TUHH
Technische Universität Hamburg-Harburg

Integration eines Regelinterpreters in eclipse

Studienarbeit

Vorgestellt von:

Mathias Freier
Mathias.Freier@tu-harburg.de

Matrikelnummer: 15338

Unter Aufsicht von:

Prof. Dr. J. W. Schmidt
STS - TUHH

M.Sc. Miguel Garcia
STS - TUHH

Hamburg, den 10.03.2004

Inhaltsverzeichnis

Abbildungsverzeichnis

Zeichenerklärung

Drei Formatierungen zur Abgrenzung zu ‚Text in deutscher Sprache':

Begiff:

Begriffe aus dieser Domain, auch Begriffe aus anderen Sparten, die nicht als Bestandteile der Umgangssprache zu verstehen sind.

Ausnahmen:

- Manche Begriffe werden so häufig verwendet, dass deren Identifikation durch den Leser vorausgesetzt wird.
- Einigen Begriffen sind ganze Abschnitte gewidmet, somit werden jene ebenfalls als bekannt vorausgesetzt.

`Quellcode:`

```
Bezeichnungen von Klassen, oder auch Begriffe, die sich mit der
Funktionalität der repräsentierenden Klassen decken.
Auszüge aus Programmtexten.
```

[Literaturverweis]:

Literaturhinweise Seite I im Anhang

Abkürzungen

#	Anzahl
API	Application Program(ming) Interface
AWT	Advanced Widget Toolkit
CLIPS	C Language Integrated Production System
ECA	Event Condition Action
EMF	Eclipse Modeling Framework
GUI	Graphical User Interface
HTML	HyperText Markup Language
JDT	Java Development Tools
Mac OS	Betriebssystem
MS Windows	Betriebssystem
PDE	Plugin Development Environment
SDK	Standard Development Kit
SWT	Standard Widget Toolkit
UML	Unified Modeling Language
UNIX	Betriebssystem
XML	Extensible Markup Language

Kapitel 1

Annäherung an das Studienarbeitsthema

Dieser Studienarbeit liegt nicht nur eine konkrete Implementierungsfunktion zu Grunde, wie Beispielsweise die Erstellung eines Programms, welches eine bestimmte Aufgabe zu erfüllen hat. Vielmehr handelt es sich um eine Untersuchung darüber, in wie weit Regeldefinitionen und Integration einer Regelmaschine in ein Laufzeitsystem, hier eclipse, helfen könnten, Funktionalität von anderen Programmteilen zu erweitern.

1.1 Problemstellung

Eine mögliche Problemstellung wäre die folgende:
Zwei verschieden Plugins beschreiben den gleichen Sachverhalt, ohne voneinander zu wissen.

Man kann auf der Abbildung 1.1 sehen, dass auf verschiedenen Ebenen / Schichten eine Darstellung eines Projektes stattfinden kann. Die Synchronisation solcher darstellenden Plugins kann eine der Aufgaben eines eingebundenen Regelinterpreters sein.

Auch kann es innerhalb eines bestehenden Plugins zu Inkonsistenzen kommen. Betrachtet man beispielsweise die Modellierung von Projekten mittels UML, so kann es vorkommen, dass die Kombination von Symbolen zwar der offiziellen Syntax von UML entsprächen, jedoch keinen Sinn machten. Solche ‚Fehler' könnten dann mittels Regeln detektiert werden. Man vergleiche hierzu die entsprechende Veröffentlichung, auf die im Literaturverzeichnis: Anhang S. II [IinUML] verwiesen wird.

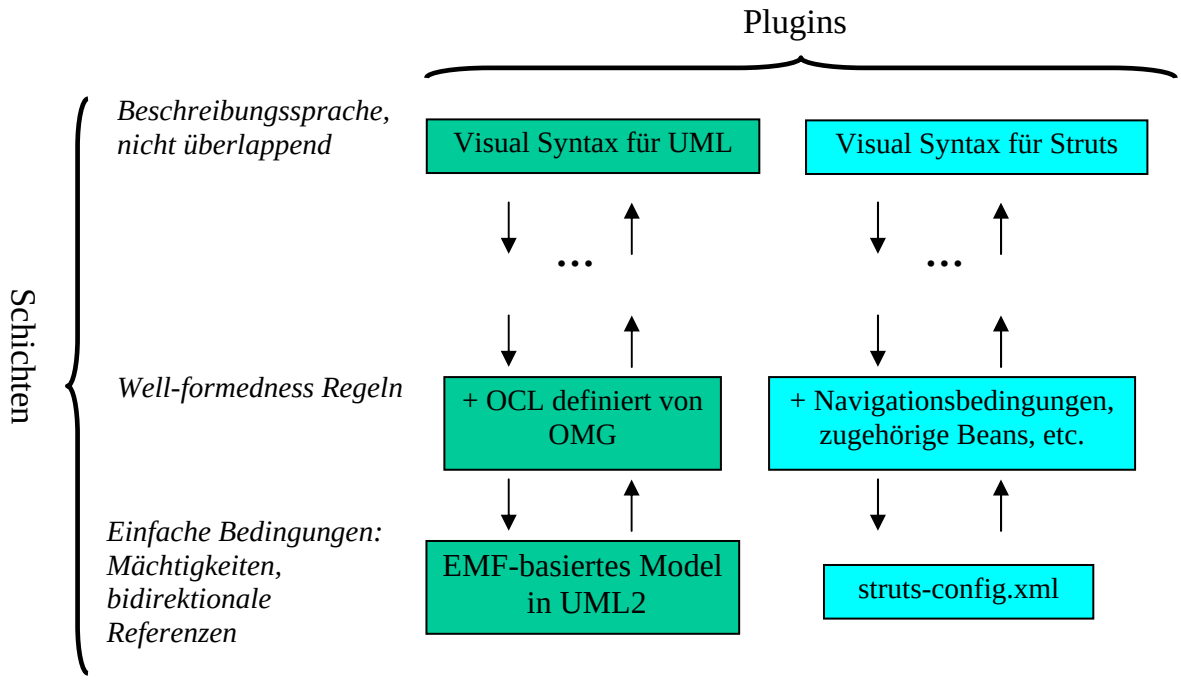

Abbildung 1.1: Problemstellung

1.2 Vorgehensweise

Die Implementierung eines Plugins, welches eine Regelmaschine in eclipse integriert, ist Teil und Kern dieser Studienarbeit.
Es werden dazu vielerlei Untersuchungen vorgenommen, die sich auf verschiedene Themengebiete erstrecken.

Aufbau und Eigenschaften von Regelmaschinen werden untersucht und exemplarisch an der Regel-Maschine jess aufgezeigt.

Die folgende Untersuchung ist in zwei unabhängige Aufgabenstellungen aufteilbar. Zum Einen gilt es einen Regel-Interpreter so einzubinden, dass aus der Laufzeitumgebung heraus Regeln in diesen eingefügt werden können.
Um Funktionalität von anderen Programmen, die Teil von eclipse sind, zu erweitern, müssen Fakten automatisch in die Regel-Maschine gelangen.

Der Aufbau und die Funktionsweise von eclipse, als Laufzeitsystem und Entwicklungsumgebung, sollen beschrieben werden. In eclipse wird dann im ersten Teil eine Regelmaschine integriert, die es ermöglicht, beliebigen jess eigenen Quellcode auszuführen. Nun können Fakten und Regeln manuell zur Regelbasis hinzugefügt werden.

Dieser zweite Teil der Untersuchung macht es notwendig, von bestehenden Plugins, die nicht für so etwas vorgesehen sind, Informationen abzugreifen. Es wird eine Klasse von Programmteilen herausgegriffen, die als solche Datenquellen dienen werden. Dabei handelt sich hierbei um Plugins, die mit Hilfe des Eclipse Modeling Framework (EMF) generiert wurden.

Eine Untersuchung und Vorstellung von EMF, als Teilprojekt von eclipse, ist somit auch Teil dieser Studienarbeit geworden.

Die Präsentation der Erkenntnisse wird sich an dem Ablauf der Entwicklung des Plugins, wie zuvor beschrieben, orientieren. Notwendiges Vorwissen zum Verständnis der Vorgehensweise, wie sie im Plugin implementiert ist, wird vor der Beschreibung der Programmteile erläutert.

Abschließend werden Überlegungen angestellt, ob die Mächtigkeit dieses Werkzeuges, durch Einbindung von so genannten Querysprachen erhöht werden kann. Es wird ein kurzer Einblick in ein Subprojekt von EMF gegeben. Diese Projekt nennt sich SDO und stellt Funktionalität ähnlich eines Applikations-Servers zur Verfügung.

Kapitel 2

Regel-Maschinen

Die Motivation für den Einsatz von Regelmaschinen liegt in der Möglichkeit, Daten und Verarbeitungslogik zu separieren. Daten werden in diesem Zusammenhang Fakten (englisch: facts) und Verarbeitungslogik Regeln (englisch: rules) genannt. Regeln können dann auf Fakten mit einer generischen Regel-Maschine (englisch: rule-engine) angewendet werden.
Regeln können von Domain-Experten erstellt werden. Anders als bei der Implementation eines Programms durch einen Programmierer, der gewöhnlich nicht aus der Domain stammt, die die Regeln hervorbringt. Auch solch ein Programmierer erstellt ein Programm, in das die Regeln eingebunden sind.
Eine Regelmaschine bietet nun die Möglichkeit, die Regeln, unter gewissen Vorraussetzungen, effizient zu evaluieren.
Eine Regel besteht im allgemeinen aus zwei Teilen, der Voraussetzung und der Folgerung, auch Prämisse und Konklusion, manchmal auch Aktion, genannt.
Sollte die Prämisse einer Regel erfüllt sein, so ist diese Regel aktiviert und wird in die Agenda verschoben. Aus allen aktivierten Regeln wird nun, nicht deterministisch, *eine* herausgesucht, die ausgeführt wird. Man spricht davon, dass die Regel feuert. Aus dem Feuern von Regeln kann eine Veränderung der Faktenmenge resultieren, sowie es zur Aktivierung weiterer Regeln kommen kann.
Man beachte hier, dass die nicht deterministische Reihenfolge des Feuerns von Regeln auch Auswirkungen auf Entwicklung von der Faktenmenge, auch Arbeitsspeicher genannt, haben kann.

Allgemein spricht man bei dieser resultierenden Hintereinanderausführung von Regeln von Regelverkettung. Es wird in Vorwärtsverkettung und Rückwärtsverkettung unterschieden.

Die Rückwartsverkettung setzt die Vorgabe einer zu prüfenden Schlussfolgerung voraus. Dann wird geprüft, für welche Regeln die Folgerungen mit der vorgegebenen übereinstimmen. Deren Prämissen werden dann als neue Schlussfolgerung genommen. Solch eine Rückwärtsverkettung bietet beispielsweise Prolog (a.a.O. S. v) und soll hier nicht weiter verfolgt werden.

Bei der Vorwärtsverkettung werden Fakten vorgegeben und alle Regeln durchsucht, ob deren Prämissen erfüllt sind. Der Aufbau solcher Regeln gleicht dem von `if... then...` Konstrukten in Programmiersprachen. Wenn die Prämisse erfüllt ist, werden bestimmte Aktionen ausgeführt.

Man betrachte hierfür so genannte ECA-Regeln (a.a.O. S. v), wie sie auch von jess evaluiert werden. Bei einem Ereignis wird ein Fakt in die Wissensbasis übernommen. Daraufhin werden ‚alle' Regeln überprüft und Instanzen von denjenigen, für die die Prämisse erfüllt ist, in die Agenda verschoben (vgl. Abbildung 2.1).
Eine dieser Regeln wird dann ausgewählt und feuert. Es kommt dabei zumeist zu einer Veränderung des Arbeitsspeichers. Die gefeuerte Regel wird aus der Agenda entfernt. Der Kreislauf wird dann so lange fortgesetzt, bis keine Regeln mehr aktiviert werden.
Dieser Mechanismus, wie gerade beschrieben, wird auch von dem Regelinterpreter / der Regel-Maschine, jess ausgeführt.
Für die Überprüfung der Übereinstimmung von Prämissen und Fakten hat sich als ‚de facto Standard' der Rete-Algorithmus etabliert. Die daraus entstehenden Vorteile werden anhand des jess Regelinterpreters beschrieben.

