedenen Ansätze zur Umsetzung des Werttyp-Konzepts erfüllen nicht alle Kriterien und z.T. sind überdimensioniert. Wegen dieser Unzulänglichkeiten wurde mit VJ ein anderer Weg gegangen. VJ führt die benutzerdefinierten Werttypen als einen eigenen Sprachbestandteil ein und ändert dabei nichts an den bestehenden Java-Konzepten. Somit ist VJ eine Obermenge der Sprache Java und jedes korrekte Java Programm ist auch ein korrektes VJ Programm.

Ähnlich wie AspectJ führt VJ neue Syntax ein, um das Werttyp-Konzept umzusetzen. Dabei ergänzen zwei neue Schlüsselwörter die Java Syntax. `const` ist der Wertselektor über den die Referenz auf einen Wert zurückgegeben wird. `valueclass` bezeichnet eine Werttyp-Klasse und ist das Analogon zu `class`, welches wiederum Klassen von Objekttypen bezeichnet.

Im Rahmen der Diplomarbeit [Rat06] wurde ein Compiler für VJ entwickelt. Dieser übersetzt VJ in Java kompatiblen Quelltext. Diesen Quelltext muss dann ein Java Compiler in Java Binärcode nachträglich übersetzen. Diese Design-Entscheidung wurde getroffen, um mit dem VJ Compiler möglichst gut das Werttyp-Konzept zu unterstützen. Die Übersetzung von Java Quelltext nach Java Binärcode ist in diesem Fall nicht entscheidend und wäre unnötig. Da VJ auf Java zurückübersetzt wird und reines Java das Werttyp-Konzept nicht unterstützt, muss der VJ-Compiler dieses Verhalten bei der Übersetzung sicherstellen. Die Werttyp-Klassen werden als nicht veränderbare Objekttyp-Klassen ausgegeben. So dürfen keine verändernden Methoden enthalten sein und die Felder der Klasse werden mit dem Modifier `final` als nicht veränderbar markiert. Weiterhin wird überprüft, dass die Felder keine Objekttypen sind. In VJ ist der Vergleich von zwei Werten über die Referenz mit Hilfe des Operators `==` möglich. Dieser Vergleich muss in Java mit Hilfe eines tiefen Vergleichs realisiert werden. Der VJ-Compiler übersetzt den Operator in eine `equals` Methode, die an den jeweiligen Werttyp angepasst ist. Durch den zusätzlichen Übersetzungsschritt kann geprüft werden, ob die Werttyp-Konventionen eingehalten werden. Auf diese Weise wird das Werttyp-Verhalten auch nach der Übersetzung in Java weiterhin sichergestellt. Da VJ die Konzepte von Java nicht ändert und nur neue einführt kann diese Erweiterung von Java als graduell kategorisiert werden. Die Erweiterung kann auch als individuell angesehen werden, da sie nur von einer kleinen Gruppe im universitären Bereich und

nicht im größeren Rahmen eingesetzt wird.

Wie bereits erwähnt übersetzt der VJ-Compiler VJ nicht direkt in Java Binärcode, wie dies der Compiler von AspectJ tut. Auch unterstützt der VJ-Compiler nicht alle Konzepte von Java. Zwar ist VJ in der Theorie eine Obermenge von Java, aber mangels eines vollständigen Compilers ist die praktische Umsetzung nur eingeschränkt möglich. Zusätzlich ist der VJ-Compiler nur von der Kommandozeile aufrufbar. Heutzutage werden Programme im großen Maße innerhalb von IDEs entwickelt. Die Bedienung des Compilers über die Kommandozeile ist ein Rückschritt. Der Übersetzungsschritt von VJ nach Java ist weder integriert mit den weiteren Werkzeugen und Schritten noch ist die Bedienung bei der Übersetzung größerer Projekte einfach. In diesem Fall sind die Aussichten auf eine größere Verbreitung von VJ nahezu nicht vorhanden. Die Einbindung von VJ in eine IDE ist ein erforderlicher Schritt, den auch andere Erweiterungen gehen müssen. Diese Arbeit befasst sich mit diesem Problem und stellt die Möglichkeiten für die Einbindung einer Spracherweiterung in eine IDE im Kapitel 5 beispielhaft an VJ dar.

3.5. Zusammenfassung

In diesem Kapitel wurden ausgewählte Spracherweiterungen vorgestellt. Anhand dieser Spracherweiterungen wurden die Begriffe individuell und graduell als Eigenschaften einer Spracherweiterung eingeführt. Die Tabelle 3.1 zeigt die Zuordnungen der einzelnen Spracherweiterungen zu diesen Begriffen. Eine der vorgestellten Spracherweiterungen ist VJ. VJ ist eine graduelle und individuelle Spracherweiterung zu Java. Im Gegensatz zu den in Abschnitt 2.3 beschrieben Ansätzen zur Umsetzung des Werttyp-Konzepts kann VJ alle Konvetionen dieses Konzeptes zusichern. Heutzutage werden Anwendungen innerhalb von IDEs entwickelt, welche an die Spracherweiterungen angepasst werden müssen. Ansonsten ist der Einsatz der Spracherweiterung bei der Entwicklung kaum möglich. In Abschnitt 4 werden deshalb die gängigen IDEs näher vorgestellt und ihre Erweiterbarkeit untersucht.

4. Entwicklungsumgebungen

In [Boe03] wird die Überfüllung von IDEs mit zu vielen Werkzeugen bemängelt. Diese Werkzeuge nehmen Ressourcen weg sind aber oftmals nicht ausgereift genug, sodass stattdessen andere, nicht in die IDE integrierte Werkzeuge benutzt werden. Werkzeuge müssen eine gute Unterstützung für den Entwickler sein, um nicht an der Akzeptanz zu scheitern. Sie müssen homogen zu bedienen sein, und keine heterogene Landschaft von zusammengewürfelten Werkzeugen bilden. Die Integration und Interaktion muss soweit gewährleistet sein, dass keine Brüche in der Bedienung unterschiedlicher Tools für den Benutzer entstehen. Eine IDE muss Konzepte bieten, um mit neuen Werkzeugen leicht erweiterbar zu sein. Auch bestehende Werkzeuge sollten erweiterbar sein. Beispielhaft kann man einen Editor nennen, der sich am besten identisch sowohl für Java als auch für VJ Quelltexte verhalten sollte. IDEs wie IDEA oder Eclipse bieten bereits Konzepte zur Erweiterung um neue Sprachen. Im Abschnitt 4.1 werden diese Ansätze näher vorgestellt und diskutiert wieso sie für Spracherweiterungen nicht geeignet sind. Bei der Erweiterung von IDEs um neue Sprachkonzepte kann deshalb nicht auf diese zurückgegriffen werden, die Erweiterung muss vollständig selbst entwickelt werden. Eine gute, flexible und mächtige Unterstützung für Erweiterungen kann die Entwicklung einer Erweiterung erheblich vereinfachen. In Abschnitt 4.2 werden die Plugin-Architekturen von Eclipse, IDEA und Netbeans unter diesem Gesichtspunkt miteinander verglichen. Die Wahl von Eclipse für eine genauere Untersuchung in dieser Arbeit wird im Abschnitt 4.3 motiviert.

4.1. Bestehende Ansätze zur Erweiterung von IDEs mit neuen Sprachkonzepten

Entwickler von IDEs haben bereits Ansätze ausgearbeitet um IDEs um neue Sprachkonzepte bzw. Sprachen zu erweitern. Dazu zählen die *IDEA Language API*[Jem05], *Dynamic Langauges Toolkit* (DLTK) [EF08] von Eclipse und das *Meta Programming System* (MPS) [Dmi05].

Die Language API von IDEA ist ein Rahmenwerk, mit dessen Hilfe die Unterstützung für neue Programmiersprachen in die IDEA-IDE eingebaut werden kann. Für jede neue Sprache müssen entsprechende Analyse-Tools wie *Lexer* und *Parser* geschrieben und bereitgestellt werden. IDEA stellt abstrakte Klassen mit denen eigene Tools einfacher entwickelt werden können. Die Erweiterung bereits vorhandener Tools ist nicht möglich. So können Java-Tools nicht um VJ-Konzepte erweitert werden. Man kann VJ nur als eine eigenständige Sprache einbinden.

Das Dynamic Languages Toolkit von Eclipse nimmt den gleichen Ansatz. Auch

hier gibt es nur die Möglichkeit Sprachen als ganzes einzubinden. Die Erweiterung vorhandener Tools ist nicht möglich.

Diese beiden Ansätze sind nicht geeignet. Das Einbinden einer Erweiterung als eine vollständig neue Sprache führt zu:

- Heterogenen Tools für die ursprüngliche Sprache und die Spracherweiterung
- Schlechterer Interaktion zwischen den Tools der ursprünglichen Sprache und der Erweiterung
- Erheblichem Mehraufwand bei der Entwicklung der Unterstützung für die Spracherweiterung.

Das Meta Programming System ist ein vollkommen anderer Ansatz. Der Entwickler entwickelt in dieser IDE kein Programm, sondern die Sprachen in denen er das Programm später schreiben wird. Die Idee ist, für jeden Bereich des Programmes eine eigene Sprache zu haben, die für diesen Bereich am besten geeignet ist. So kann z.B. für die Logik des Programmes eine spezielle Sprache sehr gut geeignet sein, für die Entwicklung der Benutzungsoberfläche kann sie aber schlecht einsetzbar sein. Dem Entwickler steht es frei eine eigene Sprache für die Entwicklung der Benutzungsoberfläche zu entwickeln. Ein Programm kann so in verschiedenen Sprachen geschrieben sein, von denen jede an die entsprechenden Aufgaben angepasst ist.

Das MPS wurde maßgeblich von Sergey Dmitriev von Jetbrains entwickelt. IDEA stammt auch aus diesem Hause. Das MPS ist an IDEA gekoppelt, sodass die in MPS neu entwickelten Sprachen in IDEA benutzt werden können. Auch bei diesem Ansatz ist es nicht möglich eine bestehende Sprachunterstützung um neue Sprachkonzepte zu erweitern, es wird stattdessen die Entwicklung neuer spezialisierter Sprachen unterstützt.

Ein weiterer Ansatz ist in der Studienarbeit [Hei05] von Markus Heiden beschrieben. Dieser unterscheidet sich grundlegend von den oben genannten. Es ist kein Ansatz, der das Erweitern einer IDE um neue Sprachkonzepte vereinfacht. Die Arbeit von Markus Heiden beschäftigt sich mit der Unterstützung von Werttypen in Eclipse. Dem Entwickler wird ein Tool geboten, welches Werttypklassen generiert. Das Tool fragt dazu zuerst Daten des gewünschten Werttyps ab. Anhand dieser Daten erstellt es eine Java-Klasse, die eine Werttyp-Klasse ist. Bei diesem Ansatz wird sichergestellt, dass die generierte Klasse keine der Regeln verletzt. So erzeugt der Generator z.B. keine verändernden Methoden. Allerdings kann bei diesem Ansatz nicht überprüft werden, ob Werte in Klienten-Klassen korrekt benutzt werden. Ein Vergleich über die Referenz kann nicht stattfinden, es muss ein Tiefenvergleich über die `equals`-Methode durchgeführt werden. Weiterhin ist der generierte Quelltext Java, welches mit Java-Tools geöffnet und bearbeitet wird. Diese kennen aber den `Wertekontext` für diese Klassen nicht, und werden bei Änderungen am Quelltext den Entwickler nicht warnen, wenn dieser gegen Konventionen verstößt.

Die bisherigen Ansätze geben nicht die Möglichkeit die IDEs für eine Sprache um neue Konzepte zu erweitern. Die Unterstützung muss vollständig neu entwickelt werden. Auch wenn die IDEs die Erweiterung um neue Sprachen unterstützen ist dieser

Prozess sehr zeitintensiv und kann viele Spracherweiterungen daran hindern verbreitet zu werden. Es muss nämlich nicht nur die Unterstützung für die Erweiterung entwickelt werden, sondern auch die ganze Unterstützung für die ursprüngliche Sprache nachgebaut werden.

Ein anderer Ansatz muss gewählt werden. Die Unterstützung für eine Spracherweiterung muss selbst entwickelt werden. Dabei wird darauf geachtet, dass möglichst wenig von der Unterstützung für die ursprüngliche Sprache nachgebaut wird. Entscheidend ist hier die Plugin-Architektur. Diese muss die Möglichkeit bieten bereits bestehende Funktionalität nicht nur zu ersetzen sondern auch zu erweitern. Die Plugin-Architekturen der gängigen IDEs werden im Abschnitt 4.2 untersucht.

4.2. Vergleich von Plugin-Architekturen

Da die Unterstützung für eine Spracherweiterung selbst entwickelt werden muss, sollte eine gute Plugin-Architektur der IDE vorhanden sein. Zuerst wird die Flexibilität der Plugin-Architekturen miteinander verglichen.

Plugins können in der Konfiguration einer IDE deaktiviert werden, ohne dass sie gelöscht werden müssen. Diese Plugins werden beim Start und beim Nachladen von benötigten Komponenten ignoriert und können durch geeignetere ersetzt werden. Diese Möglichkeiten besitzen sowohl Eclipse, IDEA und Netbeans. Im Gegensatz zu Netbeans sind Eclipse und IDEA in der Lage benötigte Plugins erst dann zu laden, wenn diese benötigt werden. Dies beschleunigt den inititalen Startprozess entscheidend und ist ressourcenschonender. Von dem Nachladen muß das Hotplug unterschieden werden. Hierbei ist die Möglichkeit gemeint, ob ein Plugin zur Laufzeit eingebunden und benutzt werden kann, ohne dass es zur Startzeit der Entwicklungsumgebung bekannt gewesen ist. Hier ist IDEA nicht in der Lage ohne einen Neustart ein Plugin einzubinden.

Das Deaktivieren von Plugins, Hot Plug und Nachladen von Plugins auf Nachfrage haben einen Schwerpunkt auf dem Ressourcenmanagment, und der daraus resultierenden Flexibilität beim Laden von Plugins. Die im folgenden aufgezählten Eigenschaften sind eher allgemeinerer Art, die für diese Arbeit wichtig sind.

	Eclipse	Netbeans	IDEA
De-/Aktivieren von Plugins	*	*	*
Nachladen auf Anfrage	*	-	*
Hot Plug[1]	(*)	(*)	-
Eigene Debug Instanz	*	*	*
Java Compiler	Eclispe	javac, jikes, Eclipse	javac, jikes, Eclipse
Extension Points	*	-	-

Tabelle 4.1.: Vergleich der Java IDEs

[1] Manche Plug-ins erfordern einen Neustart von Eclipse und Netbeans

Um die Funktion eines Plugins testen zu können, bieten alle Entwicklungsumgebungen dafür die Möglichkeit das Plugin in einer sicheren Umgebung zu prüfen. Dafür wird eine eigene IDE-Instanz gestartet, sodass nur Einfluss von der Entwicklungsumgebung heraus auf die Testinstanz ausgeübt werden kann und nicht andersherum. So kann gezielt die Funktion des Plugins untersucht werden und andere Aufgaben, die in der Entwicklungsumgebung ablaufen, interagieren nicht direkt mit der Testinstanz.

Im Gegensatz zu IDEA und Netbeans besitzt Eclipse einen eigenen Java Compiler, der Teil der JDT ist. Dieser Compiler ist vollkommen kompatibel zur Spezifikation der Java Programmiersprache, besitzt aber zusätzliche Möglichkeiten, wie inkrementelles Übersetzen, wobei nur Teilstücke des Quelltextes übersetzt werden. Eclipse ist auf diesen Compiler angewiesen. IDEA und Netbeans haben keinen eigenen Compiler, sondern benutzen für alle Aufgaben einen externen Compiler, der kompatibel zur Sprachspezifikation von Java ist.

Eclipse führt eine eigene Idee für Erweiterungen über *Erweiterungspunkte* (s. Abschnitt 4.3.1) ein. IDEA und Netbeans können nur um weitere Eigenschaften erweitert werden, wohingegen Eclipse über Erweiterungspunkte die Möglichkeit gibt bestehende Eigenschaften an eigene Bedürfnisse anzupassen oder zu erweitern.

Von den hier betrachteten IDEs soll Eclipse, wegen der sehr starken Wandlungsfähigkeit, für eine genauere Untersuchung in Betracht gezogen werden. Einen großen Einfluß auf die Auswahl hat auch die freie Verfügbarkeit und die starke Verbreitung (s. Abschnitt 4.3) von Eclipse Diese Wandlungsfähigkeit baut auf der Plugin-Architektur mit Erweiterungspunkten und Erweiterungen, die es ermöglicht haben sämtliche Funkionalität in Form von Plugins in die Entwicklungsumgebung einzubauen.