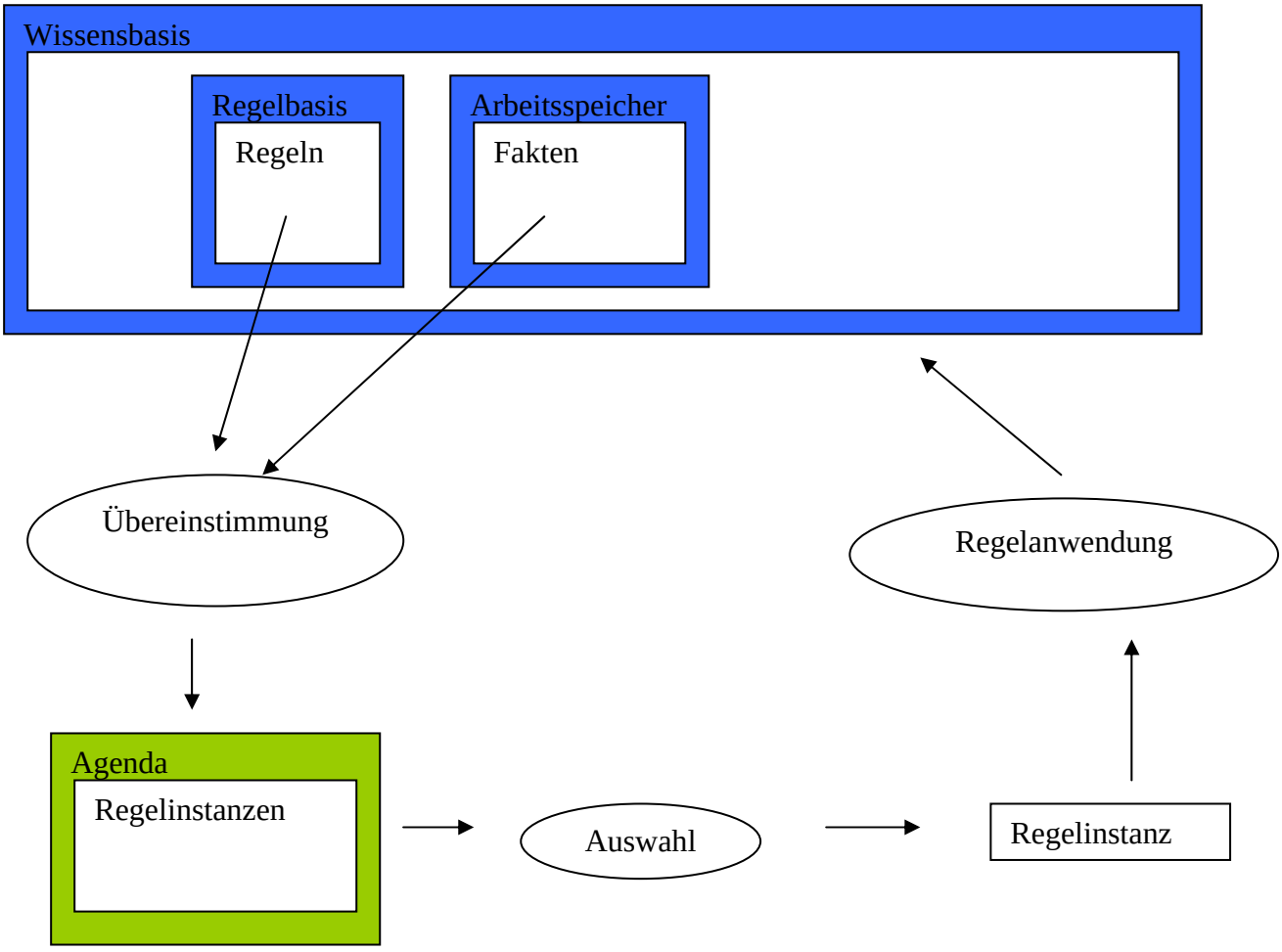

Abbildung 2.1: Arbeitskreislaufs eines Regelinterpreters

Kapitel 3

Der Regelinterpreter / die Regel-Maschine jess

Jess ist der Nachfolger von CLIPS (a.a.O. S. v) und ist komplett in Java geschrieben. Ernest J. Friedman-Hill von der Abteilung Distributed Computing Systems von Sandia National Laboratories stellt diese Regel-Maschine für nicht gewerbliche Nutzung kostenlos zur Verfügung (siehe http://herzberg.ca.sandia.gov/jess/). Ein typisches Anwendungsgebiet von jess ist das Erstellen von Expertensystemen.

Ich wähle jess für meine weiteren Betrachtungen und für die Integration in eclipse aus folgenden Gründen. Die Tatsache, dass jess in Java geschrieben ist und auch über eine umfassende API (a.a.O. S. v) für die Anbindung an Java-Applikationen bietet, macht es einfacher, jess in eclipse zu integrieren.
Es stellt sich leider im Laufe der Studienarbeit heraus, dass die Dokumentation diesbezüglich mangelhaft ist. Jess implementiert den effizienten Rete-Algorithmus und kann zudem Java-Beans beobachten und bei Änderungen automatisch Fakten erzeugen. Während CLIPS als älterer Vertreter von Regelinterpretern Verbreitung gefunden hat, ist jess durch seine Implementation in Java kontaktfreudig zu neuen Technologien. Es ist auch möglich, beliebige in Java geschriebene Methoden von jess aus auszuführen. Java-Instanzen können der Wissensbasis übergeben werden und der Laufzeitzustand von jess kann archiviert werden.
Jess hat eine eigene Definitionssprache, die sich sehr stark an die des Vorgängers CLIPS anlehnt, es wird hier von jess-Quellcode oder einfacher von jess-Code gesprochen.

3.1 Der Rete-Algorithmus

Wie schon im vorangehenden Artikel erwähnt, wird der Rete-Algorithmus benutzt, um zu überprüfen, welche Prämissen der Regeln aufgrund der Menge der Fakten des Arbeitspeichers erfüllt sind. Dabei werden 2 Annahmen gemacht, die Vorrausetzung für die Effizienz des Algorithmus' sind.

Hierzu ein Auszug [Marti], S. I: ...

- Der Arbeitsspeicher verändert sich bei der Anwendung von Regeln nur langsam. Typischerweise liegt die Summe der Anzahl gelöschter und neu hinzugefügter Fakten unter 10% der Anzahl Fakten im Arbeitsspeicher. Damit empfiehlt es sich, die Agenda nicht bei jedem Durchlauf neu zu berechnen. Vielmehr kann die anfänglich berechnete Agenda abgespeichert und danach anhand der Veränderungen des Arbeitsspeichers inkrementell nachgeführt werden.

- In vielen Anwendungen tritt die gleiche (oder zumindest eine ähnliche) Bedingung in verschiedenen Regeln auf. Solche Bedingungen müssen nur einmal mit den Fakten im Arbeitsspeicher verglichen werden, und nicht einmal pro Vorkommen in einer Regel. Um dies auszunützen, müssen beim anfänglichen Laden der Regeln alle vorkommenden Bedingungen in einem Netzwerk angeordnet werden. ...

Der Rete-Algorithmus erstellt nun einen Graphen von Teilbedingungen der Prämissen der Regeln. Es werden Meta-Informationen mit gespeichert, *welche* der Teilbedingungen zu überprüfen sind. Nun werden nur diejenigen Teilbedingungen überprüft, die auf geänderte Fakten weisen. Wichtig ist hier auch, dass wenn verschiedene Regeln gleiche Teilbedingungen haben, diese nicht für jede Regel neu überprüft werden müssen.

Da die Auswahl der zu aktivierenden Regeln nach jedem Feuern einer Regel geschieht, ist der Arbeitsaufwand in diesem Punkt immens.
Sollte man diese Auswahl ‚naiv' mittels drei `Schleifen` implementieren: über alle Regeln, alle Teilbedingungen, Suche über alle Fakten, so wäre der Aufwand nach [Marti], S. I O(Regeln x #Teilbedingungen je Regel im Durchschnitt x #Fakten).

3.2 Shadow-Facts

Eine interessante Möglichkeit, Fakten dem Arbeitsspeicher hinzuzufügen, ist in jess implementiert. Man kann so genannte shadow-facts erstellen. Damit ist folgendes gemeint:
Ein Java-Bean (siehe hierzu deren Spezifikation unter http://java.sun.com/products/javabeans/) hat Attribute, so genannte properties, auf die mittels `get()`- und `set()`-Methoden zugegriffen werden kann. Für Fakten ist es notwendig, eine ‚Schablone' vorher bereitzustellen. Zu diesem Zweck können bei Java-Beans die Laufzeitanalyse-Fähigkeiten ausgenutzt werden.
Eine Java-Bean Instanz kann nun statisch oder dynamisch zum Arbeitsspeicher hinzugefügt werden. Statisch bedeutet, dass so etwas wie ein Schnappschuss von den Werten des Beans gespeichert wird. Wenn das Bean dynamisch eingebunden werden soll, so muss es die Möglichkeit unterstützen, `PropertyChangeListener` einzutragen. Werden bei einer Änderung alle Listener informiert, so wird auch der Fakt in dem Arbeitsspeicher von jess generiert. Genaueres zu solchen Informationsmechanismen, die auf dem Event-Konzept beruhen, wird später noch erklärt.
Es ist somit möglich, solche Java-Beans genau zu beobachten, ohne Java-Code schreiben zu müssen. Um die Serialisierbarkeit des Laufzeitzustandes weiterhin gewährleisten zu können, ist es notwendig, die Schnittstelle `java.io.Serializable` bei den Beans zu implementieren.

Ich benutze diesen Mechanismus in dem Plugin nicht, da ich aufgrund fehlender Dokumentationen den Aufwand dieses Mechanismus' nicht abschätzen kann. Ein weiterer Grund ist, dass ich die zu observierenden Objekte (hier sei auf später verwiesen) nicht selbst instanziere und somit auch keine Kontrolle über deren Eignung für diesen Mechanismus habe.

Diese Möglichkeit bleibt jedoch auch im Zusammenhang mit dieser Untersuchung interessant, wenn man auf entsprechende Datenquellen stoßen sollte, die diese Vorraussetzungen erfüllen. Erweiterungen des Plugins, siehe Kapitel 9.8: Punktweise Erklärung zur Erweiterung der Observation, könnten einfach realisiert werden.

3.3 Ausführung von Java-Methoden

Von jess aus kann man sowohl statische Methoden wie auch objektgebundene Methoden aufrufen. Zur Laufzeit können Klassen mittels `Import`-Anweisungen bekannt gemacht werden, und dann können beliebige Methoden, auch mit Parametern, ausgeführt werden.
Auf diese Art und Weise ist es auch möglich, durch Fakten aus einem Plugin, Änderungen an einem anderen Plugin auszuführen.
Denkbar sind auch Suchanfragen auf Datenstrukturen; Überlegungen dazu werden am Ende des Berichtes zu finden sein.

Kapitel 4

eclipse

Zu untersuchen ist ja, ob die Einbindung einer Regel-Maschine, wie jess, die Funktionalität einer Laufzeitumgebung sinnvoll erweitern kann. Die zu betrachtende Laufzeitumgebung soll eclipse sein. Um alles aus diesem Bericht verstehen zu können, ist eine Einführung über den Aufbau von eclipse notwendig, auch wenn in späteren Abschnitten zusätzlich einiges, wie zum Beispiel der Benachrichtigungsmechanismus, detaillierter beschrieben wird.
Dazu werden im Folgenden einige Ausschnitte aus einer von Sven Pecher und mir zusammengestellten Präsentation zu Hilfe genommen.

Anders als bei manchen anderen Open Source Projekten ist eclipse eines, das die Unterstützung vieler gewichtiger Firmen vorweisen kann. Hierzu ein kurzer historischer Rückblick, der auch die Größe dieses Projektes verdeutlichen soll:

1999 - IBM und OTI Teams beginnen mit der Entwicklung
- 40 Vollzeit Entwickler

2001 - Version 1.0
- Code wird Open Source (unter der Common Public License)
- Eclipse.org startet

2003 - Version 2.1
- 40+ Firmen im Eclipse Board
- 400+ Plugins verfügbar
- 100+ kommerzielle Projekte sind aus diesem Open Source Projekt entstanden.