4.3. Eclipse

Eclipse ist eine weit verbreitete Entwicklungsumgebung für Java. Seit der Veröffentlichung im Jahr 2001 ist die Akzeptanz für diese IDE rasant gestiegen. Unter Linux Entwicklern ist vom Februar 2003 bis Februar 2004 die Nutzung von Eclipse als Haupt-IDE um 80% gestiegen. Im März 2005 ergab eine Umfrage unter 515 Teilnehmern des Java Symposiums, dass 53% Eclipse als ihre erste IDE bevorzugen. An zweiter Stelle kam Jetbrains' IntelliJ IDEA mit 19% der Stimmen [Got05, s. 108]. Mit Hilfe der Plugin-Architektur unterstützt Eclipse mittlerweile außer Java diverse weitere Programmiersprachen, u.a. C/C++, PHP oder Cobol. Die rasante Entwicklung und das schnelle Wachstum dieser IDE birgt aber auch Probleme. Nicht nur Stabilität und Performanz sind Problematisch, auch Überlappung oder Konflikte unter den einzelnen Eclipse-Projekten müssen gehandhabt werden. Je mehr Projekte und Menschen, die sich bei Ihrer Entwicklung auf Eclipse stützen, von Eclipse abhängig sind desto mehr entsteht der Druck Eclipse als Platform langsamer und weniger zu verändern [Got05, s. 110]. Beispielhaft kann hier das AJDT Projekt erwähnt werden, welches AspectJ Unterstützung in Eclipse einbauen soll. Die Entwicklung dieser Erweiterung für Eclipse richtete sich massgeblich nach den JDTs (s. Abschnitt 3.3). Ein Starke Änderung des JDT könnte zum Bruch in der Bedienbarkeit dieser Werkzeuge führen. Dies ist ein definitiv ungewollter Effekt. Das Beispiel von JDT und AJDT zeigt, dass starke Abhän-

gigkeiten untereinander entstehen können. Diese Abhängigkeiten sind in diesem Fall wegen der Nähe von AspectJ und Java nachvollziehbar, sie fordern aber zusätzlichen Koordinations- und Wartungsaufwand. Um diese Abhängigkeiten besser verstehen und einordnen zu können, soll im nächsten Teilabschnitt die Plugin-Architektur von Eclipse näher untersucht werden.

4.3.1. Die Plugin-Architektur von Eclipse

Der Kern von Eclipse besteht aus einer Ablaufplattform für Plugins, weitere Funktionalität von Eclipse ist mit Plugins für diese Platform realisiert. Bis einschließlich der Version 2.1 hatte Eclipse eine proprietäre Platform. Seit Version 3.0 ist sie kompatibel mit OSGi [Dau06, s. 364f]. Der Ablaufkern von Eclipse, Equinox, erfüllt dabei die Rolle eines OSGi-Servers. Eine zusätzliche Kompatibilitätsschicht sorgt dafür, dass auch ältere Plugins weiterhin lauffähig sind. OSGi und die ursprüngliche Architektur von Eclipse sind für ähnliche Aufgaben entwickelt worden. Gleichzeitig unterscheiden sie sich soweit voneinander, dass es unpraktisch ist, sie beide zu vereinen. In diesem Abschnitt soll dargestellt werden, wo die Stärken der Plugin-Architektur von Eclipse sind und wieso sie sich für die Entwicklung eines Plugins für eine Spracherweiterung eignet.

Eclipse Extensions

Eclipse verfügt über eine Registrierung, in der Extensions (Erweiterungen) zu Extension Points (Erweiterungspunkten) zugeordnet werden. Ein Erweiterungspunkt wird von einem Plugin deklariert und bedeutet, dass dieses mit neuer Funktionalität in bestimmter Weise erweitert werden kann. Eine Deklaration von Erweiterungspunkten wird in der `plugin.xml` (s. Abschnitt 4.3.2) festgehalten, und ist wie folgt aufgebaut:

Listing 4.1: Deklaration eines Erweiterungspunktes in einer plugin.xml

```
...
<extension-point name="Name des Erweiterungs Punktes"
        id="Identifikation (ID) des Erweiterungspunktes"
        schema="schemaDateiFürDenErweiterungsPunkt.exsd"/>
...
```

Eine Erweiterung ist entsprechend eines Schemas für den Erweiterungspunkt erstellt. Dieses Schema ist in einer eigenen Datei enthalten dessen Inhalt weiter unten erklärt wird. Ein Eintrag einer Erweiterung in der `plugin.xml` sieht folgendermaßen aus:

Listing 4.2: Deklaration einer Erweiterung in einer plugin.xml

```
...
<extension
        point="Identifikation (ID) des Erweiterungspunktes"
        id="Identifikation (ID) der Erweiterung"
        name="Name der Erweiterung">
```

```
        <schemaSpezifischerKnoten>
                ...
        </schemaSpezifischerKnoten>
</extension>
...
```

Die weiter oben bereits erwähnte Schema-Datei ist in einem *XML Schema*-Dialekt gehalten und wird standardmäßig im Unterordner `schema` eines Plugins gespeichert. Sie trägt die Dateiendung `.exsd` (was soviel wie *Eclipse XML Schema Definition* bedeutet). In dieser Datei werden Elemente definiert, die zusätzliche Attribute haben können. Ein Element wird mindestens durch einen Namen und kann mit weiteren Angaben genauer spezifiziert werden. Die Attribute können vier verschiedene Typen haben:

- `java`: Pfad einer Java Klasse
- `resource`: Pfad einer Ressource, z.B. eines Bildes
- `string`: beliebige Zeichenkette, z.B. Namen oder Anzahl
- `boolean`: Wahrheitswert; `true` oder `false`

In einem Schema können einzelne Elemente auf andere Elemente im selben Schema in Form einer `sequence` Bezug nehmen. Eine Beispiel für eine Eclipse XML Schema Datei findet sich im Anhang (s. Abschnitt A.2.1) wieder.

Beim Anmelden eines Plugins wird dieses von Eclipse auf Erweiterungspunkte und Erweiterungen durchsucht, welche in eine Registrierung eingetragen werden. Möchte zu einem späteren Zeitpunkt ein Plugin wissen, welche Erweiterungen für seine Erweiterungspunkte zur Verfügung stehen, kann dieses Plugin diese Informationen von dieser Registrierung beziehen. Bis zu diesem Punkt werden nur XML-Metadaten über die Erweiterung an das Plugin, welches den Erweiterungspunkt anbietet, geliefert. So kann ein Plugin für VJ zwar installiert sein, aber es muss keine Java Klasse geladen werden, sofern man kein VJ benutzen möchte. Bei sorgfältiger Programmierung können so viele Ressourcen geschont werden. Manche Erweiterungen müssen keinen Java Code enthalten.

OSGi Dienste

Im Vergleich zu den Extensions sind OSGi Dienste Java Objekte. Diese werden nicht automatisch registriert, sondern müssen ein Objekt von sich selbst erzeugen und dieses als Dienst mit Hilfe der OSGi API registrieren.

Listing 4.3: Registrierung eines OSGi Dienstes

```
public void start(BundleContext context) {
  // erzeuge Dienst Objekt
  ResourceChangeLookup lookup =
    new ResourceChangeLookup(context.getBundle());
```

```
    // registriere Dienst
    context.registerService(
      ResourceChangeLookup.class.getName(),
      lookup, null);
}
```

OSGi ist dynamisch, sodass Dienste jederzeit dazu kommen oder wegfallen können. Aus diesem Grund muss beim Zugriff auf einen Dienst jederzeit dieser Dienst neu geholt werden und sollte nicht zwischengespeichert werden. Bei größerer Anzahl von Diensten kann das sehr aufwendig werden. Aus diesem Grund sollte ein speziell dafür entwickelter `ServiceTracker` benutzt werden. Die Kontrolle über das Laden von Diensten ist sehr schwierig, da ein Dienst selbst entscheidet, wann er geladen wird. Abhängigkeiten unter Diensten machen es noch schwieriger zu entscheiden welche Dienste wann geladen werden sollen. Bei Anwendungen der Größe einer IDE wird dieses Vorhaben nahezu unmöglich.

	Erweiterungen	Dienste	Deklarative Dienste
Ressourcenschonendes Laden	*	-	*
Natives Hot Plug	-	*	*
Unterstützung in Eclipse[1]	*	(-)	(-)

Tabelle 4.2.: Vergleich von OSGi Diensten und Eclipse Erweiterungen

Deklarative Dienste

Die Eclipse Extensions und OSGi-Dienste haben jeweils ihr Vor- und Nachteile. Die Entwicklung von Deklarativen Diensten wurde maßgeblich von ihnen beeinflußt, und versucht die Vorteile beider zur vereinen. Auf der einen Seite soll dieser Ansatz die Dynamik der Dienste erhalten, auf der anderen sollen die Dienste erst bei Verwendung geladen werden. Um dies zu ermöglichen wurde dem *Bundle* die Zuständigkeit für das Anmelden der Dienste weggenommen. Der Begriff Bundle steht für OSGi-konforme Plugins. Dies übernimmt ein eigens dafür geschriebenes, gesondertes Bundle - *Service Component Runtime* (im folgenden SCR genannt). Für jeden Dienst besitzt ein Bundle eine eigene XML-Datei, aus der die SCR Informationen über den Dienst bekommt. Die einzelnen XML-Dateien müssen im Bundle-Manifest (s. Abschnitt 4.3.2) aufgelistet werden, damit diese vom SCR gefunden werden. Der SCR verfügt über genügend Informationen über die Dienste, um einen Proxy als Platzhalter zu registrieren. Falls die Implementierung gebraucht wird, wird das vom SCR erkannt und der Platzhalter mit dem wirklichen Dienst ersetzt.

Die Deklarativen Dienste vereinen die Vorteile von Eclipse Erweiterungen und OSGi Diensten. Leider ist die Unterstützung seitens der Eclipse IDE für die Deklarativen

[1] Die Unterstützung mit visuellen Tools ist im Vergleich mit Extensions sehr viel eingeschränkter

Dienste noch sehr gering. Erst mit Version 3.3 des Ablaufkerns Equinox werden alle Möglichkeiten der Deklarativen Dienste unterstützt. Damit ist zumindest die Lauffähigkeit gesichert. Doch die Einbindug in Eclipse ist sehr mager. So müssen alle Dienste, mit den dazu gehörigen XML-Dateien, manuell im Plugin-Manifest eingetragen werden. Die Tabelle 4.2 fasst die hier zusammen getragenen Punkte zusammen [Bar07].

Entsprechend dem Umbruch in der Plugin-Architektur von Eclipse, wandelt sich der Aufbau eines Plugins selbst. Im Abschnitt 4.3.2 soll von der Globalen Architektur-Sicht abstrahiert werden und ein Einblick in den inneren Aufbau eines Plugins gegeben werden.

4.3.2. Aufbau eines Eclipse Plugins

In diesem Abschnitt wird genauer auf den Aufbau eines Eclipse Plugins eingegangen. Ein wichtiger Bestandteil dieser sind die Extension Points, die bereits in Abschnitt 4.3.1 beschrieben wurden.

Der Grundaufbau eines Eclipse Plugins ähnelt dem einer Java Bibliothek. Es ist eine Sammlung von Binärcode enthaltenden Klassendateien, die in Verzeichnissen eingeordnet sind. Die Dateien sind meistens in einem Java-Archiv zusammen gehalten, können aber auch als reine Ordnerstruktur vorliegen. Zu den Java Klassen können noch zusätzliche Dateien wie Bilder hinzugefügt werden. Die Datei `plugin.xml`, die das Plugin-Manifest enthält, muss vorhanden sein. Darin wird die Konfiguration des Plugins, wie Identifikation, Name, benötigte Bibliotheken, vorhandene Erweiterungen und Erweiterungspunkte, festgehalten.

Listing 4.4: Ausschnitt einer nicht OSGi-konformen plugin.xml

```
<plugin
   id="Identifkation (ID) des Plugins"
   name="Name des Plugins"
   version="Version des Plugins"
   <requires>
      <import plugin="eine.plugin.id"/>
   </requires>
   <extension
         point="id.eines.erweiterungs.punktes">
         Inhalt der Erweiterung
                ...
   </extension>
>
```

Plugins, die OSGi konform sind, müssen weiterhin ein OSGi-Manifest, *META-INF Manifest.mf*, enthalten. In dieser Datei sind grundlegende Informationen über das Bundle enthalten. Von der Ähnlichkeit zu Java Bibliotheken stammt der Eintrag über die Manifest-Version (`Manifest-Version: 1.0`). Diese Version von Manifest-Dateien ist nachdem Schema *name: wert* aufgebaut [SM03]. Die `Manifest.mf`-Datei für Plugins

enthält im Gegensatz zu Java Bibliotheken mindestens noch folgende Plugin-spezifische Einträge:

- Bundle-ManifestVersion: OSGi-Manifest Version
- Bundle-Name: Name des Plugins
- Bundle-SymbolicName: Identifkation (ID) des Plugins
- Bundle-Version: Version des Plugins

Die `Manifest.mf` Datei überlappt inhaltlich fast vollständig mit der `plugin.xml`, deshalb wurden für OSGi-konforme Plugins alle Einträge aus der `plugin.xml` in die `Manifest.mf` verschoben. Die `plugin.xml` enthält nur noch Einträge über Erweiterungen und Erweiterungspunkte, die das Plugin mitbringt. Hat das Plugin keine Erweiterungen und keine Erweiterungspunkte, kann die `plugin.xml` bei OSGi konformen Plugins wegfallen [Dau06, s. 602f].

In Eclipse wird jedem Plugin ein eigener `ClassLoader` zugeordnet. Dieser hat einen eigenen Klassenpfad und kennt nur Klassen aus dem selben Plugin und aus den referenzierten bzw. benötigten Plugins oder Bibliotheken. Auf diese Weise wird jedes Plugin als eine eigenständige Anwendung innerhalb der IDE behandelt. So können auch mehrere Versionen des gleichen Plugins parallel laufen ohne sich gegenseitig zu beeinflussen. Weiterhin kann ein Plugin-Projekt nur auf andere Plugin-Projekte referenzieren. Projekte, die nicht als Plugin entwickelt werden, müssen als Bibliothek in den Klassenpfad eingebunden werden. Diese Einschränkung hat auch Nachteile, so kann ein nicht-Plugin Projekt, wie ein Übersetzer-Projekt, nicht parallel entwickelt werden, sondern muss jedesmal für Tests als eine Bibliothek geliefert werden. Wegen der Einschränkungen für den Klassenpfad eines Plugins muss jedes Plugin seinen Klassenpfad selbst verwalten, sofern Zugriff auf weitere Klassen gebraucht wird. Soll ein Plugin auf Klassen im Workspace einer Eclipse-Instanz zugreifen, so müssen diese Klassen, dem Klassenpfad des Plugins explizit hinzugefügt werden.

Wird ein Plugin von einem anderen benötigt, so ist der Zugriff auf alle Klassen und Typen offen. Die Java Paketstruktur bietet hier nicht genug Kontrollmöglichkeiten. Ein Plugin könnte für sich intern öffentliche (`public`) Zugriffe benötigen, die aber nicht von anderen Plugins genutzt werden sollten. Diese öffentlichen Elemente können in Entwicklung sein, und ihr Verhalten kann sich in zukünftigen Versionen ändern. Um diesem Problem vorzubeugen wird der Zugriff aus fremden Plugins auf alle Pakete, die als Teil ihres Names `internal` tragen, standardmäßig blockiert. Es ist allerdings möglich diese Einschränkungen zu umgehen, in dem von Eclipse heraus entsprechende Regeln (*Project Properties -> Java Build Path -> Libraries -> Plug-in Dependencies -> Access Rules*) für den Zugriff gesetzt werden.

4.4. Zusammenfassung

Die IDEs bieten Momentan keinen dedizierten Mechanismus zur Erweiterung der Tools um neue Sprachkonzepte. Die verschiedenen verwandten Ansätze beziehen sich auf die

Unterstützung einer vollständig neuen Sprache, ohne dass eine Integration mit einer bereits vorhandenen Sprache möglich ist. Eine Unterstützung für eine Spracherweiterung muss vollständing neu entwickelt werden. Die Plugin-Architektur von Eclipse ist sehr mächtig und bietet gute Möglichkeiten die bestehenden Tools zu erweitern. In Abschnitt 5 sollen diese verschiedenen Möglichkeiten untersucht und miteinander verglichen werden.

5. Möglichkeiten für JDT-Erweiterung

JDT wird standardmäßig für die Entwicklung von Java-Applikationen unter Eclipse verwendet. Die Unterstützung für VJ durch Eclipse soll an das JDT anknüpfen. So kann eine möglichst große Übereinstimmung in der Benutzung der IDE für VJ und Java gewährleistet werden. Zu grundlegenden Anforderungen an die Unterstützung gehören das Syntax Highlighting und ein integrierter Übersetzungsprozess. Die Übersetzung von VJ-Quelltexten und Java-Quelltexten soll gleichzeitig und transparent für den Entwickler stattfinden. Zur weiteren und aufwendigeren Unterstützung gehören Code Completion und Refactoring-Möglichkeiten.

Ein Plugin muss folgende Bereiche abdecken, damit VJ erfolgreich in die IDE eingebunden wird:

- Der VJ-Compiler wird beim Übersetzungsprozess integriert
- Die Benutzungsoberfläche soll angepasst werden (z.B. ein Editor für VJ-Quelltexte mit Syntax Highlighting)
- Die Kommunikation zwischen VJ-Compiler, Plugin und IDE muss sichergestellt werden, damit Befehle an den VJ-Compiler geschickt werden können und dieser Status-Meldungen zurückgeben kann.
- Anpassung von Entwicklungsprojekten an die VJ-Unterstützung

Für jeden dieser Bereiche gibt es verschiedene Ansätze, um die Unterstützung für VJ an die JDT anzubinden. Dabei soll der bereits existierende VJ-Compiler benutzt werden. Dieser Compiler kann in ein Plugin als Bibliothek eingebunden werden und von Eclipse aus beim Übersetzen aufgerufen werden. Im folgenden sollen mögliche Lösungen zu den Anforderungen näher betrachtet werden. Für ein besseres Verständnis wird der Begriff *Arbeitsprojekt* eingeführt. Dabei handelt es sich um ein Projekt im Workspace einer Entwicklungsumgebung, auf welches die Funktion des VJ-Plugins angewendet wird und in dem VJ bei der Entwicklung benutzt wird. Der Begriff leitet sich aus *Arbeitsbereich* (engl. Workspace) und *Projekt* ab.

5.1. Übersetzungsprozess

In diesem Abschnitt werden verschiedene Möglichkeiten erörtert, die eine Lösung für die Integration des VJ-Compilers in den Übersetzungsprozess bieten. Die Einbindung des VJ-Compilers in den Übersetzungsprozess von Eclipse umfasst mindestens zwei Punkte.