4.1 Grundgedanke

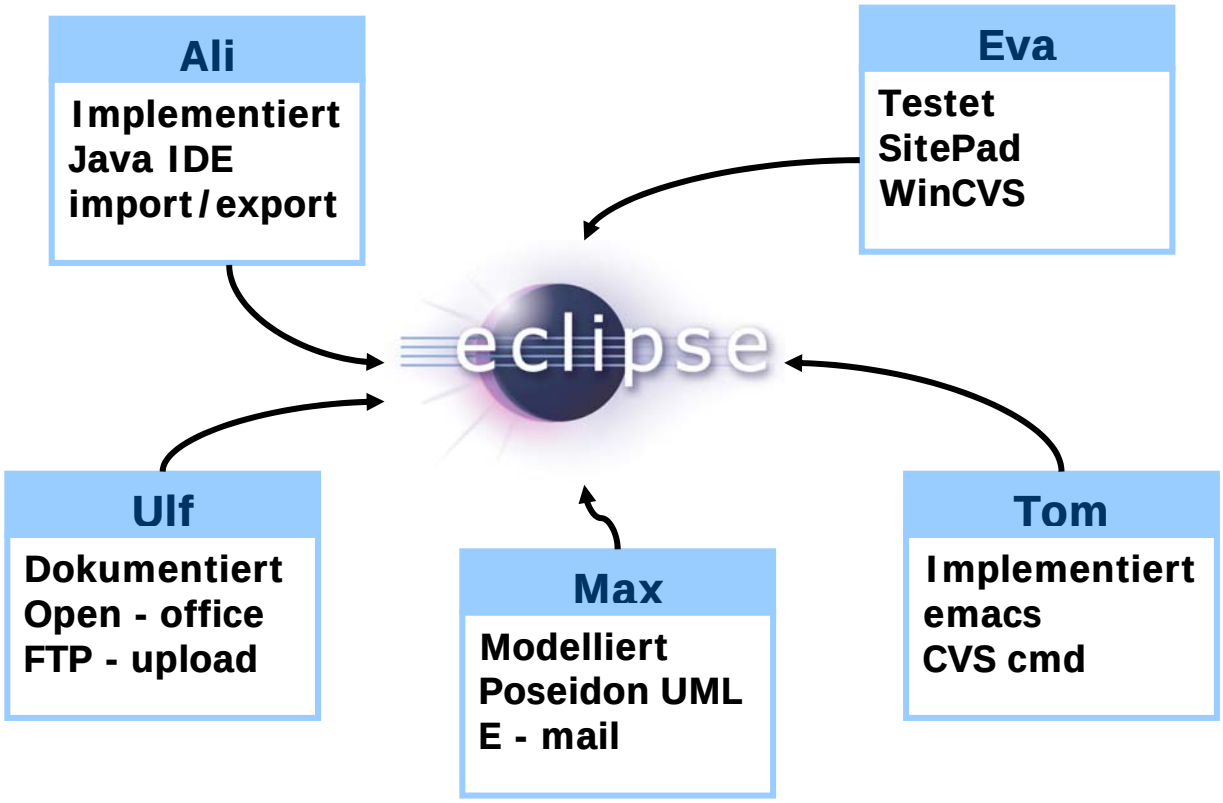

Abbildung 4.1: Grundgedanke von eclipse

Eclipse ist als Ablaufumgebung für Plugins gemacht und läuft auf der Basis von Java. Aus diesem Grunde ist eclipse nahezu Platform-unabhängig: UNIX, MS Windows, MacOS (a.a.O. S.v) werden unterstützt, um nur einige zu nennen.
Plugins können verschiedene Arten von Aufgaben erfüllen. Eclipse soll die Erstellung solcher Plugins vereinfachen und mit seinen mitgelieferten Werkzeugen unter anderem die verschiedenen Aufgaben obiger Grafik vereinen. Durch eine einheitliche Umgebung sollen Probleme, die ansonsten durch unterschiedliche Werkzeuge entstehen, umgangen werden. Insbesondere für die Java – Programmentwicklung werden mächtige Plugins zur Verfügung gestellt.

4.2 Architektur

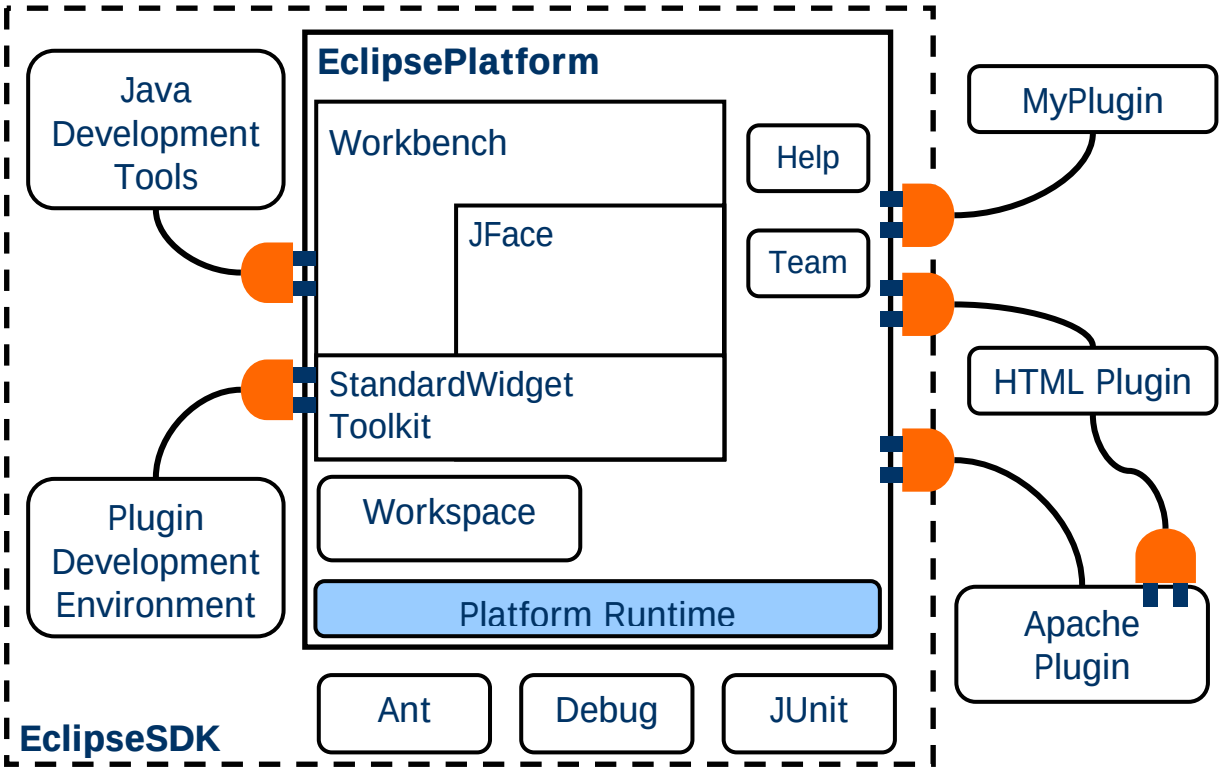

Abbildung 4.2: Architektur von eclipse

Eclipse besteht aus einem Kern, der in der Lage ist, Plugins auszuführen - und aus diversen Plugins. Zur Verdeutlichung soll die Abbildung 4.2 dienen.

Wie man aus der Grafik erkennen, kann besteht das herunterladbare Eclipse SDK (a.a.O. S. v) aus mehreren Plugins und der Einheit, die für den Ablauf der Plugins verantwortlich ist, hier ‚Platform Runtime' genannt. Die Plugins: Java Development Tools, kurz JDT, so wie das Plugin Development Environment, kurz PDE, sind nützlich für die Erstellung eigener Plugins.

4.3 Workspace

Der Workspace organisiert die Verwaltung von Projekten und deren Metadaten. Projekte können verschiedene Naturen haben: Java, C , Plugin, EMF... (a.a.O. S. v).
Alle Dateien werden in Verzeichnissen gespeichert, auf die auch von außerhalb von eclipse zugegriffen werden kann. Dateien, Verzeichnisse und Projekte werden als `Resourcen` bezeichnet und es werden vom Workspace betriebssystemunabhängige Zugriffsmechanismen zur Verfügung gestellt. Außerdem werden Metadaten gehalten. Im späteren Verlauf wird öfter von Resourcen gesprochen; die deutsche Bezeichnung ‚Ressource' wird dann absichtlich nicht verwendet.
Auch die Fähigkeiten zur Unterstützung für das Programmieren im Team und zur Rekonstruktion von Quellcode aus einer automatisch generierten Historie sollen hier kurz Erwähnung finden.

4.4 Workbench

Mit dem Workbench wird jene Plugin - Sammlung bezeichnet, welche für die Darstellung von eclipse - Plugins zuständig ist. Es können Menus erweitert, sowie Editoren, Views, Navigatoren angezeigt werden. Grafische Elemente können zu so genannten Perspektiven zusammen gestellt werden. Eine Unterstützung für Dialoge und Hilfeeinbindung machen eclipse benutzerfreundlicher.
Bestandteile des Workbench ist das Standard Widget Toolkit (SWT), welches, anders als AWT (a.a.O. S. v) bei Java, grafische Elemente durch die betriebssystemeigene Oberfläche darstellt. Somit wird die gewohnte Arbeitsumgebung erhalten und viel an Zuverlässigkeit und Geschwindigkeit gewonnen. Abstrahiert wird SWT durch JFace. JFace stellt Views, Editoren und andere grobkörnige Grafikelemente zur Verfügung.
Benutzereingaben werden von dem Workbench entgegengenommen und nach dem Prinzip ereignisbasierter Systeme (s. Kapitel 6.1) weitergeleitet.

4.5 Plugin Registry

Damit Plugins geladen werden, müssen sie sich und ihre Einstiegspunkte bekannt machen. Die Datei; `plugin.xml`, die jedes Plugin aufweist, enthält alle wichtigen Informationen, die für die Zusammenstellung der Plugins notwendig sind.
Es werden Abhängigkeiten zwischen verschiedenen Plugins definiert, um die Reihenfolge, in der diese geladen werden müssen, einhalten zu können.
Plugins werden erst geladen, wenn diese benötigt werden; man spricht hier von ‚lazy loading'. Dazu werden alle `plugin.xml` Dateien gelesen. Und Stellen, an denen sich das Plugin beteiligen möchte, herausgestellt. Solche Stellen werden als extension-points bezeichnet.
Extensions werden von den Plugins definiert, um an diesen extension-points geladen zu werden. Ein Plugin, welches die Bearbeitung von `*.clp` Dateien unterstützt, wird beispielsweise erst dann geladen werden, wenn eine solche Datei geöffnet oder erstellt wird.

Kapitel 5

Ausführung von jess-Quellcode

Einleitend wurde beschrieben, dass das Plugin in zwei Schritten entsteht. Es folgt nun eine Beschreibung derjenigen Programmteile, die dafür notwendig sind, um jess-Quellcode auszuführen. Wie in der Programmierung üblich, werden Bezeichner und Klassennamen aus dem Englischen entliehen - Hilfedateien, Kommentare und für den Benutzer Sichtbares werden der deutschen Sprache entnommen.
Die Struktur des Plugins ist in dem ersten Teil noch zu trivial, als dass eine Übersicht notwendig wäre. Im zweiten Teil des Plugins wird eine Übersicht folgen - Kapitel 9.1 Architektur des Observer Plugins. Eine Übersicht über den Ablauf bietet die Abbildung 5.1.

5.1 Benutzung

Jess-Quellcode kann direkt ausgeführt werden. Einzige Bedingung hierzu ist, dass der Code in einer Datei im Workbench mit der Endung: '`clp`' steht.

Dazu sind drei Möglichkeiten vorhanden;

- Man markiert den auszuführenden Code, und wählt aus dem Menu, welches man durch Klicken der rechten Maustaste öffnet, 'an jess senden'.
- Man wählt im Navigator (Ansicht des Workspace in Baumdarstellung) die entsprechende Datei oder eine übergeordnete aus und klickt 'an jess senden'. Es werden dann alle untergeordneten Dateien mit der Endung '`clp`' ausgeführt.
- Unter den Einstellungen zu diesem Plugin ist eine Datei als Initialisierungsdatei anzugeben. Diese Datei wird dann dem Regelinterpreter zum Initialisierungszeitpunkt übergeben.

Ausgaben werden in den ersten beiden Fällen auf die Regel-Ausgabe-Ansicht umgeleitet. Im zweiten Teil wird noch beschrieben, dass die automatische Ausgabe an- und ausgestellt werden kann.

Um diese Funktionalität zur Verfügung stellen zu können, ist folgendes zu implementieren:

1. plugin.xml
2. Selektionsverarbeitung
3. Einbettung der Regel-Maschine
4. Ausgabe-Ansicht
5. Laden des Laufzeitzustandes und von Initialdateien
6. Einstellungen
7. Hilfe für Benutzer

5.2 plugin.xml

In der plugin.xml wird festgelegt, welche Klasse das Plugin repräsentiert und wann und wo der Workbench erweitert werden soll.
Die Klasse `ObserverPlugin` aus dem Paket `observer` wird instanziert, sobald das Plugin geladen wird. Von hier aus werden `Resourcen` und eigene Instanzen verwaltet.
Eine so genannte 'contribution' ist in der plugin.xml zu erstellen, um bei dem Menu, welches mit der rechten Maustaste geöffnet wird (popupMenu), den Eintrag 'an jess senden' zu erhalten.
Hier ein Beispiel:

```
<extension point = "org.eclipse.ui.popupMenus">
     <objectContribution id="jessEngine.FileContext"
          objectClass = "org.eclipse.core.resources.IFile"
          nameFilter = "*.clp">
          <action label = "an jess senden"
               class="jessEngine.JessSelection"
               id="jessEngine.FileContext.rule_jess"
               menubarPath="additions"
               enablesFor="+">
          </action>
     </objectContribution>
</extension>
```

Die eclipse Runtime-Instanz wird bei Auswahl von '`action label = "an jess senden"`' in dem '`"org.eclipse.ui.popupMenus"`', also im Workbench, bei der Klasse '`class="jessEngine.JessSelection"`' die `run()`-Methode mit der `Action` '`id="jessEngine.FileContext.rule_jess"`' aufrufen, falls mindestens eine `Resource` ('`enablesFor="+"`') zuvor selektiert worden ist.

Andere Extensions werden definiert , um Gleiches mit Ordnern und Projekten zu verknüpfen, sowie bei Selektionen aus einem Editor heraus.

5.3 Selektionsverarbeitung

Die Klasse JessSelection aus dem Paket `jessEngine` hat, wenn man so will, „2 Eingänge und einen Ausgang“.
Ein: Sie implementiert `IActionDelegate` und wird somit immer dann informiert, wenn etwas selektiert wird. Die Selektion wird gespeichert. Damit diese Selektionen verfolgt werden können, wird das Plugin beim Start von eclipse geladen. Für die Observation, im zweiten Teil beschrieben, ist das auch notwendig.
Ein: Wird aufgrund der oben beschriebenen '`conribution`' die `run()` Methode mit einer `Action` aufgerufen, wird die letzte Selektion analysiert. Das heißt: entweder wird die Selektion aus einem Editor genommen, oder es wird eine Liste aller untergeordneten `*.clp` Dateien erstellt.
Aus: Diese Liste wird dann an die aktuelle Regel-Maschinen-Instanz übergeben `[jessEngine.jessIt((IFile)file)/jessEngine.jessIt((String)select)].`
Aufrufe von Methoden aus der Klasse `JessSelection`, beziehungsweise auch deren Instanzierung, finden ausschließlich von außerhalb des Observer-Plugins statt.

5.4 Einbettung der Regel-Maschine

Den Zugriff auf die Regel-Maschine jess, welche als `*.jar` - Datei zur Verfügung steht, wird von der Klasse `JessEngine` aus dem Paket `jessEngine` organisiert. Da diese Klasse die Funktionalität der Regel-Maschine zur Verfügung stellt, wird hier auch von dieser Klasse als Regel-Maschine, stellvertretend zu der gekapselten Regel-Maschinen-Instanz (`jess.Rete`), gesprochen.
Bei der Instanzierung von `JessEngine` trägt diese sich bei der Plugin-Klasse `ObserverPlugin` ein. Soll eine beliebige Klasse bei der `Rete`-Instanz etwas ausführen, so wird zuerst bei `ObserverPlugin` die `JessEngine` - Instanz geholt, welche die `Rete` - Instanz birgt. Daraufhin kann eine Methode von JessEngine ausgeführt werden, die dann auch Auswirkungen auf die `Rete`-Instanz haben kann.

5.5 Ausgabe-View

Die Klasse `RuleOutputView` registriert sich bei der Instanzierung bei `ObserverPlugin`. Somit kann über die Plugin-Klasse auf den `View`, gleichbedeutend mit Ansicht, zugegriffen werden. Solch eine Ansicht (ist ein `ViewPart`) besteht in diesem Fall aus miteinander verknüpften Teilen:
Zum einen die Klasse, die `org.eclipse.ui.part.ViewPart` erweitert, `RuleOutputView`, sodann ein `org.eclipse.jface.text.TextViewer`, der erstellt wird, sobald der `ViewPart` sichtbar wird. Teil eines `TextVierwer`' ist ein `org.eclipse.jface.text.Document`. Wird dieses Dokument verändert, so ändert sich auch die Anzeige im Workbench.
`TextViewer` ist eine Klasse, die in JFace (siehe Kapitel 4.2) vorgefertigt vorhanden ist. Einzig ist zu beachten, dass man bei der Veränderung des Dokumentes einen neuen Thread (`new Runnable()`) erzeugt, der eine Verzögerung des momentanen Thread', bis zur Aktualisierung der Ansicht, verhindert.
Dem View können zeilenweise Zeichenketten hinzugefügt werden. Es handelt sich hierbei um eine reine Ausgabe.
Wenn die Regel-Maschine Ausgaben macht, so sollen diese bei dem View angelangen. Dazu werden bei der Instanzierung der `Rete`-Instanz deren Ausgabe-Router auf die Regelausgabe umgeleitet. Dazu wird ein passiver Push-Filter, welcher `java.io.Writer` erweitert, erstellt. Dieser Writer beinhaltet eine Referenz auf RuleOutputView. Die Ausgabe der `Rete`-Instanz wird auf den Writer umgeleitet, und somit wird bei einem `write()` in der `Rete`-Instanz der Puffer des Writer gefüllt und bei einem `flush()` der Puffer in die Ausgabe geleert.

Das folgende Diagramm wird dieses Vorgehen verdeutlichen. Zu Gunsten der Lesbarkeit sind einige Details weggefallen und das Diagramm ist zu drehen.
Die Figur ‚command' (Strichmännchen) steht für alle Programmteile, die durchlaufen werden, wenn der Benutzer ‚(printout t „Hallo")' selektiert und dann die Anweisung ‚an jess senden' ausführt. Zur Vereinfachung ist der Vorgang der Selektion hier nicht mit aufgeführt. Wird etwas selektiert, so wird, wie in Kapitel 5.3: Selektionsverarbeitung beschrieben, ein Verweis auf die `Selection` in `JessSelection` gespeichert. Bei `run(IAction)` wird dann die letzte Selektion aufgelöst.

5.6 Laden des Laufzeitzustandes und von Initialdateien

Da beim Start von eclipse, respektive des Observer-Plugins, auch die Regel-Maschine neu instanziert wird, sind Fakten, die bei der vorherigen Session dem Arbeitsspeicher hinzugefügt wurden, verloren. Regeln und Funktionen müssen neu in die Regelbasis eingefügt werden.
Der Benutzer soll nun entscheiden können, ob der Laufzeitzustand der letzten Session bei der Initialisierung der Regel-Maschine geladen werden soll. Ähnlich verhält es sich, wenn der Benutzer das automatische Laden von selbst geschriebenen `*.clp`-Dateien zum Initialisierungszeitpunkt wünscht.

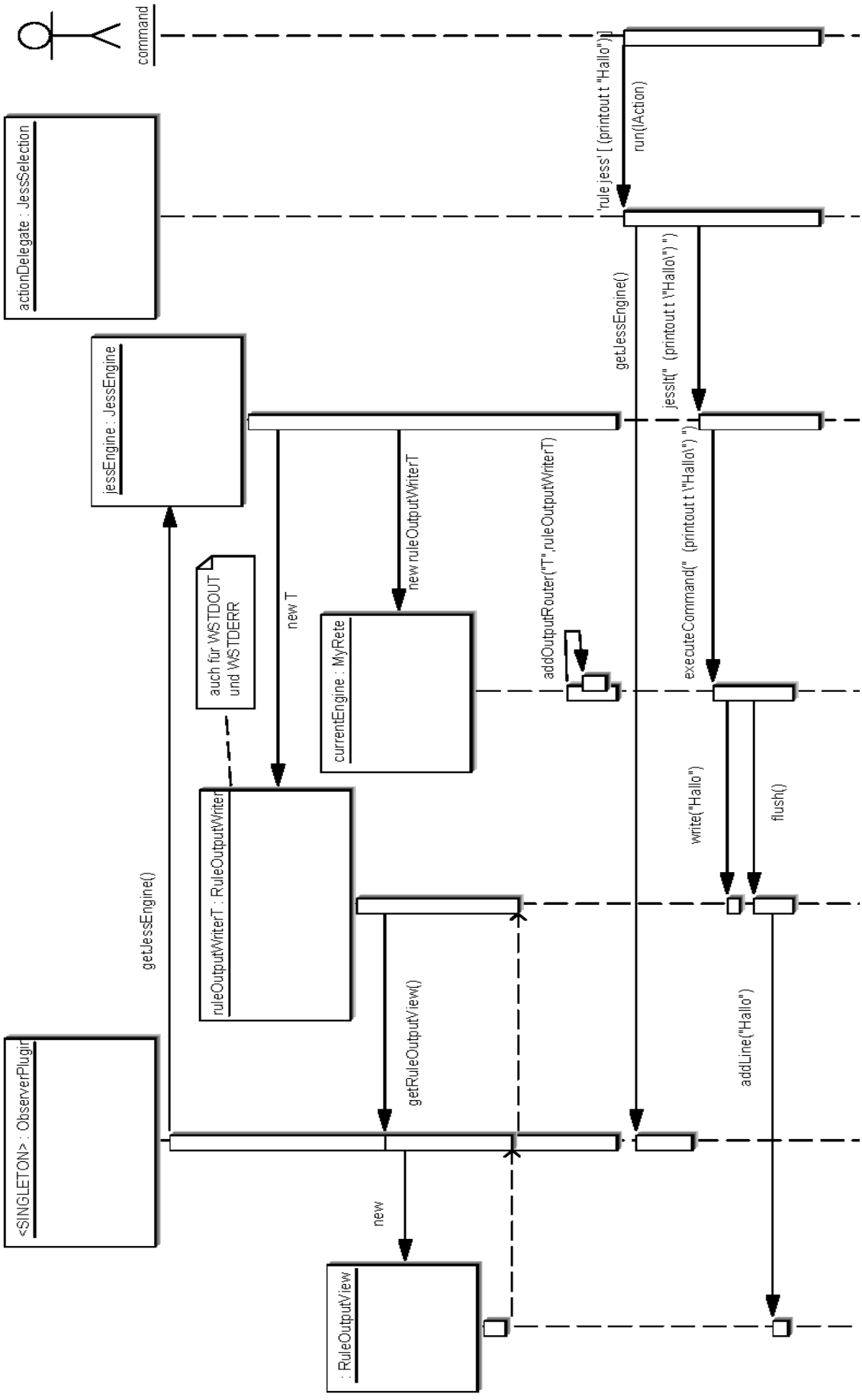

Abbildung 5.1: Ausführung von jess-Quellcode

Wie diese Einstellungen gespeichert werden, dazu mehr im folgenden Kapitel: Einstellungen.

Die automatische Ausführung von Initialdateien funktioniert analog zu der manuellen, wie in Selektionsverarbeitung (Kapitel 5.3) bereits beschrieben.

Für die Sicherung und das Wiederherstellen des Laufzeitzustandes wird die Klasse `RuntimeState` aus dem Paket `jessEngine` aufgerufen. Jess stellt hierfür zwei Methoden zur Verfügung. Für die Methode zum Sichern des Laufzeitzustandes wird ein passiver Push-Filter, und zum Laden des Laufzeitzustandes ein passiver Pull-Filter benötigt.
In die Sicherung und das Wiederherstellen sind die Ausgabe-Router, wie in Regel-Ausgabe beschrieben, sowie eventuell eingetragene Listener nicht mit eingeschlossen.
Als Ort für die Speicherung des Zustandes wird von der Laufzeitumgebung ein für das Plugin reservierter Ort erfragt. Es handelt sich hier nicht um eine Datei, bestehend aus jess-Befehlen, die editierbar wären, sondern um eine Binärdatei, so dass der Speicherort dem Benutzer nicht zugänglich gemacht werden braucht. Die Speicherung findet beim Schließen von eclipse statt.

5.7 Einstellungen

Damit Initialisierungsdateien geladen werden, muss der Benutzer einstellen können, *welche* als solche zu betrachten sind. Auch muss einstellbar sein, ob der Laufzeitzustand zum Initialisierungszeitpunkt zu laden ist.
Zuerst muss man sich vor der Implementierung Gedanken darüber machen, wie solch ein Auswahldialog aussehen soll. Die Methode `createContents(Composite parent)` wird automatisch aufgerufen, wenn es darum geht, die Anzeige zu erstellen. In ihr wird das Design der Anzeige festgelegt. In diesem Fall handelt es sich um eine Tabelle und fünf Tasten.
Dann werden so genannte Provider erstellt, welche den Inhalt des Auswahlfensters zur Verfügung zu stellen haben. Alle zu sehenden Tasten müssen mit Funktionalität versehen werden. Die Taste: ‚alle' sorgt zum Beispiel dafür, dass alle Einträge der Tabelle selektiert werden. Tasten können auch benutzt werden, um weitere Dialogfenster zu öffnen. In dem folgenden Bild ist die geöffnete Anzeige zu sehen, mittels derer man vorhandene Initialisierungsdateien zu der Tabelle der Hauptanzeige hinzufügen kann.
Ich möchte hier nicht weiter auf die Details einer solchen Implementierung eingehen. *Ein* Punkt ist jedoch für mehrere Klassen des Plugins von Interesse.
Wenn die Tasten: ‚apply' oder ‚ok' gedrückt werden, wird die Methode `performOk()`aufgerufen. Nun geht es darum, die bisher gesammelten Informationen persistent zu machen. Wichtig ist nun auch, dass andere Klassen dieser Änderungen, so der allgemeine Fall, gewahr werden. Welche Dateien und ob der Laufzeitzustand ausgesucht wurden, ist in einer Liste von speziell dafür erstellten Objekten gespeichert. Dieses ist dafür gedacht, um problemlos Objekte hinzufügen oder entfernen zu können. Eigenschaften werden in einem PreferenceStore gespeichert. Es handelt sich hier um Schlüssel-Wert Paare. Den Ort der Speicherung bekommt man durch die Plugin-Klasse