- Der VJ-Compiler muss die Möglichkeit haben auf die Quelltexte und die Binärtexte aus dem Arbeitsprojekt zuzugreifen
- Der VJ-Compilers muss zur Übersetzung vom VJ-Plugin aufgerufen werden

5.1.1. *ClassLoader*

In Eclipse besitzt jedes Plugin seinen eigenen `ClassLoader`. Dieser `ClassLoader` kennt nur die Pakete des Plugins und referenzierter Plugins, sowie die Pakete, die vom Eltern-`ClassLoader` zur Verfügung gestellt werden [Dau06, s. 365]. Dieser `ClassLoader` kennt die `packages` der Arbeitsprojekte, die im Workspace von Eclipse vorhanden sind, nicht. Beim Compilieren des Arbeitsprojektes bzw. von dessen Teilen mit Hilfe des VJ-Plugins führt das zu `ClassNotFoundException`s. Eclipse bietet zwei Möglichkeiten, *Fragmente* und *Buddies*, um den Klassenpfad eines Plugins zu erweitern.

Fragmente sind Teile eines Plugins, welche in separaten Archiv-Dateien vorliegen. Diese Aufteilung ist hilfreich für die Auslieferung des Plugins, falls nur ein Teil eines Plugins erneuert werden muss, der in ein Fragment ausgelagert wurde. Sowohl das Plugin als auch das Fragment muss selbst entwickelt werden, damit die Abhängigkeit zwischen beiden hergestellt werden kann. Der Klassenpfad des Plugins wird um den Klassenpfad des Fragments erweitert [Dau07, s. 285ff].

Buddies, die andere Möglichkeit den Klassenpfad zu erweitern, ist nicht darauf beschränkt, dass beide Plugins selbst entwickelt werden. Ein Plugin kann sich selbst als Buddy eines anderen deklarieren und so auf den Klassenpfad des anderen zugreifen. Um das Prinzip der Kapselung nicht vollständig zu umgehen, kann ein Plugin deklarieren, ob ein anderes Plugin als Buddy auf den eigenen Klassenpfad zugreifen kann und nach welchen Kriterien es geschehen soll.

Die von Eclipse direkt angebotenen Möglichkeiten den Klassenpfad zu erweitern sind in zweierlei Hinsicht nicht ausreichend. Sie geben nur Zugriff auf die Klassen eines Plugins nicht aber auf die des Arbeitsprojektes, welche zur Übersetzung benötigt werden. Der andere Aspekt beruht darauf, dass der VJ-Compiler als Bibliothek in das VJ-Plugin eingebunden ist und deshalb auf den Klassenpfad des VJ-Plugins nicht zugreifen kann. Mit Fragmenten und Buddies wird der Klassenpfad des Plug-ins aber nicht der Bibliotheken erweitert. Die Option, den VJ-Compiler als Teil des VJ-Plugins aufzunehmen, beeinträchtigt die Modularität und ist nicht gewollt. Als Ausweg kann der `ClassLoader` verändert beziehungsweise modifiziert werden. Dieser `ClassLoader` kann direkt als Parameter an den VJ-Compiler übergeben werden. Weiter sollte der `ClassLoader` über die Schnittstelle der abstrakten Klasse `java.lang.ClassLoader` benutzbar sein, um Abhängigkeiten des VJ-Compilers vom VJ-Plugin zu vermeiden.

Alternativ zur Parametrisierung kann für die Zeit der Übersetzung der ablaufende `ClassLoader`, der über `java.lang.Thread` erreichbar ist, mit einem geeigneten ersetzt werden und nach der Übersetzung wieder in den Ausgangszustand versetzt werden.

Nachdem sichergestellt ist, dass alle für die Übersetzung nötigen Klassen auf dem Klassenpfad des VJ-Compilers vorliegen, werden jetzt die Möglichkeiten vorgestellt, um den VJ-Compiler während eines Übersetzungsprozesses anzusprechen. Eclipse bietet drei Möglichkeiten, `ElementChangedListener`, `CompilerParticipant` und `Builder`,

um am Übersetzungsprozess teilzunehmen.

5.1.2. ElementChangedListener

`IElementChangedListener` ist eine Schnittstelle von Eclipse aus dem Paket `org.eclipse.jdt.core`. Bei diesem Ansatz werden Veränderungen, nachdem sie passiert sind, mit Hilfe eines `ElementChangedEvent` an alle angemeldeten Implementierungen dieser Schnittstelle übergeben. Die Schnittstelle `IElementChangedListener` muss implementiert werden. Eine vorhandene Implementation, die öffentlich zugänglich ist, ist nicht vorhanden. Der `ElementChangedListener` muss beim Laden des Plugins angemeldet werden. Bei der Anmeldung kann entschieden werden, ob der `ElementChangedListener` alle `ElementChangedEvents` bekommen soll oder nur die, die zu einem bestimmten Zeitpunkt stattfinden. Dies erlaubt unnötige Aufrufe des eigenen `ElementChangedListener` zu vermeiden. [EFb, Kapitel: Manipulating Java Code] Das `ElementChangedEvent` enthält Informationen über den Zeitpunkt des eigenen Auftretens, das Format ist `int`:

- POST_CHANGE: Dieses Event wird erstellt, nachdem eine Veränderung stattgefunden hat.
- PRE_AUTO_BUILD: Dieses Event wird erstellt, sofern vor einem AUTO_BUILD Veränderungen durchgeführt werden.
- POST_RECONCILE: Dieses Event wird erstellt, wenn eine zwischengespeicherte Arbeitskopie mit dem Original abgeglichen wird.

Weiterhin enthält es Informationen über die Veränderungen der Elemente, also Source- und Binary-Dateien sowie Verzeichnisse:

- ADDED: Element wurde neu hinzugefügt
- REMOVED: Element wurde entfernt
- CHANGED: Element wurde verändert

Die Methode `public IJavaElementDelta getDelta()` gibt wieder zu welchem Element dieses `ElementChangedEvent` gehört. Anhand des `IJavaElementDelta` kann festgestellt werden, ob der VJ-Compiler aufgerufen werden soll und welche Dateien ihm zur Übersetzung übergeben werden sollen.

5.1.3. CompilationParticipant

Eine weitere Möglichkeit den VJ-Compiler für die Übersetzung aufzurufen ist den Erweiterungspunkt `CompilationParticipant` aus dem Plugin `org.eclipse.jdt.core` zu benutzen. Es muss eine Implementierung der abstrakten Klasse `org.eclipse.jdt.core.compiler.CompilationParticipant` bereitgestellt werden. Durch das Konzept der Eclipse Erweiterungspunkte (s. Abschnitt 4.3.1) wird diese Klasse erst dann

geladen und aufgerufen, wenn ein Übersetzungsprozess angestoßen wird. Folgende für unseren Ansatz in Frage kommenden Methoden stehen dem `CompilationParticipant` zur Verfügung und können auch überladen werden:

- `public int aboutToBuild(IJavaProject project)`: Wird direkt vor dem Übersetzungsprozess aufgerufen und gibt die Möglichkeit vorbereitende Aufgaben, wie z.B. benötigte Verzeichnisse für die generierten Quelltexte zu generieren, auszuführen.
- `public void buildStarting(BuildContext[] files, boolean isBatch)`: Wird beim Start des Übersetzungsprozesses ausgeführt, als Parameter werden die zu übersetzenden Dateien übergeben. Der Parameter `isBatch` deutet daraufhin, dass `files` alle Dateien des Arbeitsprojektes enthält.
- `public void cleanStarting(IJavaProject project)`: Wird aufgerufen, wenn vor einem Übersetzungsprozess ein Säuberungsvorgang durchgeführt wird. Es gibt die Möglichkeit, früher generierte Dateien zu entfernen.
- `public boolean isActive(IJavaProject project)`: Mit diesem Aufruf kann ein `CompilationParticipant` entscheiden, ob er bei dem Übersetzungsvorgang des Projektes `project` teilnehmen will.

Dieser Ansatz gibt die Möglichkeit Vorbereitungen für die Generierung von eigenen Java Quelltexten durchzuführen. Denkbar ist die Erstellung gesonderter Verzeichnisse für die generierten Dateien. Die Methode `buildStarting()` wird nur dann aufgerufen, wenn Java Quelltext Dateien übersetzt werden sollen. VJ Quelltext Dateien sind für Eclipse zuerst keine Java Quelltext Dateien und werden nicht bekanntgegeben. Mit Hilfe des Erweiterungspunktes `org.eclipse.core.runtime.contentTypes` kann deklariert werden, dass VJ Quelltexte auf Java Quelltexten basieren. Durch diese Deklaration wird der VJ Quelltext von allen Eclipse komponenten als Java Quelltext angesehen. In Folge dessen versucht der Eclipse-interne Java Übersetzer auch die VJ Quelltext Dateien zu übersetzen, was zu Fehlern führt.

5.1.4. Builder

Ein weiterer Ansatz ist den Erweiterungspunkt `org.eclipse.core.resources.builders` zu erweitern. Dieser Erweiterungspunkt soll von inkrementellen Übersetzern erweitert werden. Sie sollen nur die veränderten Quelltexte des Projektes übersetzen. Daraus generierte Dateien können speziell markiert werden. Ein Builder wird jeweils für jedes Arbeitsprojekt einzeln konfiguriert und soll damit schnell und gut skalierbar sein. Ein Builder kann an eine Projekt-Natur (s. Abschnitt 5.4.1) gebunden werden. Er wird in diesem Fall nur dann für einen Übersetzungsvorgang aufgerufen, wenn einem Arbeitsprojekt die entsprechende Projekt-Natur zugewiesen wurde. Einem Projekt können mehrere Builder zugewiesen sein. Für Java Projekte ist der `JavaBuilder` standardmäßig eingestellt. Die einzelnen *Builder* können in ihrer Reihenfolge sortiert werden. So können von einem *Builder* generierte Daten vom nächsten *Builder* weiter übersetzt werden. Ein *Builder* muss die abstrakte Klasse `IncrementalProjectBuilder`

implementiert werden. Die Implementierung wird mit vier verschiedenen Build-Typen aufgerufen werden:

- FULL_BUILD: Alle vorherigen Übersetzungszustände werden verworfen und alle Quelltexte werden neu übersetzt.
- AUTO_BUILD: Bei eingeschalteter Funktion *Autobuild* wird bei jeder Veränderung von Quelltexten dieser Build-Typ gestartet. Unterscheidet sich von INCREMENTAL_BUILD nur durch den automatischen Aufruf.
- INCREMENTAL_BUILD: Anhand eines mitgelieferten `IResourceDelta` wird die Veränderung zur letzten Übersetzung berechnet und alle davon abhängigen Quelltexte neu übersetzt.
- CLEAN_BUILD: Alle vorherigen Übersetzungszustände und generierte Dateien werden verworfen und alle Quelltexte werden neu übersetzt.

Das beim Aufruf eines *Builder*s mitgelieferte `IResourceDelta` enthält im Gegensatz zum `ElementChangedListener` Informationen über Veränderungen sämtlicher Verzeichnisse und Dateitypen des Arbeitsprojektes. Auch Veränderungen an Konfigurationsdateien und Verzeichnissen, die nicht als Quelltext-Verzeichnisse markiert sind, werden mitgeteilt. In der Eclipse Version 3.3, die in dieser Arbeit betrachtet wird, ist dieser Erweiterungspunkt, im Gegensatz zu den anderen Lösungen für die Einbindung des VJ-Compilers in den Übersetzungsprozess, die einzige Lösung mit vollständig funktionsfähigem, vorgeneriertem Beispiel für einen XML-Builder.

Die Tabelle 5.1 fasst die drei verschiedenen Ansätze für den Übersetzungsprozess zusammen. Die ersten beiden Punkte sind dabei am wichtigsten. Die einfache Unterstützung anderer Formate als Java Quelltext und die Generierung einer funktionsfähigen Implementierung beschleunigen die Entwicklung eines Plugins. Der Builder-Ansatz bietet den größten Umfang an Funktionalität und gleichzeitig ein hohe Möglichkeiten

	ElementChangedListener	CompilationParticipant	Builder
Nicht nur für Java	-	-	+
Skelett vorgeneriert	-	-	+
Projektbezogen	-	-	+
Erweiterungspunkt[1]	-	+	+

Tabelle 5.1.: Ansätze für den Übersetzungsprozess

5.2. Benutzungsoberfläche

Die Benutzungsoberfläche (engl. User Interface) von Eclipse kann auch über Erweiterungspunkte um weitere Eigenschaften und Möglichkeiten erweitert werden.

[1] Wird erst beim Gebrauch geladen

5.2.1. Editor

Eine entscheidende Komponente ist der Editor für VJ. Dieser soll automatisch für die Bearbeitung von VJ-Quelltexten von Eclipse ausgewählt werden. Er soll eine visuelle Unterscheidung zum Java Editor besitzen, sodass die Verwechslung des Programmierkontextes minimiert wird. Der vom JDT mitgelieferte Java Editor besitzt keine Erweiterungspunkte mit Hilfe derer der Java Editor VJ korrekt unterstützen würde. Es muss ein eigener Editor bereitgestellt werden. Im folgenden werden drei Ansätze vorgestellt. Zuerst wird ein von Eclipse vorgeschlagener und regelkonformer Weg einen Editor einzubinden vorgestellt. Im nachhinein wird eine Lösung vorgestellt, die Analog zum AspectJ-Plugin (Version: 1.5) bewusst einige Regeln bricht. Zuletzt wird die Lösung vorgestellt, bei der VJ-Quelltext als Java Quelltext deklariert wird. Die ersten zwei Ansätze bauen auf dem Erweiterungspunkt `org.eclipse.ui.editors` auf. Bei der Konfiguration dieses Erweiterungspunktes können außer den vorausgesetzten (s. 4.3.2) `ID` und `Name` optional folgende Angaben gemacht werden:

- Icon - Graphik (16x16 Pixel), für visuelle Unterscheidung der Editoren
- Extensions - Dateitypen, in Form einer Kommagetrennten Liste von Datei-erweiterungen, die der Editor unterstützt.
- Class - Java Klasse, die die Schnttstelle `IEdtiorPart` implementiert und das Grundgerüst des Editors darstellt.

Weitere Optionen werden hier nicht explizit erwähnt, da diese für das VJ-Plugin nicht unbedingt notwendig sind. Eine detailiertere Beschreibung ist im Abschnitt A.2.2 zu finden.

Regelkonforme Implementierung

Zuerst wird der erste und regelkonforme Ansatz betrachtet. Hierfür bietet Eclipse beispielhaft die Generierung eines Editorrumpfes.

Zum Zeitpunkt der Erstellung dieser Arbeit wird ein vollständiger XML-Editor, analog zum Builder (s. 5.1.4), generiert. In Abbildung 5.1 ist zu sehen, wo der XML-Editor in der Hierarchie von Editoren in Eclipse angesiedelt ist. Dieser Rumpf muss an VJ angepasst werden. Bei Wahl dieser Lösung fehlt viel Funktionalität, die noch grundlegender als z.B. *Code Completion* ist. Die folgende Aufzählung ist nur ein kleiner Ausschnitt:

- Beim Erstellen eines Blocks mittels einer geschweiften Klammer { wird keine schließende Klammer hinzugefügt
- Der Umbruch landet immer in der ersten Spalte und ist nicht eingerückt
- Highlighting muss für Java angepasst werden

Soll von einer anderen Klasse in der Editor-Hierarchie geerbt werden, dann können nur die *nicht internen* Klassen benutzt werden. Die Plugin Architektur erlaubt

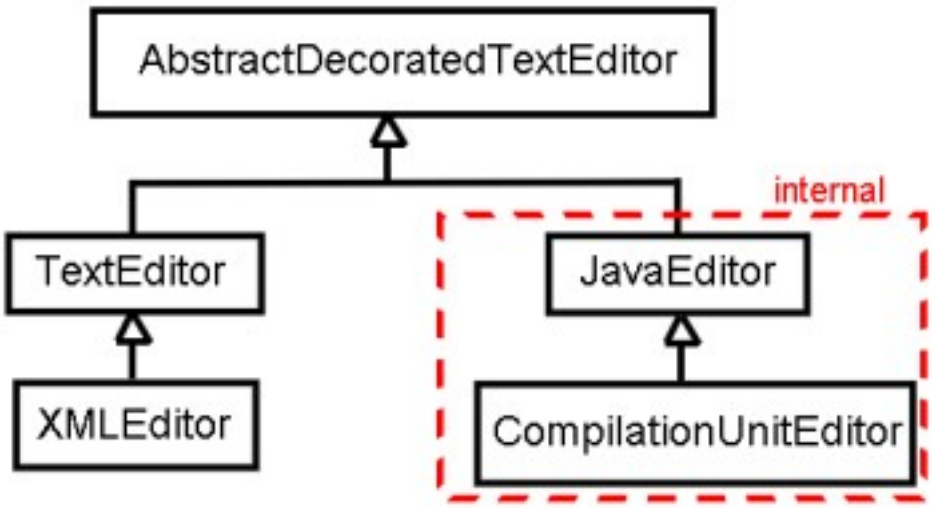

Abbildung 5.1.: Vererbungshierarchie von Editoren

nicht ohne weiteres den Zugriff auf die internen Java Editoren. Der XML-Editor unterscheidet sich allerdings kaum vom *TextEditor*. Soll noch weiter oben beim *AbstractDecoratedTextEditor* angesetzt werden, wird noch mehr Funktionalität wegfallen, die dann ggf. wieder zusätzlich implementiert werden muss.