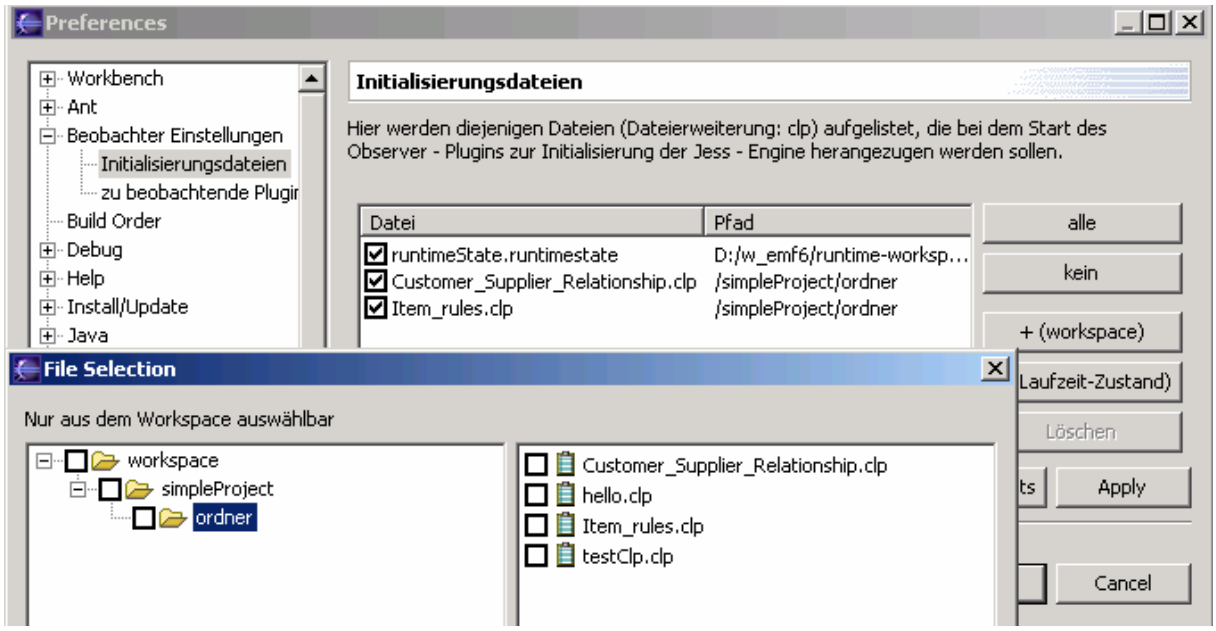

Abbildung 5.2: Einstellungen Initialisierungsdateien

`ObserverPlugin`: `ObserverPlugin.getDefault().getPreferenceStore()`.
Auf diesen Speicherplatz kann nicht nur von allen Plugins aus zugegriffen werden, das Plugin, hier die Klasse `ObserverPlugin`, kann auch EventHandler eintragen, um über Änderungen informiert zu werden. Für den Fall der Initialisierungsdateien macht dieses Vorgehen zwar keinen Sinn, um so mehr aber später bei anderen Einstellmöglichkeiten, wie sie im zweiten Teil beschrieben werden. Zur Speicherung wird eine Zeichenkette erstellt. Es wird eine inverse Methode zur Wiederbeschaffung der Dateien aus dem Format der Zeichenkette öffentlich zur Verfügung gestellt. So können andere Klassen, in diesem Fall die Klasse `JessEngine`, dieses Format der Speicherung lesen. Dafür wird von `ObserverPlugin` der PreferenceStore verlangt. Dann wird mit Hilfe des Schlüssels (ist selbst auch eine öffentliche Zeichenkette der Klasse), welcher die Einstellungen repräsentiert, der Wert verlangt. Dieser Wert wird anschließend an die Methode zur Gewinnung einer Liste von Dateien übergeben.

5.8 Hilfe für Benutzer

Eine Hilfe wird erst später in das Plugin integriert. Es ist jedoch an dieser Stelle schon darauf hinzuweisen, dass einige Hilfeseiten für diesen Teil des Plugins notwendig sind. Zu Beginn der Beschreibung dieses Plugins ist zum Beispiel erwähnt, auf welche Weise Quellcode ausgeführt werde kann. Diese Beschreibung stammt aus der Hilfe zu diesem Teil des Plugins, auch wenn die Hilfe anders gegliedert ist als dieser Bericht. Die Hilfe unterteilt sich in drei Teile: ein Teil für die Benutzung des Plugins, ein anderer für Entwickler, welche genauer wissen möchten, wie das Plugin funktioniert und welche Gedanken zur Erweiterung des Plugins gemacht worden sind. Dann noch ein Teil mit allgemeinen Informationen.

Kapitel 6

Observation anderer Plugins

Um andere Plugins funktional erweitern zu können, müssen Informationen über deren Zustand empfangen werden können. Es geht hier nicht darum, eine Zustandsdefinitionssprache zu erstellen, sondern darum, Zustandsänderungen gewahr zu werden.
Es gibt in eclipse keine Instanz, die Kontrolle über Veränderungen an den Plugins, Resourcen und Laufzeitobjekten hat. Es wird kein allgemeiner Zustand definiert, somit ist man bei der Beobachtung auf das Sammeln von Zustandsänderungen angewiesen.
Eclipse ist eine auf Komponenten basierende Struktur zugrunde gelegt, und ein Großteil der Kommunikation zwischen den Plugins ist ereignisbasiert. Insbesondere ist ein Teilprojekt von eclipse ins Auge gefallen. Das Eclipse Modeling Framework hat das Konzept einer ereignisbasierten Struktur konsequent durchgesetzt.
Ich habe mich entschieden, meine Untersuchungen bezüglich der Anbindung an andere Plugins mit einer Anbindung an EMF generierte Plugins zu beginnen. Später wird beschrieben, wie eine solche Anbindung im Allgemeinen erreicht werden kann.
Interessant ist für diese Untersuchung, wie man auch Plugins anderer Autoren überwachen kann. Die Überwachung eigener Plugins ist vollkommen problemlos, man kann einfach bei jeglicher Instanzierung eines Objektes einen EventHandler bei diesem Objekt oder bei einem zentralen EventManager eintragen.
Grundlage für das Verständnis der Vorgehensweise ist genaue Kenntnis über ereignisbasierte Systeme.

6.1 Ereignisbasierte Systeme

Die Observation von eclipse- (und EMF-) Elementen kann auf der Basis von ereignisbasierenden Architekturen, event-based architectures' genannt, geschehen.

Ereignisbasierende Systeme werde auch in den vorlesungsbegleitenden Folien zur Vorlesung ‚Software Architektures' beschrieben: [SWArch], S. I im vierten Kapitel.

Zum Aufzeigen der vier zu besetzenden Rollen folgen nun ‚Class Resposibility Cards':

Event	
Verantwortlichkeit	**Kollaboration**
•Kapselt Datails des Ereignisses •Klassifiziert das Ereignis	

EventManager	
Verantwortlichkeit	**Kollaboration**
•Entkoppelt EventSource von EventHandler •Synchronisiert Events •Koordiniert EventHandler	EventSource EventHandler Events

EventSource	
Verantwortlichkeit	**Kollaboration**
•Sammelt Event-Details •Liefert Events aus	EventManager Event

EventHandler	
Verantwortlichkeit	**Kollaboration**
•Zeigt Interesse an einer Klasse von Events •Nimmt Events entgegen	EventManager Event

Abbildung 6.1: Class Responsibility Cards: event-based systems

Zur Verwirrung beitragend wird in der Literatur, und auch hier in EMF, EventSource auch EventTarget genannt.
In Eclipse werden die Rollen EventSource und EventManager oft von einer Klasse verwirklicht.

Zur Veranschaulichung erst mal ein Beispiel, welches die Kommunikation zwischen eclipse Komponenten und eigenen Klassen aufzeigt. Auf dem Diagramm auf der nächsten Seite ist zu sehen, wie das Observer Plugin bei Änderungen an Resourcen informiert wird.

Die Darstellung im Diagramm (Abbildung 6.1) ist ein wenig vereinfacht. Der Benutzer (User) führt eine Aktion aus, die zur Speicherung einer Resource führt. Dieser Vorgang ist in dem Diagramm als ein Methodenaufruf dargestellt.
Das Laden des Plugins ist hier auch vereinfacht durch den Aufruf `new()` dargestellt.

Das Plugin registriert einen Listener / EventHandler bei dem EventManager mittels:
`addResourceChangeListener()`.
Sobald dann eine Aktion ausgeführt wird, die es erforderlich macht, die EventHandler zu informieren, hier `save()`, wird ein Event erzeugt. Eine Instanz von `IResourceChangeEvent` ist ein Event, das es ermöglicht, auch auf die veränderte Resource zuzugreifen.

Jetzt werden - Achtung: *in beliebiger Reihenfolge!* - alle eingetragenen Listener informiert und es wird ihnen das Event-Objekt übergeben. Von diesem Event können nun Informationen zu der Änderung erfragt werden.
Hier kann es zu Nebenläufigkeiten kommen. Dieses ist eines der Grundkonzepte der Eclipse-Architektur, und es ist ein nicht deterministischer Vorgang.

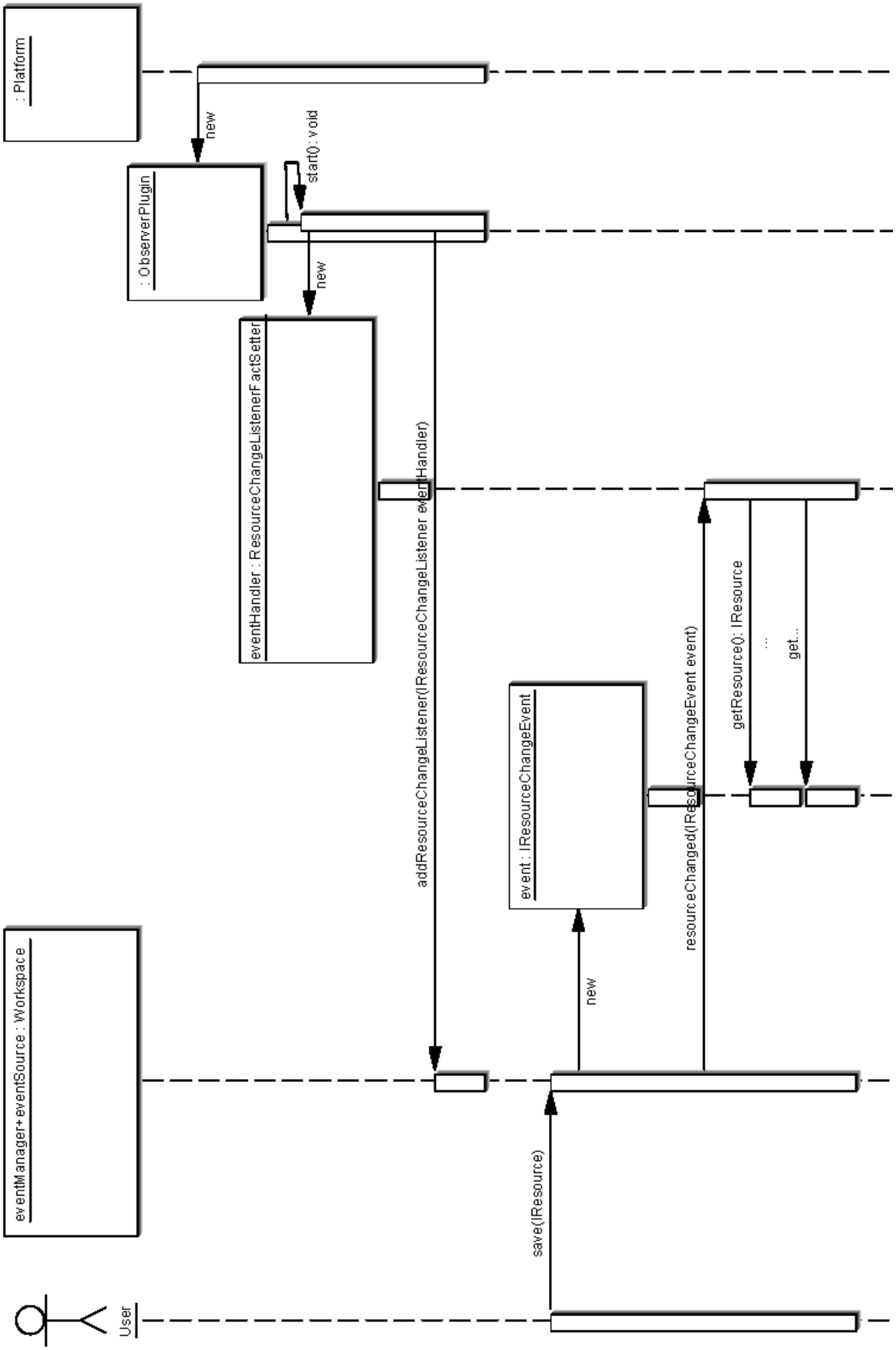

Abbildung 6.2: Änderungen an Resourcen

Kapitel 7

Eclipse Modeling Framework

Das Eclipse Modeling Framework, kurz EMF, ermöglicht es, aus bestehenden Modellen Klassen zu generieren.
Wird nun eine Objektstruktur modelliert, was man zum Beispiel mit einem grafischen Plugin für eclipse machen kann, besteht die Möglichkeit, diese Objektstruktur nicht nur in Java-Klassen abzubilden, sondern auch Klassen zu generieren, die es ermöglichen, passende Objekte in Editoren zu erstellen und persistent zu machen.
Zur Einarbeitung in EMF erstelle ich mittels UML ein Model von einem Bestellungssystem mit Bestellungen, Waren, Kunden und Zulieferern. EMF generiert mir nun Editoren und Dialoge, mit denen man Bestellungen erzeugen und verändern, Kunden zuweisen und Zulieferer mit Namen versehen kann. Alle diese Entitäten können auch als XML- Dateien gespeichert werden. Solch ein System kann innerhalb von einer halben Stunde erstellt werden.

Interessant für die Verwendung hier sind im wesentlichen vier Dinge:

1. Fast alle Objekte in EMF sind `Notifier`
2. `Adapter` sind dazu vorgesehen Funktionalität von EMF Objekten zu erweitern
3. Die Modellstruktur kann zur Laufzeit analysiert werden
4. Es besteht eine Reflective-API

7.1 Notifier

Teile einer Objektstruktur werden auf so genannte `EObjects` abgebildet. Ein `EObject` implementiert die Schnittstelle `Notifier`. Solch ein Notifier ist ein EventManager und eine EventSource, man vergleiche hierzu das Kapitel 6.1: Ereignisbasierte Systeme. Die Methode `eAdapters()` beinhaltet alle eingetragenen EventHandler. Diese EventHandler werden im EMF Kontext Adapter genannt. Wird nun ein Attribut dieses Objektes geändert, werden eingetragene Adapter, falls vorhanden, informiert. Interessant ist hierbei der Aspekt, dass die `EObjects` ja generiert worden sind, also auch für sämtliche Wertänderungen ‚von Haus aus' diesen Benachrichtigungsmechanismus unterstützen. Man kann jedoch auch nach Generierung in den Quellcode eingreifen, was auch an vielen Stellen erforderlich ist, und dadurch ist dann die Möglichkeit geschaffen, diese Benachrichtigungen zu unterdrücken. Ein Adapter kann natürlich nur dann über eine Änderung informiert werden, wenn eine Benachrichtigung über diese Änderung ausgeht. Ist es notwendig, diese Benachrichtigungen per Hand zu schreiben, wie es ja ansonsten der Fall ist, so muss vorausgesehen werden,

welche Nachrichten von Interesse sein könnten. Und das ist für den allgemeinen Fall der Observation, der hier betrachtet wird, nicht zutreffend.

7.2 Adapter

Ein Adapter kann bei einem `EObject` als EventHandler eingetragen werden, es findet also eine Observation des betreffenden Elementes statt. Adapter haben ihren Namen jedoch aus der Erkenntnis, dass EventHandler das Verhalten eines Objektes erweitern können, ohne eine Subklasse bilden zu müssen.
Auch bietet EMF an, einen `EContentAdapter` bei Wurzelobjekten zu registrieren, um über Änderungen an allen enthaltenen Objekten informiert zu werden.
Adapter werden üblicherweise durch `AdapterFactory`s erzeugt. Man vergleiche hierzu das Factory-Method Design Pattern. Die Begriffe Adapter und Adapterfabrik werden gleichbedeutend verwendet. Möchte man einen Adapter an ein `EObject` knüpfen, so kann man die Adapterfabrik dazu auffordern, einen bestimmten Adapter für das übergebene `EObject` zurückzuliefern. Das übergebene `EObject` wird hier mit EventTarget bezeichnet, was anfänglich sehr verwirrend erschien, jedoch ist es für die Adapterfabrik das Zielobjekt, an welches ein Adapter angeschlossen werden soll. Die Adapterfabrik überprüft das `EObject` nun nach vorhandenen Adaptern, und falls der gewünschte schon dabei ist, wird dieser und ansonsten ein neuer Adapter zurückgeliefert. Es ist auch möglich, Adapterfabriken miteinander zu kombinieren, so dass sich die `ComposedAdapterFactory` dann wie eine normale Adapterfabrik verhält. Zum Verständnis beitragend, möchte ich auf das Composite Design Pattern verweisen.

7.3 Metamodel

Die Objektstruktur, wie sie modelliert wurde, ist das Metamodell, und deren Abbildung in Quellcode kann zur Laufzeit analysiert werden. Es gibt Attribute und Referenzen und dazu Methoden, um sich all jene auflisten zu lassen. Man kann herausfinden, welche Klassen in welchen enthalten sind und so die gesamte Struktur des Modells und deren Eigenschaften nachsuchen. Jedes Attribut einer Klasse hat zum Beispiel eine `FeatureId`, so kann man bei Änderungen an Klassen-Instanzen nachvollziehen, welches Feature, sprich Attribut, sich geändert hat.

7.4 Reflective API

Man kann auch zur Laufzeit Objekte erzeuge oder verändern, ohne zu wissen, von welcher Klasse diese Objekte stammen. Generische Methoden wie `eGet()` und `eSet()` ermöglichen es, unter Übergabe der `FeatureId` Werte zu ändern, ohne die konkreten Methoden zur Änderung zu kennen.
Beispeisweise sind mit einem Attribut wie: `name`, entsprechende Methoden verbunden: `getName()` und `setName()`. Das Attribut Name ist nun mit einer `FeatureId` versehen. Ruft man `eGet()` mit der entsprechenden `FeatureId` auf, so wird der Aufruf an `getName()` weitergeleitet.
Die Verwendung ist auch in dem Kapitel 11: Schlussbetrachtung zu sehen.

Kapitel 8

Anbindung an EMF-Plugins

Nachdem die Interesse weckenden Eigenschaften von EMF generierten Plugins untersucht worden sind, stellt sich nunmehr folgende Aufgabe. Es muss versucht werden, einen Verweis auf die Laufzeitobjekte, welche verändert werden, zu bekommen. Dann kann ein EventHandler eingetragen werden, und dieser EventHandler kann sodann einen Eintrag in die Regel-Maschine in Form eines Faktes vornehmen.
Nach mehreren missglückten Ansätzen wird die Anbindung gelingen. Ein bisher noch nicht betrachtete Eigenschaft von EMF-Plugins wird die Observation ermöglichen.

8.1 Fehlgeschlagene Ansätze

Zunächst einmal wird betrachtet, wie Adapter in EMF benutzt werden. Man denke hierbei daran, dass die Grundidee ist, einen bestimmten Typ von Klassen beobachten zu können. Dazu soll das Model durchsucht werden, und dann soll für eine *Klasse* des Modells (also nicht für eine Instanz) festgelegt werden, dass man sie beobachten möchte. Das wäre zumindest optimal, wenn das so gelingen könnte.
Um genau abzuklären, ob es nicht möglich wäre, generell an eine Klasse zu adaptieren, anstelle von einzelnen Instanzen, wende ich mich mit einer entsprechenden Frage an die News-Group zu EMF. Ed Merks, einer der Autoren des Buches [eMF], S. I, beantwortet dort Fragen zu EMF.
Vorläufig werden erst mal einige Versuche gemacht, um mit der Vorgehensweise von Adaptern in EMF vertraut zu werden.
Für das Beobachten von Klassen ist in EMF ein Mechanismus vorgesehen: Jedes Objekt in EMF ist ein `EObject`, welches die Schnittstelle `Notifier` implementiert. Man kann also bei jedem `EObject` in seiner Liste der Adapter einen solchen eintragen
[`eObject.eAdapters().add(myAdapter)`].

Siehe dazu in [eMF] Kapitel 13 (besonders AdapterFactory...), S. I.

Für ein Modell einer Bestellung mit Gegenständen und Adressen... funktioniert es, eine Klasse zu adaptieren, wenn man sie selbst erstellt.

```
PurchaseOrder order = ExtendedPOFactory.eINSTANCE.createPurchaseOrder();
ChangeCounterAdapterFactory.INSTANCE.adapt(order, ChangeCoun-
terAdapter.class);
USAddress a = ExtendedPOFactory.eINSTANCE.createUSAddress();
order.setBillTo(a);
PurchaseOrder order2 = ExtendedPOFactory.eINSTANCE.createPurchaseOrder();
order2.setBillTo(a);
```

Der Adapter (vom Typ `ChangeCounterAdapter`) wird hier zweimal von einer Änderung informiert, auch wenn es sich um zwei verschiedene Instanzen der Klasse `PurchaseOrder` handelt. Wenn man die zweite Reihe betrachtet, kann man sehen, dass die Adapterfabrik aufgefordert wird, an das übergebene Objekt (`order`) einen Adapter vom Typ `ChangeCounterAdapter` anzuhängen.
Wenn jetzt andere Plugins `PurchaseOrder` Klassen instanzieren, so bekommt man leider keine Nachrichten. Eine Adaption in der Umgebung des eigenen Plugins an Instanzen eines Typs reichen leider nicht aus, um über alle Änderungen an Instanzen des selben Typs informiert zu werden.

Es stellt sich dann die Frage, ob nicht in einer der generierten Klassen so etwas wie eine Registrierung geschieht, oder zumindest, wie im obigen Beispiel, dass eine Klasse alle Instanzierungen verwaltet.

Plugins als EMF-Plugins zu identifizieren, ist einfach. EMF-Plugins erben alle von `EclipsePlugin`. Alle Plugins müssen sich, wie schon in dem Kapitel 5.2: plugin.xml beschrieben, der Laufzeitumgebung bekannt machen. Deswegen kann zu Beginn eine Pluginregistrierung erstellt werden, die dann auch durchsucht werden kann.

```
IPluginRegistry registry = Platform.getPluginRegistry();
IPluginDescriptor descriptors[] = registry.getPluginDescriptors();
for...
if((descriptors[count].getPlugin().getClass().getSuperclass()
==EclipsePlugin.class))
```

Hat man das Plugin seiner Wahl, kann eine Untersuchung des zugrunde liegenden Modells vorgenommen werden. Doch auch in diesem Fall sind keine Möglichkeiten zu entdecken, Adapter anbringen zu können. Dennoch wird das Wissen um die Pluginregistrierung noch von Nutzen sein.
Bisher wurde der Fall betrachtet, dass man generell observieren möchte, ohne darauf zu achten, wann und wie solche Klassen überhaupt benutzt werden. Eine solche generelle Überwachung ist jedoch, so schreibt auch Ed Merks in der News-Group, nicht vorgesehen.

Ein weiterer Ansatz ist, diejenige Resourcen zu betrachten, aus welchen Objekte erstellt werden. EMF bietet, wie schon kurz erwähnt, die Möglichkeit, Laufzeitobjekte in Dateien zu speichern. Betrachtet man die Vorgehensweise, wie man Objektkonstrukte mittels Editoren erstellt und manipuliert, so sieht man, dass zuerst eine Resource eines bestimmten Typs erstellt wird, in die dann Objekte gespeichert werden. EMF ermöglicht es, aus Resourcen leicht zu lesen und sich Objekte laden zu lassen. Ich gehe also der Frage nach, ob man sich nicht, wie diese EMF-Editoren, registrieren lassen kann, um über die Öffnung einer solchen Resource informiert zu werden. Man kann schon benachrichtigt werden. Es entsteht jedoch daraus kein Nutzen. Wird der Observer vor dem zu observierenden Editor benachrichtigt, hat man lediglich eine Resource vor sich, die Objekte in serialisierter Form in sich birgt. Die eigentlich zu observierenden Objekte existieren noch gar nicht. Wird die Resource dem Observer übergeben, nachdem der EMF-Editor sie geöffnet hat, kann man leider nicht von der

Resource zu den instanzierten Objekten gelangen. Über die Reihenfolge, in der die Plugins angesprochen werden, kann man auch keine Aussagen machen. Dieser Vorgang ist nicht deterministisch.

Die Überwachung von Plugins fremder Autoren, ohne Eingreifen in deren Quellcode, ist also auf die soeben beschriebenen Wege nicht erreichbar. Es existiert keine Instanz, welche den Zugriff auf alle Instanzen eines Typs gewährleistet und bei der man sich als Informationsempfänger eintragen lassen kann. Noch gibt es die Möglichkeit, von Seiten der Resource eine Überwachung zu starten. Zu den Eigenschaften einer Resource gehört es nicht, diejenigen Klassen zu kennen, welche aus ihr lesen.