Erben vom internen Java Editor

Wegen der Ähnlichkeit von VJ zu Java soll sich auch der Editor für VJ am Editor für Java orientieren. Der Unterschied zum vorherigen Ansatz ist der, dass von einer Editor-Klasse Gebrauch gemacht wird, die in einem *internen* Paket liegt und somit, der Zugriff darauf von Eclipse aus unterbunden wird. Der Editor für Java, an den angeknüpft werden soll, heißt `CompilationUnitEditor` und erbt von der abstrakten Klasse `JavaEditor`. Da diese beiden Klassen im Package `org.eclipse.jdt.internal.ui.javaeditor.JavaEditor` liegen ist der Zugriff auf diese unterbunden. Der Editor bietet entsprechend auch keine Erweiterungspunkte, an die angeknüpft werden kann. So könnte ein Erweiterungspunkt für Syntax Highlighting vorhanden sein. Weil auch diese Möglichkeit nicht besteht muss die API umgangen werden und der eigene Editor von den *internen* Editoren erben. Renomierte Projekte, wie das AJDT Projekt, welches AspectJ (Plugin-Version: 1.5.0.200706070619) in Eclipse einbindet, erbt beispielsweise direkt vom `CompilationUnitEditor`. Bei diesem Ansatz wird ein Editor zur Verfügung gestellt, an dem mit wenig Anpassung, der Umgang mit VJ-Quelltexten analog zum Umgang mit Java-Quelltexten stattfindet. Dadurch wird die größtmögliche Konformität mit dem Aussehen und dem Verhalten des Java Editors erreicht. Hierbei ist *Code Completion* nicht mit einbegriffen, da dies nicht alleine vom Editor berechnet wird.

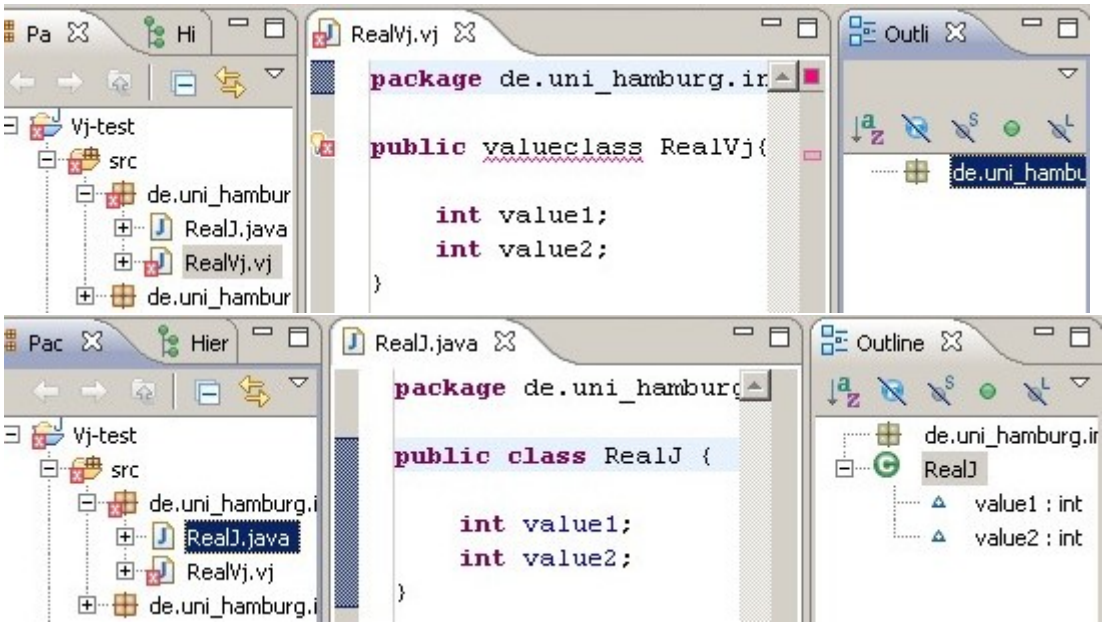

Abbildung 5.2.: (a) VJ-Quelltext im Java Editor, diverse Werkzeuge können damit nicht umgehen. (b) Zum Vergleich die gleiche Sicht mit Java-Quelltext

VJ als Java Quelltext

Dieser Ansatz unterscheidet sich grundsätzlich von den zwei vorherigen. Hier wird der Erweiterungspunkt `org.eclipse.core.runtime.contentTypes` benutzt. Mit Hilfe dieses Erweiterungspunktes kann VJ-Quelltext als Java Quelltext deklariert werden. VJ-Quelltext wird von Werkzeugen, die für Java gebaut sind, bearbeitet.

So werden VJ-Quelltextdateien im Java Editor geöffnet und von dem Eclipse eigenem Java Übersetzer übersetzt. Wegen der Differenzen zwischen VJ und Java ist das Highlighting für VJ-Syntax nicht vollständig richtig, so wird `valueclass` nicht analog zu `class` fett eingeblendet. Weiterhin wird VJ-Syntax als fehlerhaft markiert, was die Lesbarkeit des Quelltextes erschwert. Der größte Nachteil entsteht durch das Schlüßelwort `valueclass`. Die JDTs sind nicht in der Lage dieses zu interpretieren und können keinen *Abstract Syntax Tree* (AST - in Deutsch Abstrakter Syntax Baum) aus diesen Klassen aufbauen. Werkzeuge wie *Outline* benötigen diesen AST allerdings um korrekte Daten anzeigen zu können. In Abbildung 5.2 werden die hier erwähnten Probleme mit der Darstellung von VJ im Vergleich zur Darstellung von Java aufgezeigt.

Das Erben vom internen Java Editor bietet die größtmögliche Anpassung und Integration des VJ-Editors in die IDE. Die Funktion, Benutzung und Aussehen des Editors sind größtenteils identisch mit dem Java Editor. Aus diesem Grund ist dieser Ansatz für die Integration von VJ zusammen mit den JDT am besten geeignet.

5.3. Kommunikation zwischen Bibliothek, Plugin und IDE

In diesem Abschnitt werden Möglichkeiten für die Kommunikation zwischen der Bibliothek, in der der VJ-Compiler mitgeliefert wird, und dem Plugin bzw. Eclipse vorgestellt. Die Kommunikation findet in beide Richtungen statt. Ausgaben vom VJ-Compiler müssen vom VJ-Plugin entgegengenommen und interpretiert werden. Das VJ-Plugin muss den VJ-Compiler aufrufen, sodass dieser korrekt parametrisiert ausgeführt wird. Weiterhin müssen Fehlermeldungen, die vom VJ-Compiler generiert und vom VJ-Plugin entgegengenommen werden, weiter an die IDE übergeben werden, sodass die IDE diese darstellen kann.

5.3.1. Aufruf des VJ-Compilers

Der VJ-Compiler ist ursprünglich für die Kommandozeile entwickelt worden. Der Aufruf mit Hilfe der statischen `main`-Methode schränkt die Benutzbarkeit ein und lässt wenig Kontrolle über die Übersetzung zu. Zudem kann die `main`-Methode nur einen `String`-Parameter entgegennehmen, der den relativen Pfad der VJ-Quelltext Datei enthält. Über Parameter kann nicht bestimmt werden, wohin der generierte Java-Quelltext hingeschrieben wird. Quelltext-Dateien, generierte Dateien und Binär-Dateien (aus Java-Quelltext erzeugte `class`-Dateien) werden in der Regel in unterschiedlichen Verzeichnissen innerhalb eines Arbeitsprojektes gehalten. Diese Option sollte gegeben sein, um ein nachträgliches Verschieben der Dateien zu vermeiden und somit den Übersetzungsprozess unnötig zu verlängern. Die Schnittstelle soll eine Möglichkeit dafür bieten.

Auch zukünftige Erweiterungen des VJ-Compilers sollen dabei ermöglicht werden. So kann der VJ-Compiler in Zukunft die *Java Compiler API* benutzen um aus VJ-Quelltext Auch Java Binärdaten zu generieren.

Die einfachste Methode ist es statisch die Parameteranzahl von einem auf drei zu erhöhen. Die zwei zusätzlichen Parameter sind, analog zum ersten Parameter, Pfade zum generierten Java-Quelltext und zu generierten Java Binärdaten, die als `String` übergeben werden. Die Benuztbarkeit dieser neuen Schnittstelle ist einfach, da nur zwei weitere Parameter angegeben werden. Für zukünftige Erweiterungen ist dieser Ansatz nicht geeignet. Die Schnittstelle müsste ggf. wieder verändert werden. Ein weiterer Nachteil ist die steigende Parameteranzahl, die zu Unübersichtlichkeit [Blo01, item 25] führt.

Ein weiterer Ansatz, den auch die Entwickler des Java Übersetzers genommen haben, ist als Parameter ein *String-Array* zu übergeben. Bei Erweiterungen kann die Schnittstelle unverändert bleiben, denn es werden nur weitere Felder im *String-Array* hinzugefügt. Diese werden dann durch die verborgene Implementation entsprechend verarbeitet. Die Benutzbarkeit ist durch die `String`-Parameter aufwendig. Es müssen Konvertierungen zum und vom `String`-Format durchgeführt werden.

Einen ähnlichen Weg geht man, wenn anstatt eines *String-Arrays* ein prioprietäres Objekt, z.B. `VJCompilationTask`, für die Übersetzung benutzt wird. Die Angaben von

der Kommandozeile müssen in dieses Objekt eingefügt werden und können nicht sofort benutzt werden. Dafür sind die Aufrufe von dem VJ-Plugin aus sehr viel einfacher auszuführen, denn der Zugriff auf die Parameter geschieht über *getter*-Methoden und nicht mit Hilfe der Analyse des *String-Arrays*.

Schlägt die Übersetzung einer VJ-Quelltextdatei fehl, weil eine benötigte Klasse nicht auf dem Klassenpfad vorhanden ist, kann der `VJCompilationTask` auf einen Stapel zurückgelegt werden. Dabei werden auf dem Klassenpfad nur Java-Binärdateien berücksichtigt. Das VJ-Plugin sucht dem Fehler entsprechend nach einer VJ-Quelltextdatei und erstellt einen neuen `VJCompilationTask` für die gefundene VJ-Quelltextdatei. Dieser wird versucht von VJ-Compiler abgearbeitet zu werden. Das Ergebniss muss noch vom *JavaBuilder* in Java-Binärdaten übersetzt werden. Danach kann der ursprüngliche `VJCompilationTask` abgearbeitet werden. Bei den zwei vorherigen Ansätzen müssen die Informationen, die für die spätere Übersetzung benötigt werden, neu geholt werden bzw. gesondert gespeichert werden. Für Erweiterungen müßte die Schnittstelle des `VJCompilationTask` angepasst werden, was das Problem nur verschiebt.

Tabelle 5.2 fasst die einzelnen Möglichkeiten und ihre Vor- und Nachteile zusammen.

	statisch	String-Array	VJCompilationTask
Schnelle Umsetzung	+	-	-
Zukünftige Erweiterungen	-	+	-
Handhabbarkeit	+	-	+
Wenige Parameter	-	+	+
Stapelbar	-	-	+

Tabelle 5.2.: Übergabe der Parameter an den VJ-Compiler

Der `VJCompilationTask` hat Vorteile für die interne Speicherung des Übersetzungszustands für das VJ-Plugin. Damit sollte der `VJCompilationTask` im VJ-Plugin vorhanden sein. Der Aufruf des VJ-Compilers kann am besten mit dem `String-Array` durchgeführt werden. Die Konvertierung vom `VJCompilationTask` zum `String-Array` kann im VJ-Plugin stattfinden. Der VJ-Compiler muss nicht abhängig vom `VJCompilationTask` und seiner Schnittstelle sein, gleichzeitig können die Vorteile des `VJCompilationTask` genutzt werden.

5.3.2. Konsole

Die Ausgaben des VJ-Compilers werden ursprünglich auf die aufrufende Kommandozeile bzw. Konsole ausgegeben. Meldungen über den Status der Übersetzung, die gezielt vom VJ-Compiler erzeugt werden, werden über `System.out` bzw. `System.err` ausgegeben. Beim Aufruf des VJ-Compilers aus dem VJ-Plugin heraus werden die Meldungen für den Entwickler nicht mehr sichtbar. Die Meldungen werden vom aufrufenden Prozess abgefangen und nicht an eine Konsole weiter geleitet. Eclipse bietet die Möglichkeit eine eigene *Console View* zu implementieren und am `ConsoleManager` anzumelden.

Dies war bis Eclipse 3.0 nötig gewesen, um aus einem Plugin heraus eine Ausgabe auf der Konsole zu erzeugen. Ab der Version 3.0 bietet Eclipse hier die Möglichkeit auf eine generische Konsole zu zu greifen, die nur mit einem Namen parametrisiert werden muss. Dies ermöglicht eine von anderen Meldungen gesonderte Ausgabe der Meldungen vom VJ-Compiler. Es gibt zwei Möglichkeiten die Ausgaben des VJ-Compiler auf dieser Konsole auszugeben. Beim ersten Ansatz werden zwei Ströme, einer für Fortschrittsmeldungen und einer für Fehlermeldungen des Übersetzungsprozesses, als Parameter an den VJ-Compiler übergeben. Diese Ströme müssen an sämtliche aufgerufene Klassen weitergegeben werden und diese Klassen daran angepasst werden, dass sie über diese Ströme ihre Medlungen ausgeben. Klassen die nicht direkt zum VJ-Compiler gehören, wie die die mit Java ausgeliefert werden, können nicht angepasst und neu übersetzt werden. Meldungen die innerhalb des VJ-Compilers ihren Ursprung in diesen Klassen haben, können ggf. nicht oder nur teilweise ausgegeben werden. Der VJ-Compiler muss zusätzlich die `org.eclipse.ui.console.MessageConsoleStream` kennen und wird dadurch an Eclipse gebunden. Ein anderer Ansatz ist die zwei Ströme `System.out` und `System.err` neu zu belegen. Für die Zeit des Aufrufs des VJ-Compilers können diese beiden auf Ströme verweisen, die an die gesonderte Konsole für den VJ-Compiler gekoppelt sind. Es werden sowohl gezielt erzeugte Ausgaben als auch von Ausnahmen (engl. *Exceptions*) erzeugte Ausgaben auf die sichtbare Konsole umgeleitet. Ausgaben der Klassen von Drittanbietern, die der VJ-Compiler benutzt, werden auch umgeleitet und ausgegeben, sofern diese Ausgaben ursprünglich auf einer Kommandozeile ausgegeben worden wären. Die Belegung der beiden Ströme `System.out` und `System.err` ist für den VJ-Compiler dabei transparent, unabhängig davon, ob dieser von der Kommandozeile oder vom VJ-Plugin aus aufgerufen wird.

5.3.3. Fehlererkennung

Die textuelle Ausgabe der Meldungen ist eine wichtige Unterstützung für den Benutzer bei der Anwendung des VJ-Plugins. Diese Information kann teilweise auch interpretiert werden und den Arbeitsprozess für den Benutzer erleichtern. So können Fehler innerhalb einer Quelltext-Datei graphisch mit Markierungen dargestellt werden, fehlende `import`s können vorher übersetzt werden. Die textuelle Ausgabe kann ohne weiteres aber nur vom Benutzer selbst interpretiert werden. Für das VJ-Plugin sind es nur zwei Ströme, die auf einer Konsole ausgegeben werden. Es werden zwei Ansätze vorgestellt, wie die Informationen über Fehler gewonnen werden können.

Beim ersten Ansatz werden die Ströme mit den Meldungen lexikalisch analysiert. Dies gibt die Möglichkeit, sowohl Fortschrittsmeldungen als auch Fehlermeldungen des VJ-Compilers zu interpretieren. Der Aufwand für den Aufbau eines Lexers ist allerdings sehr groß. Der Lexer müsste wiederum Objekte erzeugen, die die Fehler abbilden. Dieser zusätzliche Wandlungsschritt ist unnötig.

Beim zweiten Ansatz nutzt man die bereits innerhalb des VJ-Compilers erzeugten Ausnahmen. Diese können Informationen über Art und Ort eines Fehlers tragen. Anhand dieser Information kann eine Markierung an entsprechender Stelle erzeugt werden bzw. dem Fehler entsprechende Massnahmen ergriffen werden. Nur beim Erzeugen einer Ausnahme kann Information vom VJ-Plugin empfangen werden. Auf Fortschritts-

meldungen, die i.d.R. ohne Ausnahmen erzeugt werden, kann so nicht reagiert werden. Der Aufwand für die Umsetzung ist sehr gering, da die vorhandenen Ausnahmen nur geringfügig angepasst werden müssen. Anhand des Typs einer Ausnahme kann der Fehler identifziert werden, die mitgelieferte Information über die Stelle des Fehlers, z.B. Zeilen und Spalten Angabe innerhalb einer Quelltext-Datei, kann leicht gewonnen werden.

5.3.4. Markierung der Fehler im Quelltext

Treten Fehler während der Übersetzung auf, so kann das VJ-Plugin die dazu gehörigen Fehlermeldungen interpretieren, wie im vorherigen Abschnitt beschrieben wurde. Aus den gewonnenen Informationen können hinweisende Markierungen für eine Datei erzeugt werden. Auf Dateien und Verzeichnisse eines Arbeitsprojektes wird über Schnittstelle `org.eclipse.core.resources.IResource` zugegriffen. Diese Schnittstelle bietet auch die Möglichkeit eine Markierung für die jeweilige *Ressource* mit den Daten der Meldung zu erstellen. Diese Markierungen werden ohne weiteres hinzu tun graphisch im Editor angezeigt.

Die Bereits vorhandenen Ausnahmen, die im VJ-Compiler erzeugt werden eignen sich sehr gut für die Fehlererkennung. Auch der Aufwand für die Umsetzung ist nur sehr gering, sodass dieser Ansatz zu bevorzugen ist.