8.2 EMF-Editoren und ItemProvider

Im folgenden bin ich der Frage nachgegangen, wie ein von EMF generiertes Plugin eine Darstellung der Objekte erreicht und es ermöglicht, Änderungen an diesen Objekten durchzuführen.
Prinzipiell können nämlich 3 Arten von Plugins von EMF generiert werden: Die Klassen zur Repräsentation des Modells werden in dem gleichen Projekt wie das sogenannte ECORE-Modell, das Metamodell, erzeugt.
Dann wird ein Projekt generiert, welches die GUI – unabhängigen Teile des Editors birgt. Es wird ein neues Projekt mit dem Namen: `projektnameDesModells.edit` angelegt, diesbezüglich wird, in Anlehnung an die Pakethierarchie in EMF, von EMF.edit gesprochen.
Die Ansichten (View) und Editoren für eclipse werden in einem 3. Projekt erzeugt. Die Bezeichnung ist analog: `projektnameDesModells.editor` / EMF.editor.

Üblicherweise funktionieren die Editoren mittels Content- und LabelProvider (vergleiche hierzu [JDGtoE], S. I) bei denen man Listener, also EventHandler, eintragen kann, um über Änderungen informiert zu werden. Geschieht eine Änderung an dem Dokument, welches angezeigt wird, so wird die Darstellung der Ansicht aktualisiert. Da verschiedene Ansichten eines Objektes zu synchronisieren sind, muss für den Eintrag von EventHandlern bei den Content- und LabelProvidern, welche die notwendigen Informationen für die Ansichten bereitstellen, gesorgt werden.
EMF hat ja auch schon die Fähigkeiten der Adaptierbarkeit auf der Ebene der einzelnen Objekte der Modellstruktur. Diese beiden Mechanismen vereint ein sogenannter ItemProvider aus EMF.edit. Dieser ItemProvider ist selbst ein Adapter (siehe hierzu [eMF], S. I in Kapitel 3.2).

Betrachtet man nun jene Klasse, die für die Instanzierung des Editors zuständig ist, so wird man hier die Verwendung der ItemProvider finden.

Bei Resourcen mit entsprechender Endung (siehe `plugin.xml` des Editor Projektes) werden Plugins geladen. Bei Bedarf wird dann ein Editor instanziert und folgende ItemProvider eingetragen.

```
// Create an adapter factory that yields item providers.
//
List factories = new ArrayList();
factories.add(new ResourceItemProviderAdapterFactory());
```

```
factories.add(new ExtendedPOItemProviderAdapterFactory());

adapterFactory = new ComposedAdapterFactory(factories);
// Create the command stack that will notify this editor as commands are
executed.
BasicCommandStack commandStack = new BasicCommandStack();
...
// Create the editing domain with a special command stack.
editingDomain = new AdapterFactoryEditingDomain(adapterFactory, command-
Stack);
```

Ungleich den oben angezeigten Beispielen ist es nicht sofort offensichtlich, wann und wo eine `adapt()`-Methode aufgerufen wird, wann also Adapter zu instanzierten Objekten erzeugt und diese dann eingetragen werden. Dies geschieht immer dann, wenn eine assoziierte Resource geöffnet wird. Prinzipiell ist in diesem Zusammenhang eine Adapterfabrik wie ein Adapter zu betrachten. Hier ist eine Fabrik für ItemProvider vorhanden.
`ResourceItemProviderAdapterFactory` ist eine Klasse aus dem EMF-Framework und nicht speziell für dieses Plugin generiert.
Zum Zeitpunkt des Ladens aus einer Resource werden mittels der ItemProviderAdapterFactory ItemProvider zu den neu instanzierten Objekten erstellt.
ItemProvider werden darüber informiert, wenn Änderungen an adaptierten Objekten stattfinden.

Uninteressante Events werden herausgefiltert und die anderen werden dann weitergeleitet.

Die Fähigkeit der Weiterleitung ist auch daraus entstanden, dass die Präsentation der Daten aufgeteilt ist. Zum einen wird ein GUI unabhängiger Teil generiert und dann der, welcher für die Präsentation in eclipse zuständig ist.
ItemProvider implementieren die Schnittstelle `IChangeNotifier`, welche es Views und anderen interessierten Parteien erlaubt, sich als Listener zu registrieren.

Zur Klärung der Begrifflichkeiten ist hier noch anzumerken, dass Listener EventHandler sind. Ein Event wird hier in diesem Zusammenhang als Notification gehandhabt. Ein EventSource wird auch, nach der implementierten Schnittstelle, Notifier genannt.
Die Label-, bzw. ContentProvider der Editoren (vergleiche hierzu [JDGtoE], S. I) tragen sich selbst bei den Adaptern als Listener ein, um auf Änderungen an den Instanzen reagieren zu können
[`setLabelProvider(new AdapterFactoryLabelProvider (adapterFactory))`].
Man kann sich bei diesem ItemProvider (nicht zu verwechseln mit den Label-, beziehungsweise ContentProvidern) als Listener eintragen und wird nun ebenfalls von diesem informiert, wann immer ein enthaltener Adapter informiert wird. Der entsprechende Adapter wird in einer lokalen Variable gespeichert.

Diese ItemProviderAdapterFactory ist also die Schnittstelle zu dem Observer-Plugin.

Genauer gesagt stellt also die ItemProviderAdapterFactory seinerseits, zusätzlich zu seiner Eigenschaft als EventHandler, auch Methoden in seiner Eigenschaft als EventManager zur Verfügung, um dann alle Listener, bei Änderungen an den verknüpften EObjects, informieren zu können.
Um das Ganze nicht zusätzlich zu komplizieren, möchte ich nicht weiter auf die genaue Struktur der Adapter/AdapterFactory/ComposedAdapterFactory eingehen (siehe dazu auch Composite-Pattern).

Kapitel 9

Automatische Generierung von Fakten

Nachdem die Funktionalität des ersten Teils dieses Plugins sichergestellt ist (vergleiche hierzu Kapitel 5) ist der Teil einzufügen, der die automatische Generierung von Fakten, aufgrund von Änderungen in eclipse, birgt. Dazu ist es nun notwendig, die Struktur des Plugins genauer zu betrachten, um den Überblick über die beteiligten Klassen nicht zu verlieren. Sollen Fakten erzeugt werden, so sind zwei Abläufe zu implementieren. Zum einen muss die zu überwachende Instanz identifiziert und eine Überwachung eingeleitet werden. Zum anderen ist ein Fakt zu erzeugen und in die Regel-Maschine zu integrieren.

Folgende Aktivitäten sind für diesen Teil durchzuführen. Die Punktweise Erklärung zur Erweiterung des Plugins gehört nur insofern hierher, als dass der Aufbau und die Vorgehensweisen des Plugins nochmals verdeutlicht werden.

1. Architektur des Observer Plugins
2. Auffinden der zu überwachenden Einheiten
3. Erstellen eines Faktes
4. Fakten nutzbar für benutzerdefinierte Regeln machen
5. Erweiterung der Einstellungen
6. Eignung der Editoren
7. Erweiterung der Hilfe
8. Punktweise Erklärung zur Erweiterung der Observation

9.1 Architektur des Observer Plugins

Zur Übersicht sei hier ein Diagramm (Abbildung 9.1) gegeben. Es soll die Aufteilung der Komponenten und deren Interaktion verdeutlichen.
Das Paket `jessEngine` beherbergt solche Dateien, welche für die Integration von jess in das Plugin von Bedeutung sind. Insbesondere *die* Klasse, welche die `Rete`-Instanz birgt und *jene*, welche für die Ausführung von jess-Quellcode verantwortlich ist. Zudem sind in dem Unterpaket `jessEngine.factSetter` Informationen enthalten, die für die Anbindung von automatisch generierten Fakten Vorraussetzung sind.
Das Paket `observer` enthält all jene Klassen, die für den Ablauf des Plugins zuständig sind. In ihm wird die Organisation über die zu ladenden Klassen und über die Resourcen durchgeführt.

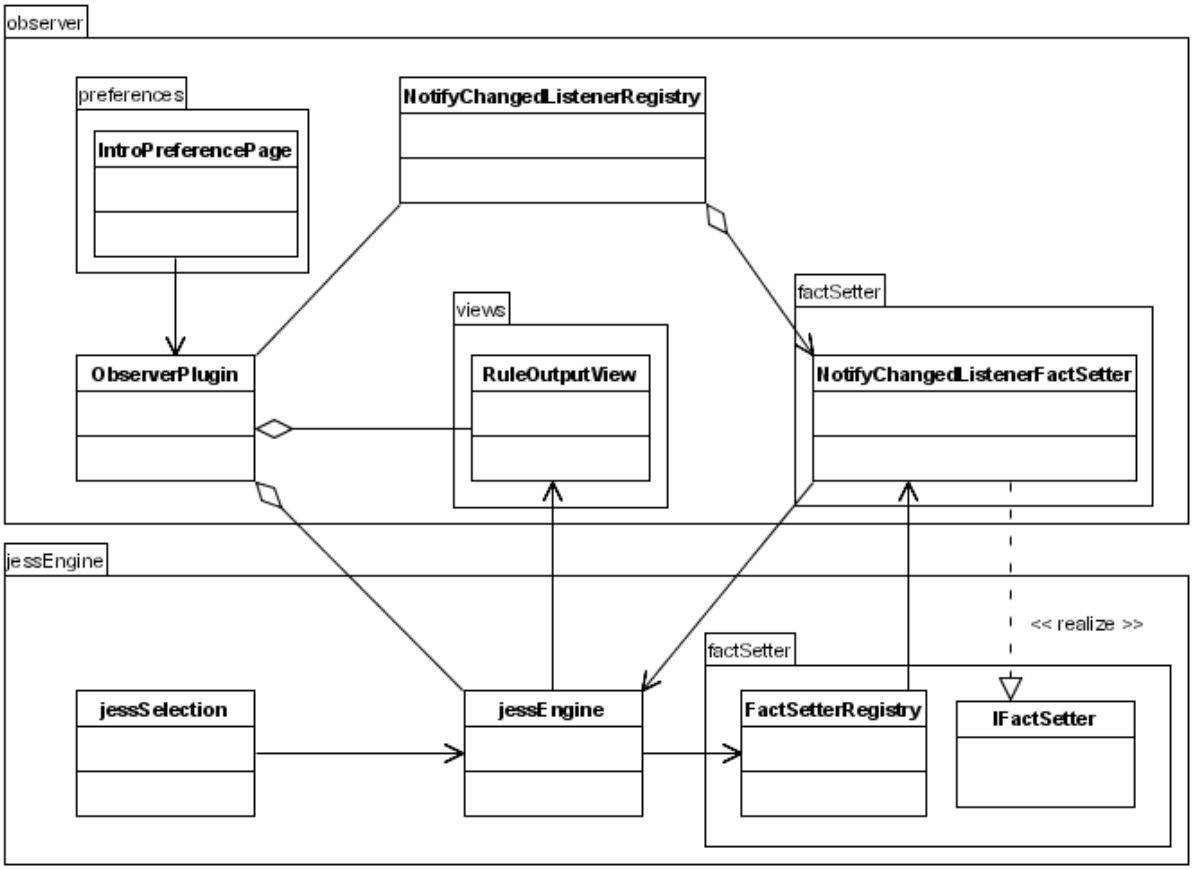

Abbildung 9.1: Architektur des Observer-Plugins

9.2 Auffinden der zu überwachenden Einheiten

Das Observer-Plugin trägt sich bei EMF-Editoren als Listener ein. Dafür verantwortlich ist die Klasse `NotifyChangedListenerRegistry`. Zur Verwaltung von den Editoren-Listener-Paaren werden zwei `Collection` verwendet. Die eine `Collection` enthält die Namen der Editoren, die observiert werden sollen. Diese Liste entspricht den Einträgen in den

vom Benutzer verwalteten Einstellungen (Änderungen darin werden durch einen Listener: `IPropertyChangeListener` in der Klasse `ObserverPlugin` berücksichtigt). Die zweite `Collection` beinhaltet die Listener-ItemProvider-Paare und diese werden je nach Aktivierung/Schließen des Editors (hierzu wird ein `IPartListener` verwendet) ein- bzw. ausgetragen. In diese Liste kommen die ItemProvider nur, wenn alle notwendigen Voraussetzungen geprüft wurden.
Um an Referenzen zu den zu observierenden Editoren zu gelangen, wird in dem sichtbaren Fenster nach Teilansichten gesucht. Es findet also eine Suche von ‚grob nach fein' statt. Als erstes wird das aktuell aktive Fenster (`IWorkbenchWindow`) betrachtet. Man bekommt eine Referenz darauf von dem Workbench, der Workbench ist der Plugin-Klasse `ObserverPlugin` bekannt. Sollte kein solches Fenster vorhanden sein, so wird ein EventHandler erstellt, welcher die Suche zu dem Zeitpunkt fortsetzt, an dem ein Fenster aktiviert wird. Daraufhin werden alle Teile dieses Fensters (`IWorkbenchPage`) durchsucht. Auch hier wird ein EventHandler eingetragen, falls sich noch andere `Parts` öffnen. Von den `Parts` kann man zu den beinhalteten Editoren und Ansichten (View) gelangen. Als erstes wird der aktive Editor, die aktive Ansicht, betrachtet. Anschließend wird wieder ein EventHandler eingetragen, der auf das Aktivieren und auf das Schließen eines Editors reagiert. Die so bekommenen Editoren werden zur Prüfung der Eignung weitergeleitet.
In dem Kapitel 9.6: Eignung der Editoren werden die Eignungsbedingungen genauer erläutert. Die ItemProvider können von den Editoren gewonnen werden und werden dann, entsprechend den betrachteten Editoren, in die `Collection` der Listener-ItemProvider-Paare mit einer neuen Instanz der Klasse `NotifyChangedListenerFactSetter` hinzugefügt. Alle Eintrage der `Collection` werden durchgegangen, ob die diesbezüglichen Benutzereinstellungen besagen, dass die Editoren überwacht werden sollen.
Abschließend wird eine Instanz von `NotifyChangedListenerFactSetter`, aus dem Paket `observer.factSetter` (falls für diesen Editor gewünscht) als Listener eingetragen. Diese Klasse implementiert die dazu notwendige Schnittstelle: `INotifyChangedListener`. Außerdem handelt es sich bei dieser Klasse um einen FactSetter. Diese Klasse erzeugt nämlich einen Fakt in der Regel-Maschine, wenn die Klasse über eine Änderung informiert wird.