5.4. Anforderungen an VJ-Projekte

Ein Projekt, in welchem mit Hilfe von VJ entwickelt werden soll, muss dafür konfiguriert werden. Es muss sichergestellt werden, dass dieses Projekt bei der Übersetzung von VJ-Quelltext den VJ-Compiler aufrufen kann und dass dieser Aufruf vor dem Aufruf des `JavaBuilder`s stattfindet.

Weiterhin werden Dateien, die nicht als Java Quelltext-Dateien klassifiziert werden, automatisch vom Quelltext-Ordner in die Zielordner mit Binärdateien mitkopiert. Dies ist der Fall mit VJ-Quelltext Dateien, die nicht als Java Quelltext-Dateien deklariert wurden. Dieses Verhalten ist unerwünscht und kann durch einen Eintrag in Konfigurationsdateien für das Arbeitsprojekt unterbunden werden.

VJ Klassen erben immer von der Klasse `vj.lang.Value`. Diese muss in jedem Arbeitsprojekt auf dem Klassenpfad vorhanden sein, damit die Übersetzung vollständig abgeschlossen werden kann.

Diese Konfigurationen können teilweise vom VJ-Plugin übernommen werden, sodass der Benutzer sie nicht manuell einstellen muss. Die Einstellungen müssen nur für Arbeitsprojekte vorgenommen werden, in denen VJ benutzt werden soll. Andere Projekte müssen unverändert bleiben. Damit dürfen diese Veränderungen nicht beim Laden des Plugins vorgenommen werden, sondern erst bei einer Zuweisung zu einem Arbeitsprojekt. Speziell dafür bietet Eclipse den Erweiterungspunkt `org.eclipse.core.resources.natures`.

5.4.1. Projektnaturen

Projektnaturen sind Erweiterungen zu dem Erweiterungspunkt `org.eclipse.core.resources.natures`. Eine Projektnatur ist eine Zuweisung bestimmter Eigenschaften zu einem Arbeitsprojekt. Einem Projekt mit Java Natur werden neben anderen Eigenschaften z.B. der `JavaBuilder` oder ein Piktogram für die Kennzeichnung eines Java Arbeitsprojektes zugewiesen. Die Zuweisung einer Projektnatur wird in die Konfigurationsdateien eines Arbeitsprojektes geschrieben und gilt damit nur für dieses. Bei der Zuweisung der Projektnatur wird der Zugriff auf die meisten Eintstellungen eines Arbeitsprojektes ermöglicht, sodass diese verändert werden können. Falls ein *Builder* für den Aufruf des VJ-Compilers zuständig sein soll, kann an dieser Stelle der entsprechende `VJBuilder` vor den `JavaBuilder` platziert werden. Damit die VJ Quelltext-Dateien nicht in den Ausgabeordner kopiert werden kann der Filter angepasst werden.

Für VJ Arbeitsprojekte ist der Zugriff auf die Klasse `vj.lang.Value` nötig. Bei der Initialisierung der Projektnatur kann auch der Klassenpfad daraufhin angepasst werden, dass der Zugriff auf diese Klasse vorhanden ist.

5.5. Zusammenfassung

Eclipse bietet viele Ansatzpunkte für eine Erweiterung. Ein Plugin für eine Spracherweiterung muss in den Übersetzungsprozess von Eclipse eingebunden werden. Dies wird am besten über den Builder-Erweiterungspunkt erreicht. Der Editor für VJ sollte durch Vererbung vom Java Editor an jenen angepasst werden, um eine möglichst hohe Übereinstimmung zu erreichen. Die Schnittstelle zwischen VJ-Plugin und VJ-Compiler sollte für zukünftige Veränderungen felxibel aber auch stabil bleiben. Der Aufruf des VJ-Compilers sollte deshalb über ein anpassbares String-Array und die Fehlermeldungen über Ausnahmen realisiert werden. Projektnaturen bieten schliesslich den Vorteil einer automatisierten Konfiguration von Arbeitsprojekten für den Einsatz des VJ-Plugins. In Abschnitt 6 wird die Umsetzung des VJ-Plugins dargestellt. Für die Umsetzung werden die in diesem Abschnitt ausgewählten Ansätze benutzt.

6. Umsetzung

In diesem Kapitel wird die Umsetzung des VJ-Plugins für Eclipse vorgestellt, die aus den vorhandenen Teillösungen die beste Gesamtlösung bietet. Auch die Änderungen am VJ-Compiler, die nötig sind, um ihn in das Plugin einzubinden, werden dabei hier dargestellt. Die Veränderungen werden möglichst klein ausfallen, sodass die Unabhängigkeit des VJ-Compilers gewahrt bleibt. Dieser Lösungsweg führt aber zu geringen Änderungen an der Benutzung von VJ. In Abschnitt 6.3 werden die Auswirkungen der Einbettung auf die Sprache VJ selbst betrachtet.

6.1. Aufbau des Plugins

In diesem Abschnitt wird der Aufbau des Plugins vorgestellt. Grundsätzlich lässt sich die Umsetzung in vier große Gebiete unterteilen

- Der Übersetzungsprozess und die dafür notwendige Kommunikation zwischen Plugin und VJ-Compiler-Bibliothek
- Die Benutzungsoberfläche
- Fehlererkennung und Fehlerbehandlung
- Konfiguration von Projekten, bei denen das VJPlugin benutzt wird.

6.1.1. Übersetzungsprozess und Kommunikation

Für den Übersetzungsprozess wird der *Builder* benutzt. Eine vollständige Implementation eines XML-*Builder*s wird von Eclipse generiert. Gleichzeitig gibt Eclipse die Option eine entsprechende Projektnatur zu generieren und beide mit einander zu verknüpfen. Wenige Schritte sind nötig, um zum ersten mal den VJ-Compiler anzusprechen. Ausser der Umbenennung interner Methoden und Attribute von *XMLNamen* zu *VJNamen*, muss nur der Aufruf des generierten `XMLParser`s durch den Aufruf des VJ-Compilers ersetzt werden. Vorher muss noch aus dem übergebenen Parameter `IResource` der Pfad der zu übersetzenden VJ-Quelltextdatei gewonnen und daraus der Pfad der resultierenden Java-Quelltextdatei abgeleitet werden.

Listing 6.1: Simple Übersetzung von VJ-Quelltextdateien - verkürzte Fassung

```
public class VJBuilder extends IncrementalProjectBuilder {
...
  void buildVJ(IResource resource) {
```

```
    // Nur VJ-Dateien sollen übersetzt werden
    if (resource instanceof IFile
       && resource.getName().endsWith(".vj")) {
      IFile file = (IFile) resource;

      // Pfad der VJ-Quelltextdatei und
      // der generierten Java-Quelltextdatei
      VJCompilationTask vjCompilationTask = new
        VJCompilationTask(
          file.getLocation().toOSString(),
          getJavaSourceOutputPathName(file),
          null);

      // Übersetzung mit dem VJ-Compiler
      VJCompiler.vjcompile(
        "-s " + vjCompilationTask.getVjInputFileName() +
        " -d " + vjCompilationTask.getJavaSourceFileName());

      // Aktualisiere das Arbeitsprojekt, damit der
      // JavaBuilder die Java-Quelltextdateien übersetzt.
      getProject().refreshLocal(IResource.DEPTH_INFINITE,
        null);
    }
  }
...
}
```

Die erzeugten Java-Quelltextdateien sind Eclipse unbekannt. Durch eine erzwungene Aktualisierung des Arbeitsprojektes stellt Eclipse eine Veränderung fest und meldet dies allen *Buildern*. Der `JavaBuilder` übersetzt die generierten Java-Quelltextdateien in Java-Binärdateien. Nach diesen wenigen Schritten, die in Listing 6.1 zusammengefasst sind, besteht die Möglichkeit VJ-Quelltextdateien übersetzen zu lassen.

Als weiteren Schritt sollen die Meldungen, die beim Aufruf des VJ-Compilers entstehen, auf einer Konsole sichtbar gemacht werden. Dafür wird eine Eclipse-Konsole geöffnet und die zwei Ströme `System.out` und `System.err` für die Zeit der Übersetzung zu dieser Konsole gerichtet.

Listing 6.2: Konsole für den VJ-Compiler - verkürzte Fassung

```
public class VJBuilder extends IncrementalProjectBuilder {
...
  void buildVJ(IResource resource) {
...
    // Konsole
    MessageConsole vjConsole = findConsole("VJConsole");
```

```
    MessageConsoleStream vjOut = vjConsole.newMessageStream();
    MessageConsoleStream vjErr = vjConsole.newMessageStream();
        // In rot auf dem System.Err-Strom ausgeben
    vjErr.setColor(new Color(null, new RGB(255, 0, 0)));

        // sichere ursprüngliche Ströme für Wiederherstellung
    oldErr = System.err;
    oldOut = System.out;
    try {
      // Ströme umlenken
      System.setOut(new PrintStream(vjOut));
      System.setErr(new PrintStream(vjErr));
      // Übersetzung mit dem VJ-Compiler
      VJCompiler.vjcompile(
        "-s " + vjCompilationTask.getVjInputFileName() +
        " -d " + + vjCompilationTask.getJavaSourceFileName());
    } finally {
        // Stelle ursprüngliche Ströme wieder her
      System.setErr(oldErr);
      System.setOut(oldOut);
    }
  }
...
  private MessageConsole findConsole(String name) {
    ConsolePlugin plugin = ConsolePlugin.getDefault();
    IConsoleManager conMan = plugin.getConsoleManager();
    IConsole[] existing = conMan.getConsoles();
    for (int i = 0; i < existing.length; i++){

    // Wenn die Konsole bereits existiert, gib diese zurück
    if (name.equals(existing[i].getName()))
        return (MessageConsole) existing[i];
    }

    // keine Konsole gefunden, erzeuge eine neue
    MessageConsole myConsole = new MessageConsole(name, null);
    conMan.addConsoles(new IConsole[]{myConsole});
    return myConsole;
  }
}
```

In Listing 6.2 wird gezeigt, dass kaum Quelltext nötig ist, um eine Konsole für die Ausgabe zu erzeugen.

Der VJ-Compiler kann bis jetzt noch nicht auf die Klassen im Arbeitsprojekt zugreifen. Nur die Klasse, die zur Übersetzung an ihn direkt durchgereicht wird ist für ihn

sichtbar. Der `Classloader` der VJ-Compiler Bibliothek wird durch einen präparierten `VJClassLoader` ersetzt. Der Ablauf ist identisch mit dem der Konsole. Direkt vor der Übersetzung wird der `Classloader` des ablaufenden `Threads` durch den präparierten `VJClassLoader` ersetzt und gleich danach wieder zurückgesetzt. Der `VJClassLoader` implementiert die abstrakte Klasse `java.lang.ClassLoader`. Die Methode `findClass` muss angepasst werden, sodass sie Klassen auch im Arbeitsprojekt suchen kann. Der `VJClassLoader` ist Teil des VJ-Plugins, sodass dieser Zugriff möglich ist. Dabei werden nur binäre *class*-Dateien berücksichtigt. Liegt eine Klasse nicht in binärer Form vor, kommt es im VJ-Compiler zu der Ausnahme `JavaClassNotFoundException`. Diese ist eine `RuntimeException`, die in einer Exemplarvariable den Namen der gesuchten Klasse enthält. Das VJ-Plugin fängt diese Ausnahme und kann mit Hilfe der dort enthaltenen Informationen eine entsprechende VJ-Quelltextdatei suchen. Der Übersetzungsvorgang wird für diese Zeit angehalten. Das `VJCompilationTask` wird auf einem Stapel zurückgelegt. Wird die benötigte VJ-Quelltextdatei gefunden so wird die Übersetzung dieser initiert. Der hieraus generierte Java-Quelltext muss noch vom Java Übersetzer übersetzt werden. Pro Projekt kann nur ein *Builder* aktiv sein und der `JavaBuilder` nicht parallel zum `VJBuilder` gestartet werden. Der `BuildManager` von Eclipse bricht einen entsprechenden Aufruf ab. Deshalb wird die Übersetzung von Java-Quelltext mit Hilfe der *Java Compiler Api* durchgeführt.

Tritt bei der Übersetzung der Fall ein, dass zwei Klassen sich gegenseitig importieren, so soll der Übersetzungsvorgang abgebrochen werden. Beim Ablegen eines neuen `VJCompilationTask` auf den Stapel muss überprüft werden, ob bereits ein anderer `VJCompilationTask` auf dem Stapel vorhanden ist, der den gleichen Pfad einer VJ-Quelltextdatei enthält.

Der Aufruf des VJ-Compilers wird im `VJBuilder` rekursiv durchgeführt. Bei einer erfolgreichen Übersetzung wird der `VJCompilationTask` vom Stapel genommen, und der vorherige `VJCompilationTask` wird abgearbeitet. Der VJ-Compiler bricht jeweils beim ersten Fehler ab und gibt nur diesen wieder. Deshalb kann bei mehreren `imports` die Tiefe der Rekursion sich immer wieder erhöhen. Der VJ-Builder arbeitet ähnlich dem Backtracking-Verfahren.

6.1.2. Anpassungen am VJ-Compiler

Aus dem obigen Abschnitt geht hervor, dass Veränderungen am VJ-Compiler nötig gewesen sind. Die massgeblichen Änderungen sind die neue Schnittstelle und die Ausnahme `JavaClassNotFoundException`. Hier sollen noch die weiteren Änderungen, die in diesem Zusammenhang durchgeführt wurden, vorgestellt werden.

Die Übersetzung ist bis jetzt innerhalb der `main`-Methode abgelaufen. Der Ablauf wurde in zwei Methoden ausgelagert, die von der `main`-Methode aus aufgerufen werden, aber auch vom VJ-Plugin aus aufrufbar sind. Die erste Methode enthält die Analyseschritte der Übersetzung. Die zweite Methode ist für die Generierung des Java-Quelltextes zuständig. Diese Aufteilung gibt die Möglichkeit Quelltexte laufend auf Fehler zu scannen, ohne dass die Generierung von Java Binärdaten durchgeführt wird.

Der des VJ-Compilers wird mit einem `String-array` parametrisiert. Auf diese Weise kann bei bleibender Schnittstelle die Funktionalität des VJ-Compilers später erweitert

werden.

Eine weitere Änderung innerhalb des VJ-Compilers ist der Zugriff auf den `VJClassLoader`. Dieser wird über die Schnittstelle `java.lang.ClassLoader` benutzt, sodass keine Abhängigkeit zwischen VJ-Compiler und VJ-Plugin entsteht. Der `VJClassLoader` muss im VJ-Compiler von dem ablaufenden `Thread` bezogen werden, so kann sichergestellt werden, dass über keinen anderen `ClassLoader` versucht wird auf die Klassen im Arbeitsprojekt zuzugreifen. Weiter müssen die Aufrufe `Class.forName(name)` im VJ-Compiler durch `Class.forName(name, null, classLoader)` ersetzt werden, damit der richtige `ClassLoader` benutzt wird. Diese Änderungen müssen nur in der Klasse `TypeCheckerActions` durchgeführt werden. Listing 6.3 zeigt ausschnittsweise diesen Aufruf.

Listing 6.3: Aufruf des VJClassLoader im VJCompiler

```
class TypeCheckerActions {
. . .
  private void resolveType(ExprType exprType) {

  . . .
    // ClassLoader vom Thread holen
    ClassLoader classLoader = Thread.currentThread()
                              .getContextClassLoader();
    // Erweiterter Aufruf
    Class clazz = Class.forName(
                  exprType.getCompleteTypeName,
                  true,
                  classLoader); // VJClassLoader
        }
    . . .
}
```

Ein weiterer Punkt welcher für den Einsatz in Eclipse verändert werden muss ist die interne Speicherung über den Zustand der Übersetzung. Der VJ-Compiler speichert in einer `boolean` Variable, ob Fehler bei der Übersetzung aufgetreten sind. Diese Variable wurde bei jedem Neustart von der Kommandozeile aus zurückgesetzt. In Eclipse wird der VJ-Compiler nicht für jede Übersetzung neu gestartet. Der `TypeChecker` im VJ-Compiler ist ein *Singleton*. Stellt der `TypeChecker` einen Fehler bei der Übersetzung einer Datei fest, speichert er diesen in der boolschen Variable. Beim Aufruf des VJ-Compilers mit der nächsten Datei ist diese Variable mit dem Fehler-Wert belegt. Die Übersetzung bricht daraufhin ab, auch wenn der Quelltext einwandfrei ist. Der `TypeChecker` wird aus der ANTLR-Grammatik heraus generiert, deshalb muss die Anpassung in der ANTLR-Grammatik des VJ-Compiler durchgeführt werden. Die Entwicklung des VJ-Compiler für die Kommandozeile ist auch hier ein Problem gewesen.

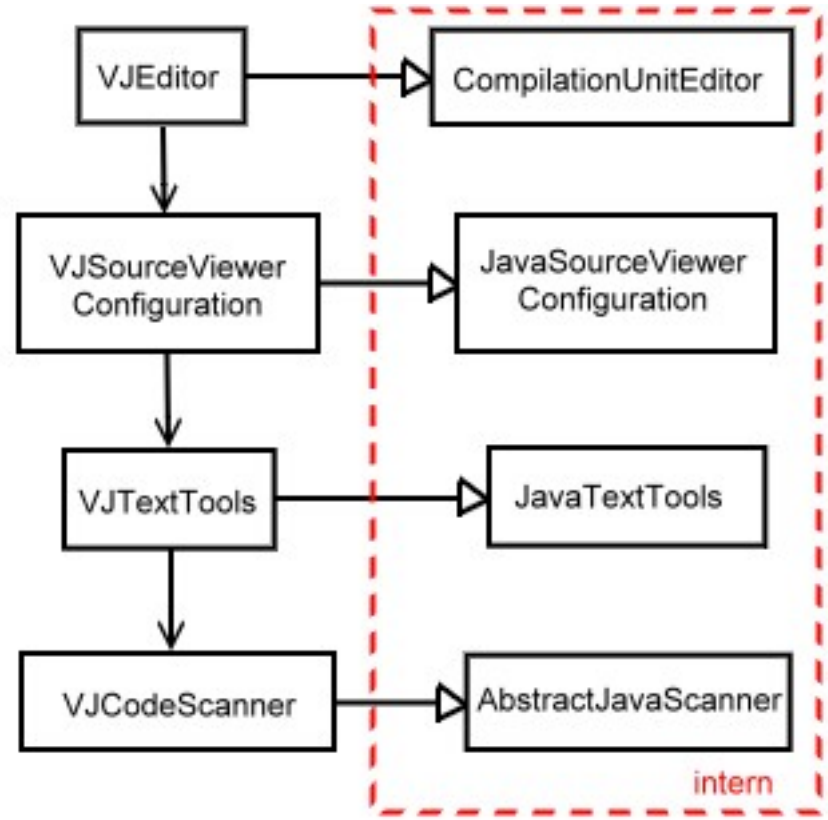

Abbildung 6.1.: Editorklassen für das Highlighting

6.1.3. Benutzungsoberfläche

Die Benutzungsoberfläche trägt stark zur Akzeptanz des Produktes bei. Für die Umsetzung soll der von Eclipse generierte XML-Editor an VJ angepasst werden. Es soll vom *internen* `CompilationUnitEditor` geerbt werden, wie dies auch vom AJDT Projekt gemacht wird. So wird größtmögliche Konformität mit dem Java-Editor bei gleichzeitig geringem Anpassungsaufwand des VJ-Editors gesichert.

Die Syntax-Hervorhebung muss an VJ angepasst werden. Hier kann genauso auf bereits exisitierenden Quelltext zurückgegeriffen werden. Drei an VJ angepasste Klassen `VJSourceViewerConfiguration`, `VJTextTools` und `VJCodeScanner`, werden erstellt. Die entsprechenden Aufrufe des Java-Editors im VJ-Editor müssen überladen werden. Wie in Abbildung 6.1 gezeigt wird, erben die drei zusätzlichen Klassen von entsprechenden *internen* JDT-Klassen. Die Schlüsselwörter für das *Highlighting* sind im `AbstractJavaScanner` lokalisiert, entsprechend werden diese für VJ im `VJCodeScanner` fest reingeschrieben.

Ein vergleichbarer Weg muss auch für *Code Completion* gegangen werden. Ein `VJCompletionProcessor` erbt vom *internen* `JavaCompletionProcessor` und wird an der `VJSourceViewerConfiguration` angemeldet. Hier besteht die Schwierigkeit, dass Vorschläge für das *Code Completion* nur kontextabhängig erzeugt werden. Der *interne* `JavaCompletionProcessor`, von dem geerbt wird, kann eine `valueclass` nicht interpretieren und generiert keine Vorschläge. Diese müssen im eigenen `VJCompletionProcessor` generiert werden. Folgende Schritte müssen dabei durchgangen werden:

- Erzeugen des Kontextes anhand der Quelltextdatei
- Sammlung der kontextabhängigen Vorschläge (`VJCompletionProposal`)
- Filtern und Sortieren der gesammelten Vorschläge

Das Gerüst für *Code Completion* ist schnell aufgebaut, sodass Vorschläge zur Verfügung gestellt werden können. Die Implementierung der ersten zwei Punkte ist allerdings sehr aufwendig. Die Logik des VJ-Compilers kann nicht benutzt werden, da dieser für diesen Anwendungsfall nicht gebaut wurde. Die Analyse des Quelltextes muss in den `VJCompilationProcessor` nochmal eingebaut werden.

6.1.4. Fehlererkenung

In diesem Abschnitt wird die Fehlerbehandlung im VJ-Plugin beschrieben. Für die Darstellung von Fehlern, die bei der Übersetzung auftreten, müssen Anpassungen am VJ-Compiler, `VJBuilder` und `VJEditor` vorgenommen werden.

Im ursprünglichen VJ-Compiler werden alle Fehlermeldungen mit Hilfe von `System.out` auf die Konsole ausgegeben. Das Analysieren dieses Ausgabestroms ist sehr aufwendig. Deshalb soll stattdessen der VJ-Compiler im VJ-Plugin so angepasst, dass an entsprechenden Stellen zusätzlich eine Ausnahme erzeugt und geworfen wird. Meldungen die auf Fehler in der VJ-Quelltextdatei hinweisen werden mit der im VJ-Compiler bereits vorhandenen `SemanticErrorException` bekannt gemacht. Die `JavaClassNotFoundException` wird neu eingeführt und unterscheidet sich nur geringfügig von der `SemanticErrorException`. Die erstere Ausnahme wurde angepasst, um Informationen über die Lokalisierung des aufgetretenen Fehlers im Quelltext zu transportieren. Diese Ausnahme wird bei der Erzeugung mit `SourceLocation` parametrisiert, die bis dahin allerdings nur zur Erzeugung der Fehlemeldung diente und danach verworfen wurde. Die `SourceLocation` soll für weitere Verarbeitung gespeichert werden.

`JavaClassNotFoundException` beinhaltet zusätzlich zur `SourceLocation` auch den Typ einer Java-Klasse, die bei der Übersetzung nicht auf dem Klassenpfad gefunden werden konnte. Diese Information wird für die Suche einer VJ-Quelltextdatei dieses Typs gebraucht. Die Abbildung 6.2 fasst die Gemeinsamkeiten und Unterschiede beider Ausnahmetypen.

Die Benutzung von Ausnahmen deckt nur einen begrenzten Teil aller Meldungen ab, die vom VJ-Compiler geliefert werden. Die Umleitung der `System.out`-Ströme (s. 6.1.1) ist deshalb weiterhin nötig.

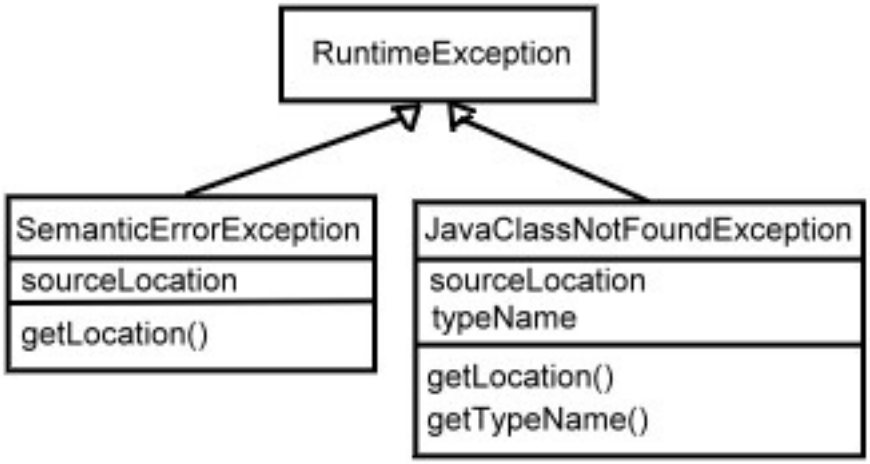

Abbildung 6.2.: Ausnahmen des VJ-Compilers zur Kommunikation von Übersetzungsfehlern

Damit die Ausnahmen sinnvolle Daten tragen, müssen zwei weitere Veränderungen am VJ-Compiler durchgeführt werden. Eine Veränderung bezieht sich auf die Sicherung der `SourceLocation` beim Analysieren des Quelltextes, damit sie im Fehlerfall übergeben werden kann. Im `MMCreator` wird den einzelnen Elementen des Modells einer VJ-Quelltextdatei ihre `SourceLocation` als Exemplarvariable mitgegeben.

Die zweite Veränderung bezieht sich auf interne von der ANTLR-Grammatik verarbeitete `RecognitionException`s. Diese dürfen nicht direkt im `VJParser` abgefangen werden, sondern weiter gegeben werden, sodass diese vom VJ-Compiler verarbeitet werden können.

Abbildung 6.3 zeigt die Reihenfolge der Ereignisse zwischen dem vom der ANTLR-Grammatik generierten Teil des VJ-Compilers, der Klasse `VJCompiler` und dem `VJBuilder`, welcher ausserhalb des VJ-Compilers im VJ-Plugin platziert ist. Der `VJParser` steht für den ersten von sechs Bearbeitungsschritten einer Übersetzung. Abbildung 6.3 soll die ganze Abfolge deshalb nur Ansatzweise darstellen (s. Abschnitt 3.4.

Die Fehlermeldungen werden vom `VJBuilder` entgegengenommen und verarbeitet. Die Schnittstelle `IFile` über die Dateien in Eclipse angesprochen werden können besitzt Methoden zur Erzeugung von *Markern*. Dem *Marker* kann eine Meldung und auch die schwere dieser übergeben werden. Für die genaue Positionierung der Fehlermeldung innerhalb der Quelltextdatei sorgen weitere Attribute, `LINE_NUMBER`, `CHAR_START` und `CHAR_END`.

Die Entwicklungsumgebung wertet alle Markierungen für sämtliche Dateien und Verzeichnisse ohne weiteres hinzutun aus und zeigt diese im Editor als auch in der *Problems*-Ansicht an. Die Navigation von der *Problems*-Ansicht zur entsprechenden Stelle im Quelltext ist auch integriert.

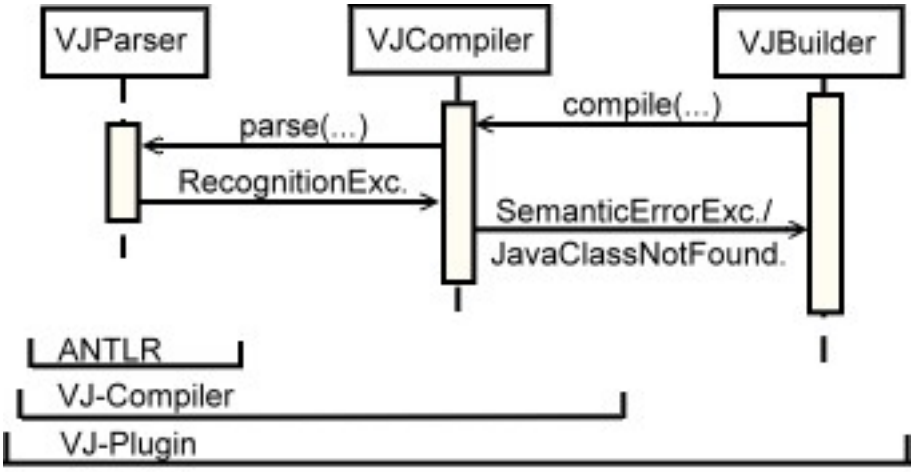

Abbildung 6.3.: Ausnahmen in ihrer zeitlichen Abfolge und Zugehörigkeit zu einzelnen Komponenten

6.1.5. Konfiguration von Projekten

Arbeitsprojekte bei denen VJ eingesetzt werden soll müssen dafür angepasst werden. Dafür wird die Projektnatur *VJNature* verwendet. Diese kann jedem Projekt einzeln hinzugefügt oder wieder entfernt werden. Beim Hinzufügen der Projektnatur wird ein Projekt als ein VJ-Projekt designiert. Dabei wird der `VJBuilder` zusätzlich zum `JavaBuilder` hinzugefügt. Es wird ein Dateifilter angepasst, sodass VJ-Quelltextdateien nicht als Konfigurationsdateien angesehen werden und in den Ordner mit Binärdaten kopiert werden. Weiterhin wird eine Markierung am Projektordner generiert, die das Projekt sichtbar als VJ-Projekt markiert.

Die Umsetzung benötigt die Erweiterung von drei Erweiterungspunkten. Zwei Erweiterungspunkte `org.eclipse.ui.popupMenus` und `org.eclipse.core.resources.-natures` werden von Eclipse vorgeneriert und müssen nur geringfügig angepasst werden. Der dritte Erweiterungspunkt `org.eclipse.ui.ide.projectNatureImages` benötigt nur eine Bilddatei, sodass diese Erweiterung keinen Quelltext benötigt.

Der erste Erweiterungspunkt fügt einen Menüpunkt im Kontextmenü des Projektes hinzu. Über diesen Menüpunkt kann die `VJNature` einem Projekt sehr einfach hinzugefügt werden. Ausser den Umbennenungen von XML nach VJ, muss noch die Sortierung der Projketnaturen korrigiert werden, sodass die `VJNature` in der internen Reihenfolge die erste Projektnatur ist. Die Reihenfolge ist wichtig für den dritten Erweiterungspunkt. Es wird nur das Bild der Projektnatur am Projektordner angezeigt, die die erste in der Reihenfolge ist.

Der zweite Erweiterungspunkt ist für die Projektnatur selbst zuständig. Die Implemetierende Klasse besitzt zwei Methoden `conigure()` und `deconigure()` mit denen

Anpassungen an der Konfiguration des Projektes durchgeführt werden können. Die Konfiguration des Projektes wird über zwei Schnittstellen durchgeführt. Der Filter für die VJ-Quelltextdateien wird am `IJavaProject` gesetzt. Ein *Builder* wird wiederum an einem `IProject` hinzugefügt.

Die *Builder* und die Projektnaturen eines Projektes werden in *arrays* gespeichert. So kann durch einfache *array*-Manipulationen die Reihenfolge festgelegt werden. Der Filter für die VJ-Quelltextdateien ist eine Option des Projektes, die als `String` vorgehalten und um **.vj* verlängert wird.

6.2. Einschränkungen

Dieser Ansatz ist sehr starr. Bei Veränderungen interner Schnittstellen von JDT kann es dazu kommen, dass das Plugin nicht mehr funktioniert. Erhöhter Wartungsaufwand ist für das Plugin bei diesem Ansatz wahrscheinlich. Weiterhin ist dies keine generische Lösung. Wird das JDT in zukünftigen Versionen um neue Funktionalität erweitert, so wird diese im VJ-Plugin fehlen. Es muss eine erneute Anpassung stattfinden. Dieser Ansatz führt auch dazu, dass der gesamte Quelltext nicht in VJ geschrieben werden kann, sondern in Java und VJ aufgeteilt werden muss. Da VJ ursprünglich dafür entwickelt wurde, dass der gesamte Quelltext vom VJ-Compiler übersetzt wird, müssen für diesen Ansatz einige Änderungen an VJ und dessen Benutzung vorgenommen werden. Diese Änderungen werden im Abschnitt 6.3 dargestellt.

6.3. VJ als graduelle Erweiterung

Der Erfolg von VJ hängt von vielen Punkten ab. Einerseits kann der Einsatz von VJ dem Programmierer in seiner Arbeit unterstützen. Andererseits ist VJ kein Standard bzw. keine Erweiterung, die in die Sprache integriert ist. Ein Kunde, für den Software entwickelt wird, wird eine nicht standardisierte Sprache nur ungern einsetzen. Er müsste entweder eigene Entwickler schulen oder sich an den Hersteller für die Lebenszeit der Software binden. Damit sind die Entwickler mit ihren Freiheitsgraden extrem eingeschränkt. Der Ansatz von VJ muss abgeändert werden, damit es eingesetzt werden kann. So kann vielleicht nur der Entwickler mit VJ arbeiten und sich damit die Arbeit erleichtern. Durch die Konvertierung in Java Code, wird der ausgelieferte Code wiederum dem Standard entsprechen und kann vom Kunden eingesetzt und vor allem auch gewartet. Es muß beachtet werden, daß ein System nicht vollständig in VJ entwickelt werden kann. Die Umsetzung mittels eines Übersetzungsvorgangs von VJ nach Java ist die Schwachstelle. Die Unterstützung aller Java Konstrukte müsste fortwährend in den VJ-Compiler in den VJ-Compiler nachgebaut werden. Dieser Aufwand ist enorm und i.d.R. wenig Bezug zu Werttypen. VJ würde in naher Zukunft mehr hinderlich als nützlich werden.

Es gibt drei Möglichkeiten diesem Problem zu begegnen. Eine davon ist den VJ-Compiler um die fehlende Funktionalität zu erweitern und den Übersetzungsprozess für den ganzen Quelltext um einen VJ-zu-Java Übersetzungsschritt zu erweitern. Bei

diesem Ansatz wird sichergestellt, dass global die VJ-Syntax verwendet werden kann, was dem Entwickler mehr Freiheit gibt. Weiterhin kann dieser Ansatz auch zu weniger Programmierfehlern führen, da global die gleichen Syntax-Regeln gelten. Bei den weiteren Ansätzen ist dies nicht der Fall, dort wird deutlich wie die Einschränkung der Benutzung von VJ auf bestimmte Software-Teile sich negativ auswirken kann. Ein großer Nachteil bei diesem Ansatz ist die Erweiterung des VJ-Compilers um die fehlende Java-Funktionalität. Diese Erweiterung kann sehr zeitaufwendig sein. Der interne Java Compiler von Eclipse wurde nach erscheinen von Java 5 fast ein Jahr lang an die neue Funktionalität angepasst [SM08] [EFa]. Auch würde der VJ-Compiler mit jeder neuen Version von Java an diese angepasst werden müssen. Die Übersetzungszeit verlängert sich durch den zusätzlichen Übersetzungsschritt. Dies kann sich negativ bei der Entwicklung auswirken [HJ74, s. 198]. Um die Zeit zu verkürzen, sollte der Schritt dort eingespart werden, wo VJ nicht benutzt wird.

Ein zweiter Ansatz sieht vor VJ nur lokal einzusetzen. Der Quelltext, der keine VJ-Konstrukte, wie `const` oder `==` zum Vergleich benutzt, soll nicht vom VJ-Compiler sondern nur noch vom Java kompatiblen Übersetzer übersetzt werden. Bei diesem Ansatz können drei verschiedene Klassentypen unterschieden werden:

- reine VJ-Wertklassen (`valueclass`)
- Java Klassen, die VJ-Konstrukte benutzen (im folgenden Hybrid-Klasse genannt)
- reine Java Klassen, die keine VJ-Konstrukte enthalten

Bei diesem Ansatz wird der VJ-Compiler nicht um neuere Java Konstrukte erweitert, die für VJ nicht notwendig sind. Der Einsatz dieser Java Konstrukte ist auf reine Java Klassen beschränkt, welche wiederum von einem Java Übersetzer übersetzt werden. Dies spart für diese Klassen einen unnötigen Übersetzungsschritt. Die zwei ersten Klassentypen, VJ und Hybrid-Klassen, müssen weiterhin vom VJ-Compiler übersetzt werden und können diese Konstrukte nicht enthalten.

Die Entkopplung des VJ-Compilers von der Entwicklung der Sprache Java ist ein großer Vorteil, da weiterhin Konstrukte neurer Java Version zusammen mit VJ in einem Projekt genutzt werden können. Nachteilig ist die Vielfalt der Klassentypen, die sich untereinander nur geringfügig unterscheiden, sodass Programmierfehler durch Verwechslung einfacher entstehen können. Einerseits ist VJ eine an Java angelehnte Sprache, sodass vom Ursprung her Verwechslungsgefahr besteht (s. Kapitel 7). Andererseits ist der Quelltext einer reinen Java Klasse von einer Hybrid-Klasse oft auch bei genauem betrachten nicht zu unterscheiden. Anhand der Dateierweiterung .vj können Hybrid-Klassen identifiziert werden. In dem Listing 6.4 wird am Beispiel des Vergleichs mit dem Operator `==` dieser Umstand gezeigt. Bei einer reinen `valueclass` besteht dieses Problem auch, ist aber durch das Schlüsselwort `valueclass`, welches den Klassentyp eindeutig definiert, abgeschwächt.

Listing 6.4: Diese Klasse kann sowohl eine Java Klasse als auch Hybrid-Klasse sein

```
public class IPFilter {
  public boolean isBlacklisted(IpAddress ip) {
```

```
    for (Iterator it = blacklist.iterator(); it.hasNext();) {
      IpAdress badIp = (IpAddress) it.next();

      // Der Vergleich mit == ist nur in Hybrid-Klassen gültig
      if (ip == badIp) {return true;}
    }
    return false;
  }
}
```

Die im zweiten Ansatz entstehenden Probleme stoßen bei Entwicklern negativ auf. Vor allem die Zweideutigkeit der Hybrid-Klassen kann zur Verwirrung führen. Durch Eliminierung der Hybrid-Klassen kann die Verwechslungsgefahr gemildert werden. Diesen Weg nimmt der nächste Ansatz, der in dieser Arbeit als Lösung gewählt wurde. Durch die Eliminierung der Hybrid-Klasse bleiben nur die zwei Klassentypen, in denen rein Java bzw. VJ geschrieben werden kann. Die Selektion von Werten aus Java heraus ist so nicht mehr möglich, denn das `const` kann nicht sinngemäß benutzt werden. Diesem Umstand soll aber mit Fabrikmethoden begegnet werden, mit denen die Selektion des Wertes in der Wertklasse selbst gekapselt wird [Blo01, s. 5].

Listing 6.5: Selektion eines Wertes aus einer Java Klasse heraus (korrekter Code)

```
public class InternationalBankAccount {
  int amount;
  String currency;

  public Money getBalance () {
    // Die Selektion erfolgt mit Hilfe
    // einer Fabrikmethode: valueOf(int, String)
    return Money.valueOf(amount, currency);
  }
}
```

Der VJ-Compiler muss hierfür angepasst werden. Ursprünglich generiert er einen `public` Konstruktor. Er soll zusätzlich eine Fabrikmethode generieren, die die gleichen Parameter wie der Konstruktor besitzt und diesen mit ihnen wiederum aufruft.

Die Reduktion der Anzahl von Klassentypen, führt zu klareren Regeln, in welcher Klasse welche Konstrukte benutzt werden dürfen. Die Unterscheidung dieser Klassentypen zeigt, dass VJ eine Erweiterung ist, die sich nur in einem stark eingegrenztem Gebiet bewegt. Es ermöglicht Klassen zu schreiben und zu benutzen, die einen Wertecharakter besitzen. Dieser eingeschränkte Einsatzbereich macht VJ zu einer Spracherweiterung, die keinen Einfluß auf andere Konzepte von Java nimmt, also graduell ist.

6.4. Zusammenfassung

Die Umsetzung des VJ-Compilers kann den JDT sehr ähnlich nachgebaut werden. Damit ist die Homogenität der Werkzeuge für VJ und Java gesichert. Andererseits ist der Ansatz der Vererbung von JDT-Klassen sehr starr. Bei zukünftigen Änderungen kann das VJ-Plugin unter Umständen nicht mehr lauffähig oder sein. Neue Werkzeuge bzw. Funktionen müssen fest nachgebaut werden. Der Ansatz der Da der ganze Quelltext nicht in VJ geschrieben werden kann, sind auch Anpassungen in der Benutzung von VJ nötig. In Abschnitt 7 wird der Einsatz dieses Plugins mit Hilfe von Testbenutzern evaluiert. Die Evaluation fand in mehreren Schritten statt, sodass die einzelnen Zwischenergebnisse sich massgebend auf die Umsetzung des VJ-Plugins und die Anpassungen an VJ, die in diesem Abschnitt beschrieben wurden, ausgewirkt haben.

7. Evaluation

Der Erfolg eines Softwareprodukts ist, wie bei jedem anderen Produkt, maßgeblich von der Akzeptanz der späteren Benutzer abhängig. Oftmals werden deshalb bereits während der Entwicklungszeit Akzeptanztests durchgeführt, um möglichst früh Feedback zu erhalten. So können oft Designänderungen früher geplant werden. Je früher eine Änderung durchgeführt werden kann, desto leichter ist diese umzusetzen. Die Entwicklung kann durch frühes und regelmäßiges Feedback besser kontrolliert und an die Wünsche der Benutzer angepasst werden. Die Akzeptanz und damit der Erfolg dieser Software wird dadurch vergrößert [BA05, Kapitel 14].

Für das VJ-Plugin wurde genauso Feedback von Benutzern eingeholt. Damit das Feedback präzise und anwendungsorientiert ist, sollten die Benutzer folgende Kriterien erfüllen:

- Gute Java Kenntnisse
- Programmierung mit Hilfe von IDEs, vor allem Eclipse
- Kenntnis über das Werttyp-Konzept
- Kenntnis des Ansatzes von VJ

Vor allem die Kenntnisse von VJ schränken die Auswahl der Benutzer hauptsächlich auf den Hochschulbereich ein. Zwei der Benutzer, zu der Zeit als diese Arbeit geschrieben wurde, sind Diplomanden gewesen, zwei weitere haben erst kürzlich einen Hochschulabschluss erworben.

Die Testbenutzer waren aufgefordert Werttypen zu programmieren. Dabei sollten sie sich möglichst an realen Anwendungsfällen, in denen sie mitgearbeitet haben, orientieren. Sie haben dabei die Benutzungsoberfläche, den Übersetzungsprozess und die Integration des VJ-Plugin mit dem JDT bewertet. Eine übergreifende Frage war, inwiefern VJ zusammen mit dem VJ-Plugin die Arbeit mit Werttypen erleichtert. Die Benutzer waren mit dem JWAM-Rahmwerk vertraut und haben einen Vergleich zu diesem gezogen.

Schwierig für das Feedback ergab sich die interne Trennung zwischen VJ-Plugin und dem VJ-Compiler. Für die Benutzer war es *eine* Erweiterung. Es wurde Feedback zum VJ-Compiler, zu VJ und dem VJ-Plugin gegeben.

Die erste Testversion war nach drei Wochen Entwicklungszeit noch sehr rudimentär. Der Editor erbte noch nicht von einem Java-Editor und unterstütze deshalb kein Syntax Highlithing bzw. korrekte Einrückung und Klammerergänzung. Es existierte kein angepasster Classloader, sodass nur sehr simple Klassen compiliert werde konnten. Die Kommunikation zwischen IDE, VJ-Plugin und VJ-Compiler war sehr lückenhaft,

sodass viele Meldungen von einer Übersetzung nicht angezeigt wurden. Vor allem Fehlermeldungen wurden nicht ausgegeben.

Für die zweite Testversion wurde der VJ-Editor an den Java-Editor angepasst. Das Syntax Highlithing wurde an VJ angepasst. Weitere Änderungen am VJ-Plugin wurden nicht vorgenommen. Das Syntax Highlighting hat die subjektive Akzeptanz der Benutzer sehr gesteigert. Es führte allerdings auch dazu, dass es bei der Benutzung zur Verwirreung kam. Optisch waren der VJ-Editor und Java-Editor nahezu identisch. Die Benutzer mussten jedoch darauf achten welchen Editor sie gerade benutzen. VJ ist eine Obermenge von Java, also kann im VJ Editor Java benutzt werden, andersherum gilt es nicht. Es wurden zwei Schlüsse hieraus gezogen. Einerseits muss der VJ-Editor sich deutlicher von dem Java-Editor unterscheiden. Andererseits sollten Hybrid-Klassen gemieden werden, da diese am meisten Verwechslungsgefahr bieten. Die hieraus gewonnenen Erkenntnisse über Trennung von Java und VJ sind näher in Abschnitt 6.3 beschrieben.

Auch bei dieser Testversion wurde die eingeschränkte Übersetzung und die unzureichende Meldungen des VJ-Compilers und VJ-Plugins bemängelt. Diese beiden Punkte hindern an einer sinnvollen Benutzung des VJ-Plugins. Als weitere wichtige aber fehlende Eigenschaften wurden *Code Completion* und *Organize Imports* aufgezählt.

In der dritten Testversion wurde der Übersetzungsvorgang erheblich verbessert. Ein selbstentwickelter Classloader ermöglichte die Übersetzung viel komplexerer Dateien, sodass imports eingesetzt werden konnten. Auch die Kommunikation zwischen IDE, VJ-Plugin und VJ-Compiler wurde verbessert. Alle Meldungen wurden korrekt ausgegeben und wenn nötig auch im Editor angezeigt. Weiterhin fehlten Funktionen, wie Code Completion oder Organize Imports.

Die Benutzung von VJ in der dritten Version des VJ-Plugins erlaubte den Benutzern sich genauer und länger mit VJ zu beschäftigen. Wegen der weiterhin fehlenden Funktionen ist das VJ-Plugin noch weit vom produktiven Einsatz entfernt, aber es machte für die Benutzer die Vorteile von VJ sichtbar. Die Ihnen wichtigsten Punkte waren, dass die Überprüfung der Korrektheit von Werttypklassen abgenommen wird. Sie müssen die Konventionen nicht selbst prüfen, weil der VJ-Compiler dies übernimmt. Ein weiterer Punkt war die Einfachheit der Benutzung, da das VJ-Plugin die Integration des VJ-Compilers in die IDE übernimmt und einen sinnvollen Einsatz von VJ ermöglicht.

8. Ausblick

Die Umsetzung des VJ-Plugins, wie sie in Abschnitt 6 beschrieben wurde, ist möglich. Doch diese Umsetzung ist sehr starr. Das VJ-Plugin kann bei Änderungen interner Schnittstellen nicht mehr einsatzbereit sein. Auch müssen, wie aus dem Abschnitt 5 zu erkennen ist, viele einzelne Schritte und Erweiterungen im VJPlugin für eine erfolgreiche Lösung zusammengetragen werden. In diesem Abschnitt wird der Ausblick auf einen alternativen Ansatz gegeben. Dieser versucht das Werttypkonzept mit den bereits vorhandenen Sprachmitteln von Java umzusetzen.

8.1. Java based Extension

In Abschnitt 5 wurden Konstrukte der Entwicklungsumgebung in den Vordergrund gestellt und untersucht. In diesem Abschnitt soll untersucht werden, wie Konstrukte der Sprache Java benutzt werden können, um die Unterstützung für die Erweiterung zu verwirklichen. Dieser Ansatz bietet einen sehr großen Vorteil der Unabhängigkeit von Entwicklungswerkzeugen, da nur auf die Konstruke der Sprache und nicht mehr auf die der Entwicklungsumgebung zurückgegriffen wird. Damit soll die Lösung unabhängig und nicht nur für eine Entwicklungsumgebung benutzt werden.

8.1.1. Java Annotations

Seit Java 5 werden Annotationen unterstützt. Sie sind vergleichbar mit Kommentaren, da sie in den Quelltext integriert sind, aber nicht vom Java Compiler verarbeitet werden. Diese Annotationen können jedoch während der Übersetzung verarbeitet werden. So kann die Annotation `@Valueclass` eine `class` als Wertklasse markieren, in Listing 8.1 wird dies beispielhaft gezeigt.

Listing 8.1: Beispiel einer Java Annotation

```
@Valueclass
public class Money{. . .}
```

Annotationen können im Quelltext an den Stellen platziert werden, an denen Modifizierer benutzt werden. So können Klassen, Methoden und auch Attribute annotiert werden. Für die Attribute von zusammengesetzten Werten, die auf Zahlen abgebildet werden, wie IP-Nummern, können Annotationen `@Min` und `@Max` für die Bestimmung des Werteuniversums benutzt werden. Bei unregelmäßigen Wertklassen, wie dem Datum, ist das allerdings nur eingeschränkt möglich. Dieser Ansatz unterscheidet sich sehr stark, da die VJ spezifische Syntax nicht benutzt wird, sondern durch Annotationen

ersetzt werden soll. Der so erstellte Quelltext ist vollkommen mit Java kompatibel und durch Java Werkzeuge benutzbar. Ein weiteres wichtiges Merkmal ist, dass dieser Ansatz bei den meisten Entwicklungsumgebungen eingesetzt werden kann. Dabei müssen die *IDE*s Java 6 unterstützen. Bei Eclipse, welches einen eigenen Übersetzer besitzt, muss in den Übersetzungsprozess mit Hilfe des Erweiterungspunktes `compilationParticipant` eingegriffen werden.

8.1.2. Markierung

Mit Hilfe von Annotationen kann eine Klasse als Wertklasse markiert werden. Während der Übersetzung wird ein eigener *Annotations Processor* aufgerufen und kann die Annotationen zusammen mit dem sonstigen Quelltext auswerten. Die Auswertung ist nicht trivial, da bei der Prozessierung der Inhalt der Annotation und der Kontext der Annotation übergeben wird. Der Kontext einer Annotation ist das annotierte Element. Aus dem Listing 8.1 ist für die Annotation `@Valueclass` der Kontext die Klasse `Money`. Diese ist über die Schnittstelle `javax.lang.model.element.Element` erreichbar.

8.1.3. Validierung

Im ersten Schritt der Übersetzung wird die Klasse darauf überprüft, ob sie die Voraussetzungen für eine Wertklasse erfüllt. Dabei werden die Regeln überprüft, die auch der VJ-Compiler überprüft, z.B. dass die Parameter und Rückgabewerte von Methoden nur Werte sind. Die Validierung kann von den entsprechenden Teilen des VJ-Compilers übernommen werden, da der VJ-Compiler diese Funktion bereits bietet. Alternativ müssten die Logik für die Validierung neu geschrieben werden.

8.1.4. Modellerzeugung

Im nächsten Schritt erfolgt die Auswertung der Annotationen. Da der Annotation Prozessor nur lesenden Zugriff auf die Klasse hat, kann diese nicht bereits in diesem Schritt verändert werden. Mit den ausgewerteten Informationen wird ein neues abgeändertes Modell der bestehenden Klasse erstellt. Dies ist ein zweiter Teil dessen Logik sich an der des VJ-Compilers orientiert.

Dieses Modell wird in Java Binärcode übersetzt. Dafür wird die Java 6 *Compiler Api* benutzt. Der neue Binärcode muss in einer gesonderten Datei vorgehalten werden. Er wird beim Laden der Klasse gebraucht werden.

8.1.5. Laden der Klasse

Es sind nun zwei Binär-Dateien für unsere Klasse vorhanden. Eine ist unverändert, die zweite ist die zusätzlich erzeugte Klasse. Beim Laden der Klasse kann die *Java Instrumentation API* für die Instrumentierung nicht benutzt werden. Die Mächtigkeit dieser API ist nicht ausreichend. So kann mit der *Instrumentation API* folgendes verändert werden:

- Körper einer Methode

- Felder
- Die Menge der Konstanten

Das Hinzufügen, Entfernen oder Umbennen einer Methode oder eines Attributes ist jedoch nicht möglich. Die Erstellung der Methode `equals()` und `hash()` ist aber für VJ essentiell.

Im Gegensatz zur *Instrumentation API* muss in den Prozess des Ladens einer Klasse aktiv eingegriffen werden. Es wird ein eigener `ClassLoader` gebraucht, von dem aus ein Instrumentierer die originale Klasse beim Laden mit dem im vorherigen Schritt erstelltem Modell anreichert.

Weil die *Instrumentation Api* nicht ausreichend mächtig ist, muss für die Instrumentierung, wie sie für VJ nötig ist, ein fremder Instrumentierer benutzt werden bzw. ein eigener entwickelt werden.

8.1.6. Limitierung

Das Annotation Processing ist auf Lesenden Zugriff beschränkt. Hierdurch ist die Instrumentierung nicht statisch zur Übersetzungszeit durchführbar. Die Instrumentierung muss bei jedem Ladevorgang viel kostspieliger erkauft werden. Die Instrumentierung selbst kann nicht mit mitteln der *Java Platform* durchgeführt werden, wodurch momentan eine Abhängigkeit von Dritt-Software bzw. erheblicher Mehraufwand für die Entwicklung eines Instrumentierers entsteht. Es ist allerdings nicht ausgeschlossen, dass die *Instrumentation Api* in ihrer Mächtigkeit erweitert wird [SM06, java.lang.instrument.Instrumentaion].

Durch die Instrumentierung der Klasse unterscheidet sich der ablaufende Code vom geschriebenen Quelltext. Dies kann zu Schwierigkeiten bei der Suche nach Fehlern führen.

8.2. Zusammenfassung

Im Vergleich zu dem Ansatz eines Eclipse-Plugins ist dieser Ansatz sehr flexibler. Dieser Ansatz bezieht sich nicht auf Klassen von Eclipse und ist damit auch mit anderen IDEs einsetzbar. Da hier ausschliessllich Konstrukte von Java benutzt werden, ist die Unterstützung durch sämtlich Werkzeuge gewährleistet, sofern diese Annotationen unterstützen.

VJ als eigene Sprache ist bei diesem Ansatz nicht mehr vorhanden, allerdings wird durch einen zusätzlichen Bearbeitungsschritt (Annotation Processing) ein vergleichbarer Ansatz zu VJ gewählt. Dieser Ansatz kann das Werttyp-Konzept ähnlich gut umsetzen.

9. Zusammenfassung

Für die meisten Programmiersprachen existieren IDEs, welche den Programmierer in seiner Arbeit unterstützen. Sprachen werden konstant um neue Konzepte erweitert. Diese Erweiterungen dienen auch dazu, den Entwickler besser zu unterstützen. Während IDEs an neue, offizielle Versionen von Sprachen von eigenen Entwicklerteams angepasst werden, muss bei individuellen Spracherweiterungen die Anpassung selbst durchgeführt werden. Graduelle Erweiterungen erweitern die ursprüngliche Sprache nur geringfügig.

In dieser Arbeit wurde die IDE-Unterstützung für graduelle und indviduelle Spracherweiterungen untersucht. Wegen des graduellen muss die Unterstützung nicht vollständig nachgebaut werden. Es kann auf die Unterstützung der ursprünglichen Sprache zurückgegriffen werden.

Als Fallbeispiel wurden VJ und Java zusammen mit Eclipse gewählt. Trotz einer sehr flexiblen Plugin-Architektur kann die Unterstützung für VJ nur sehr starr an die Java-Unterstützung angebaut werden. Nur so kann eine gute Übereinstimmung zwischen den Werkzeugen sichergestellt werden, ohne die Werkzeuge nachbauen zu müssen. Hauptsächlich dafür verantwortlich ist die Verschlossenheit der JDT. JDT nutzt nicht die Möglichkeiten von Eclipse Extensions, um selbst erweitert werden zu können. Die JDT sind nur durch Verletzung von Schnittstellen sinnvoll einzusetzen.

Ein alternativer Ansatz ist VJ mit Java Annotations umzusetzen. Er bietet eine sehr gute Verbreitung, da die Bindung an eine Entwicklungsumgebung entfällt. Dieser Ansatz ist auch mit zukünftigen Java-Versionen kompatibel, sofern Annotations in ihrer jetzigen Form weiter unterstützt werden.

A. VJ-Plugin

A.1. Console

Der VJ Compiler ist eine eingebundene Bibliothek in das VJ Plugin. Somit werden alle Meldungen, die der VJ-Compiler ausgibt an keine Konsole weitergegeben. Im folgenden werden zwei Ansätze gezeigt, wie dieses Problem gelöst werden kann.

A.1.1. MessageConsoleStream

Beim ersten Ansatz wird der org.eclipse.ui.console.MessageConsoleStream an den VJ Compiler übergeben, sodass der VJ Compiler über diesen Stream auf eine Eclipse Console seine Meldungen ausgeben kann. Bei diesem Lösungsansatz muss der VJ Compiler vollständig in das VJ Plugin integriert oder zu einem eigenen Eclipse Plugin Projekt konvertiert werden, um den Eclipse-eigenen MessageConsoleStream importieren und benutzen zu können.

Listing A.1: MessageConsoleStream als Parameter

```
public class VJBuilder {
  MessageConsole console = findConsole("vjconsole");
  MessageConsoleStream out = console.newMessageStream();
  try{
    VJCompiler.vjcompile(file.getLocation().toOSString(),
    getOutputPathName(file), out);
  } catch (Exception e) {
    out.println(e.toString());
  }
}

public class VJCompiler{
  .
  public static void vjcompile(
  String inputFileName,
  String outputName,
  OutputStream outStream) throws Exception{
    outputStream = outStream;
    .
    out("Running VJCompiler");
    .
```

```
  }
  .
// es muss festgestellt werden, ob java.io.PrintStream
// oder org.eclipse.ui.console.MessageConsoleStream
// für die Ausgabe benutzt werden soll
  private static void out(String output){
    if (outputStream instanceof PrintStream){
      ((PrintStream) outputStream).println(output);
    } else if (outputStream instanceof MessageConsoleStream) {
      ((MessageConsoleStream) outputStream).println(output);
    } else throw new RuntimeException("No Console for output");
  }

  private static OutputStream outputStream;
}
```

A.1.2. Umleiten von System.out und System.err

Bei der Übergabe des MessageConsoleStream an den VJ Compiler entstehen viele ungewollte Bedingungen. Dieser Ansatz versucht diese Probleme zu umgehen. Es wird nicht der VJ Compiler verändert sondern die Umgebung um den Compiler herum. Der VJ Compiler benutzt ausschliesslich System.out und System.err Ströme für die Ausgabe an die Console. Für die Zeit des VJ Compiler Aufrufs werden diese globalen Ströme umgeleitet und nach dem Aufruf wieder zurück gestellt. Somit ist kein Eingriff in den VJ Compiler nötig.

Listing A.2: Umleitung der System.out und System.err Ströme

```
public class VJBuilder{
  .
  MessageConsole console = findConsole("vjconsole");
  MessageConsoleStream out = console.newMessageStream();
  oldErr = System.err;  //Save for restore
  oldOut = System.out;  //Save for restore
  System.setErr(new PrintStream(out));
  System.setOut(new PrintStream(out));

  try{
    VJCompiler.vjcompile(file.getLocation().toOSString(),
    getOutputPathName(file));
  } catch (Exception e) {
    System.err.println(e.toString());
  } finally {
    System.setErr(oldErr);       //restore
    System.setOut(oldOut);       //restore
```

```
  }
  .
}
```

A.2. Eclipse Extensions

Hier sollen die einzelnen Erweiterungspunkte, die vom VJPlugin benutzt werden aufgelistet werden. Ihre einzelnen Einstellungsoptionen sollen genauer erklärt werden. Zuerst soll der Allgemeine Aufbau von Erweiterungspunkten und Erweiterungen gezeigt werden, später die im VJ Plugin benutzten Erweiterungspunkte genauer betrachtet werden.

A.2.1. Aufbau eines Erweiterungspunktes

In einer EXSD-Datei wird das Schema für einen Erweiterungspunkt gespeichert. Alle Erweiterungen zu diesem Erweiterungspunkt, müssen anhand dieses Aufgebaut sein. Die Erweiterung muss die in der Schemadatei geforderten Informationen bieten, dabei sind einige erforderlich andere aber nur optional.

Listing A.3: Beispielhafter Aufbau einer Schemadatei für ein Erweiterungsplugin

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- Schema file written by PDE -->
<schema targetNamespace="leerEcl33">
<annotation>
  <appInfo>
    <meta.schema plugin="leerEcl33"
          id="de.uni_hamburg.informatik.vj.leerEcl33"
          name="Leerer Erweiterungspunkt"/>
  </appInfo>
  <documentation>
      [Enter description of this extension point.]
  </documentation>
   </annotation>
   <element name="ErweiterungsPunkt">
      <complexType>
         <attribute name="point" type="string" use="required">
            <annotation>
               <documentation>
               </documentation>
            </annotation>
         </attribute>
         <attribute name="id" type="string">
            <annotation>
```

```
            <documentation>
            </documentation>
         </annotation>
      </attribute>
      <attribute name="name" type="string">
         <annotation>
            <documentation>
            </documentation>
            <appInfo>
               <meta.attribute translatable="true"/>
            </appInfo>
         </annotation>
      </attribute>
   </complexType>
</element>
<annotation>
   <appInfo>
      <meta.section type="since"/>
   </appInfo>
   <documentation>
      [Enter the first release in which
                  this extension point appears.]
   </documentation>
</annotation>
<annotation>
   <appInfo>
      <meta.section type="examples"/>
   </appInfo>
   <documentation>
      [Enter extension point usage example here.]
   </documentation>
</annotation>
<annotation>
   <appInfo>
      <meta.section type="apiInfo"/>
   </appInfo>
   <documentation>
      [Enter API information here.]
   </documentation>
</annotation>
<annotation>
   <appInfo>
      <meta.section type="implementation"/>
   </appInfo>
   <documentation>
```

```
        [Enter information about supplied implementation
                        of this extension point.]
      </documentation>
   </annotation>
   <annotation>
      <appInfo>
         <meta.section type="copyright"/>
      </appInfo>
      <documentation>
      </documentation>
   </annotation>
</schema>
```

A.2.2. Editor

Der Erweiterungspunkt `org.eclipse.ui.editors` wird von Editoren benutzt, um von der Eclipse Umgebung eingebunden zu werden. Der Erweiterungspunkt kann mit folgenden Optionen konfiguriert werden.

- id[1] - eindeutiger Identifikator für den Editor
- name[1] - Name des Editor. Der Name wird in der Benutzungsoberfläche angezeigt
- icon - Graphik (16x16 Pixel), für visuelle Unterscheidung der Editoren, kann als relative Pfadangabe zum Plugin angegeben werden.
- extensions - Dateitypen, in Form einer Kommagetrennten Liste von Datei erweiterungen, die der Editor unterstützt.
- class[2] - Eine Klasse, die `org.eclipse.ui.IEditorLauncher` und damit den Editor implementiert.
- command[2] - Ein Ausführungskommando zum Starten eines externen Editor. Diese Ausführungsdatei muss auf dem Systempfad oder im Pfad des Plugins vorhanden sein
- launcher[2] - Eine Klasse, die wiederum einen externen Editor öffnet. Die Klasse muss `org.eclipse.ui.IEditorLauncher` implementieren
- contributorClass[3] - Fügt neue Aktionen in die Toolbar und das Menü, die der Editor als zusätzliche Features mitbringt. Muss `IEditorActionBarContributor` implementieren.
- default - Wahrheitswert ob dieser Editor der StandardEditor ist. Zum Zeitpunkt der Arbeit wird Anhand der Dateierweiterung und des Dateinamens s. nächsten Punkt, entschieden. Nützlich bei mehreren Editoren für einen Dateityp.

- filenames - Filter für Dateinamen. Nur Dateien mit Dateinamen, die zum Filter passen, werden im Editor geöffnet.
- symbolicFontName - Font für den Editor, s. `org.eclipse.ui.fontDefinitions`
- matchingStrategy[3] - Mit Hilfe dieser Klasse kann weiter unterschieden werden, ob der Inhalt der Datei passend für diesen Editor ist, und ob dieser Editor dementsprechend zum öffnen benutzt werden soll. Die Klasse muss `org.eclipse.ui.IEditorMatchingStrategy` implementieren.

[1] Diese Felder müßen ausgefüllt werden
[2] Diese Felder schließen sich gegenseitig aus
[3] Darf nur gesetzt sein, wenn 'class' gesetzt ist

B. Datenmedium

Dieser Diplomarbeit wird ein Datenmedium beigelegt. Auf diesem Medium befindet sich folgendes:

- Gesamter Quelltext der Projektes
- VJ-Plugin Bibliothek
- Literatur von flüchtigen und schnell änderbaren Medien (Websites, integrierte Hilfefunktion von Eclipse)

Die Literatur findet sich in einem Unterordner 'Literatur' wieder. Jeder Eintrag besitzt wiederum einen eigenen Ordner, dessen Name der Signatur im Literaturverzeichnis entspricht.

Literaturverzeichnis

[BA05] Beck, Kent und Cynthia Andres: *Extreme Programming Explained.* Addison-Wesley, Pearson International Edition, 2 Auflage, 2005.

[Bar07] Bartlett, Neil: *A Comparison of Eclipse Extensions and OSGi Services.* EclipseZone, 2007. Veröffentlicht im Internet unter http://www.eclipsezone.com/articles/extensions-vs-services/. Letzer Abruf: 03. Jan. 2008.

[Böh06] Böhm, Oliver: *Aspecktorientierte Programmierung mit AspectJ 5.* dpunkt.verlag, 2006.

[Blo01] Bloch, Joshua: *Effective Java.* Addison Wesley, 2001.

[Boe03] Boekhoudt, Caspar: *The Big Bang Theory of IDEs.* ACM Press, 1(7):74–82, 2003.

[BOSW98] Brach, Gilad, Martin Odersky, David Stoutamire und Philip Wadler: *Making the future safe for the past: Adding Genericity to the JavaTM Programming Language.* ACM, 1998.

[Dau06] Daum, Berthold: *Java-Entwicklung mit Eclipse 3.2.* dpunkt.verlag, 4 Auflage, 2006.

[Dau07] Daum, Berthold: *Rich-Client-Entwicklung mit Eclipse 3.2.* dpunkt.verlag, 2 Auflage, 2007.

[Dmi05] Dmitriev, Sergey: *Meta Programming System*, 2005. Veröffentlicht im Internet unter http://www.jetbrains.com/mps/index.html. Letzer Abruf: 30. Apr. 2008.

[EFa] Eclipse Foundation, Inc.: *Eclipse Project 3.1 Maintenance Release Schedule.* Veröffentlicht im Internet unter http://www.eclipse.org/eclipse/development/eclipse_maintenance_schedule_3_1_x.html. Letzer Abruf: 24. Jan. 2008.

[EFb] Eclipse Foundation, Inc.: *Help Contents in Eclipse.* JDT Plug-in Developer Guide.

[EF08] Eclipse Foundation, Inc.: *Dynamic Languages Toolkit*, 2008. Veröffentlicht im Internet unter http://www.eclipse.org/dltk/. Letzer Abruf: 30. Apr. 2008.

[GJSB05] Gosling, James, Bill Joy, Guy Steele und Gilad Bracha: *Java(TM) Language Specification, The (3rd Edition)*. Prentice Hall PTR, 3 Auflage, 2005.

[Got05] Goth, G.: *Beware the March of this IDE: Eclipse is overshadowing other tool technologies*. Software, IEEE, 22(4):108–111, 2005.

[Hei05] Heiden, Markus: *Generierung von Fachwerten aus einem abstrakten Sprachmodell*. Diplomarbeit, Universität Hamburg, 2005.

[HJ74] Hoare, C.A.R. und C.B. Jones: *C. J. Bunyan, State of the Art Report 20: Computer Systems Reliability*, Kapitel Hints on programming language design, Seiten 505–534. Pergamon/Infotech, 1974.

[Jem05] Jemerov, Dmitry: *Language API, IDEA 5.0*, 2005. Veröffentlicht im Internet unter http://www.jetbrains.com/idea/plugins/developing_custom_language_plugins.html. Letzer Abruf: 30. Apr. 2008.

[Lad03] Laddad, Ramnivas: *AspectJ in Action*. Manning, Greenwich, 2003.

[Mül99] Müller, Klaus: *Konzeption und Umsetzung eines Fachwertkonzepts*. Diplomarbeit, Universität Hamburg, 1999.

[OW97] Odersky, Martin und Philip Wadler: *Pizza into Java: Translating theory into practice*. ACM, 1997.

[Rat06] Rathlev, Jörg: *Ein Werttyp-Konstruktor für Java*. Diplomarbeit, Universität Hamburg, 2006.

[Rie07] Rieken, Johannes: *Design by Contract for Java - Revised*. Diplomarbeit, Carl von Ossietzky Universität Oldenburg, 2007.

[Rit03] Ritterbach, Beate: *Eigene Werttypen in Java*. JavaSpektrum, 04:46–50, 2003.

[RRS07] Rathlev, Jörg, Beate Ritterbach und Axel Schmolitzky: *Auf der Suche nach Werten in der Softwaretechnik: Werte und Objekte in objektorientierten Programmiersprachen*. In: *Software Engineering*, Seiten 261–262, 2007. Die veröffentlichte Version ist eine Kurzfassung. Die lange Version wurde nicht veröffentlicht.

[SM03] Sun Microsystems, Inc.: *Understanding the Manifest*. Sun Developer Network, 2003. Veröffentlicht im Internet unter http://java.sun.com/developer/Books/javaprogramming/JAR/basics/manifest.html. Letzer Abruf: 02. Jan. 2008.

[SM06] Sun Microsystems, Inc.: *Java^{TM} Platform, Standard Edition 6 API Specification*, 2006. Veröffentlicht im Internet unter http://java.sun.com/javase/6/docs/api/. Letzer Abruf: 23. Feb. 2008.

[SM08] Sun Microsystems, Inc.: *J2SE Code Names*, 2008. Veröffentlicht im Internet unter http://java.sun.com/j2se/codenames.html. Letzer Abruf: 24. Jan. 2008.

[Sof08] Software, Tiobe: *TIOBE Programming Community Index*, 2008. Veröffentlicht im Internet unter http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. Letzer Abruf: 16. Apr. 2008.

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich bin mit einer Einstellung in den Bestand der Bibliothek des Departments Informatik einverstanden.

Hamburg, den 5. Mai 2008
Marek Walczak