9.3 Erstellen eines Faktes

Anknüpfend an den vorangegangenen Artikel, zeigt das Diagramm (Abbildung 9.2) den Ablauf, wie er aussehen könnte, wenn eine Änderung an den observierten Elementen auftritt.
Als erstes wird von der Erstellung eines EventHandler' ausgegangen. Ein `NotifyChangedListenerFactSetter` übernimmt diese Aufgabe. Wie der Name schon sagt, handelt es sich hier um einen FactSetter. Dieser FactSetter implementiert die Schnittstelle `IFactSetter` und ist somit für die Erstellung eines Faktes mit einem bestimmten einzigartigen Namen zuständig. Zur weiteren Beschreibung der Vorgehensweise ist es notwendig, erst einmal zu erläutern, warum man nicht so einfach einen Fakt hinzufügen kann.

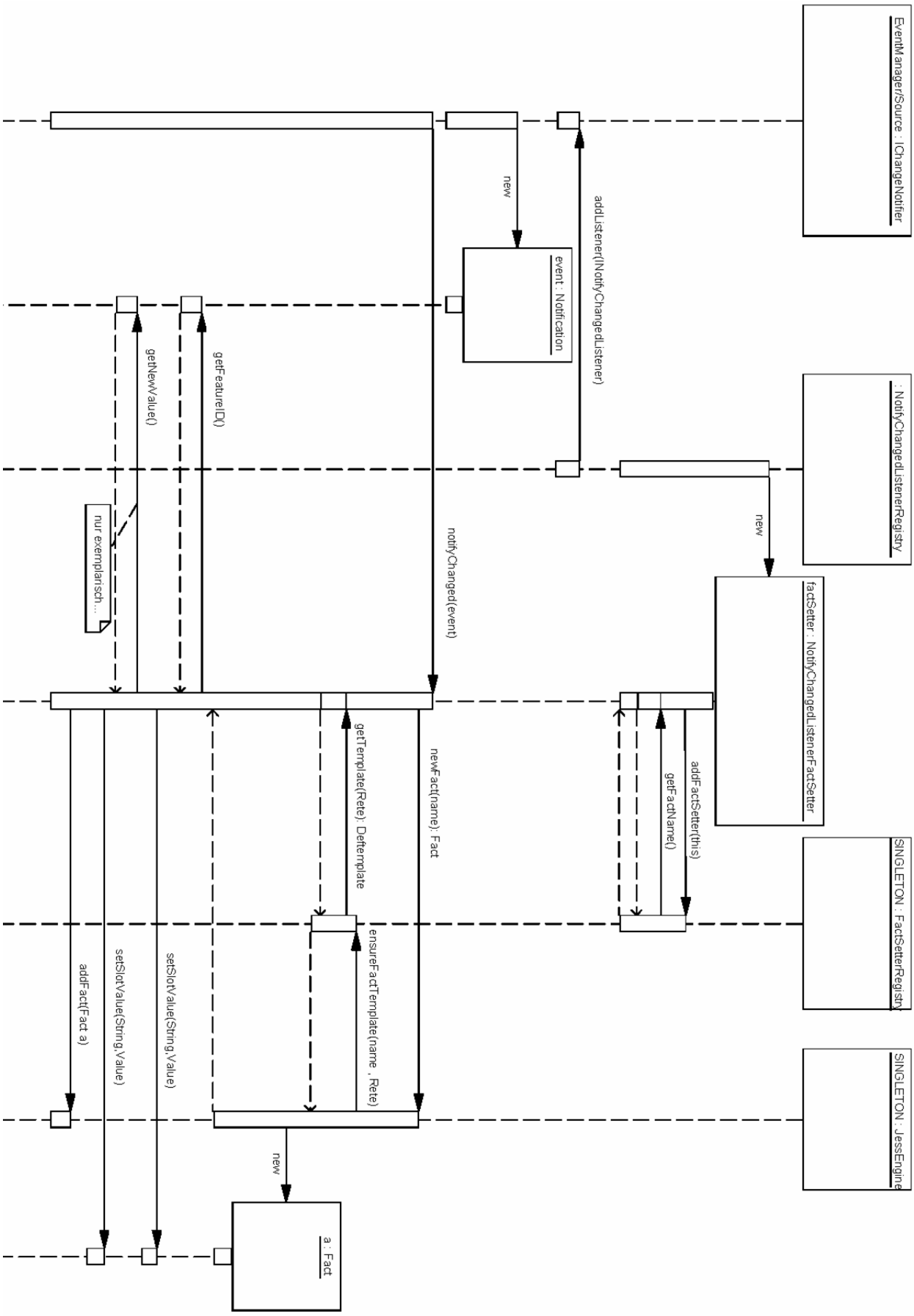

Abbildung 9.2: Erstellen eines Faktes

9.3.1 Aus jess - Sicht:

Man kann einen neuen Fakt [`new Fact(String factName, Rete currentEngine)`] nur erstellen, falls ein entsprechendes ‚Template' (`jess.Deftemplate`) mit gleichem Namen bereits existiert. Es handelt sich hier um so genannte unordered facts.
Ein `Deftemplate` [`new Deftemplate(String factName, String beschreibung, Rete currentEngine)`] kann mehrere slotValues, ähnlich einer Relationalen-Datenbank-Tabelle, haben [`template.addSlot(String slotName, jess.Value initialWert, String typDesInhaltes)`].
Der Typ des Inhaltes muss einer Zeichenkette wie 'STRING', 'INTEGER', 'OBJECT'...entsprechen. [`currentEngine.addDeftemplate(template)`] fügt dann das entsprechende `Deftemplate` zu der aktuellen `Rete`-Instanz hinzu. Wird nun ein `Fact` erstellt, müssen seine 'Spalten' natürlich zu denen des `Deftemplate` passen. Hinzugefügt werden Spalteneinträge mittels: [`fact.setSlotValue(String name, jess.Value irgendEinValue)`]. Man möge beachten, dass der `jess.Value` ein Objekt des gleichen Typs enthalten muss, wie in dem Template vorgegeben, z.B.: [`new Value(irgendEineZeichenkette, RU.STRING)`].
Mittels [`currentEngine.assertFact(fact)`] wird ein `Fact` der `Rete`-Instanz hinzugefügt.

9.3.2 Aus Observer-Plugin Sicht:

In diese Vorgehensweisen sind in dem Observer-Plugin hauptsächlich drei Klassentypen und eine Schnittstelle involviert: die Klasse `FactSetterRegistry`, die Klasse `JessEngine` und ein FactSetter, welcher `IFactSetter` implementiert. Es wird von einem FactSetter gesprochen, wenn die entsprechende Klasse die Schnittstelle `IFactSetter` implementiert, auch wenn es keine Klasse mit dem Namen FactSetter gibt.
Die Klasse `FactSetterRegistry` beinhaltet eine Liste von FactSetter. Ein FactSetter hat einen Namen für den Fakt, den es hinzufügen soll [`getFactName()`] und ist in der Lage, ein `Deftemplate` zu erzeugen. Die gesamte Struktur eines `jess.Fact` ist in der FactSetter - Implementierung gespeichert. Wenn ein neuer Fact erstellt werden soll (ein FactSetter wird informiert), so holt sich dieser von der einzigen Instanz der Klasse `JessEngine` einen Fact mit dem von ihm selbst definierten Namen. Die JessEngine lässt überprüfen, ob der Fact mit dem Namen in der `Rete`-Instanz vorhanden ist. Dies Überprüfung nimmt die `FactSetterRegistry` vor, die ja alle FactSetter verwaltet. Sollte ein solches `Deftemplate` nicht vorhanden sein, so kann die `FactSetterRegistry` ein `Deftemplate` von einem entsprechenden FactSetter erstellen lassen [`getTemplate(Rete currentEngine)`]. Die `JessEngine` - Instanz hält die `Rete`-Instanz und wird deshalb in diesen Vorgang involviert. Der FactSetter füllt nun, mit der nur ihm bekannten Struktur, den `jess.Fact` und lässt diesen dann von `JessEngine` der `Rete`-Instanz hinzufügen [`jessEngine.addFact()`].
Dies Art der Verteilung der Aufgaben ist so gewählt, dass man die Struktur eines automatisch generierten Faktes im Quellcode leicht anpassen kann, indem man nur in einer Klasse, nämlich dem FactSetter, Änderungen vornimmt. Selbstverständlich kann das auch Auswirkungen auf bereits in jess geschriebene Regeln haben, und es könnte auch eine Anpassung der Hilfe-Dateien notwendig sein. Die `FactSetterRegistry` ist auch noch aus einem weiteren Grund vorhanden, der in dem nächsten Kapitel erläutert wird.

Analog werden auch Fakten für den Fall automatisch erstellt, wenn Resourcen im Workspace geändert werden. Die Vorgehensweise ist jedoch näherer Betrachtung nicht Wert. Das Setzen eines Faktes funktioniert analog zu dem bei dem eben beschriebenen Vorgang. Ein Listener kann einfach bei dem Workspace eingetragen werden, auf den die Plugin-Klasse `ObserverPlugin` einen Verweis bereitstellt. Das Eintragen eines solchen EventHandlers ist auch schon in dem Kapitel 6.1: Ereignisbasierte Systeme beschrieben worden.

9.4 Fakten nutzbar für benutzerdefinierte Regeln machen

In der ersten Entwicklungsphase des Plugins wurde die Möglichkeit geschaffen, Initialisierungsdateien bei dem Plugin bekannt zu machen und zu laden. Betrachtet man nun den vorangegangenen Artikel: Erstellen eines Faktes, so stellt sich die Frage, ob Regeln immer eingetragen werden können, die in ihrer Prämisse nach bestimmten Bedingungen in Fakten suchen. Es ist der Frage nachzugehen, ob nicht zu diesem Zeitpunkt schon zu jedem Fakt ein `Template` (`jess.Deftemplate`) vorhanden sein muss. Leider ist genau dieses der Fall.
Um nun die `Templates` der `Rete`-Instanz bekannt zu machen, wird zum Startzeitpunkt, noch vor dem Laden der Initialisierungsdateien, die `FactSetterRegistry` aufgerufen, sie solle alle vorhandenen FactSetter aufsuchen, um deren `Templates` der `Rete`-Instanz hinzuzufügen.
Somit ist eigentlich eine spätere Überprüfung, ob ein entsprechendes `Template` in der `Rete`-Instanz vorhanden ist, überflüssig. Es handelt sich nur darum sicher zu gehen, dass der neu erstellte Fakt auch wirklich gesetzt werden kann, ohne dass die `Rete`-Instanz stoppt. Der Rechenaufwand zur Laufzeit ist in so fern gering, als dass das Setzen der Fakten nur sporadisch, aufgrund von Veränderungen des Benutzers, geschieht. Erheblich höherer Rechenaufwand kann anfallen, wenn viele Fakten und Regeln vorhanden sind und es an die Evaluation der Prämissen geht.

Der Regelprogrammierer muss natürlich über die Form der automatisch generierten Fakten informiert sein. Man kann zwar mit Hilfe von jess-Befehlen alle Deftemplate und deren Aufbau betrachten. Das erscheint hier jedoch nicht praktikabel. Es ist in dem Hilfe-Teil für Benutzer der Aufbau von automatisch erzeugten Fakten erklärt. In Kapitel 9.7: Erweiterung der Hilfe sind noch Erklärungen dazu vorhanden.

9.5 Erweiterung der Einstellungen

Verweisend auf das Kapitel 5.7 : Einstellungen, sollen hier noch zusätzliche Einstellmöglichkeiten beschrieben werden.
Diese Einstellungen sind zweierlei Art. Zum einen kann das generelle Verhalten des Plugins eingestellt werden, zum anderen kann ausgewählt werden, für welche Editoren eine Überwachung vorgenommen werden soll. Wenn Änderungen durch diese Editoren initiiert werden, werden Fakten automatisch erzeugt.

Folgend nun ein Blick auf die Einstellmöglichkeiten bezüglich der zu überwachenden Editoren.

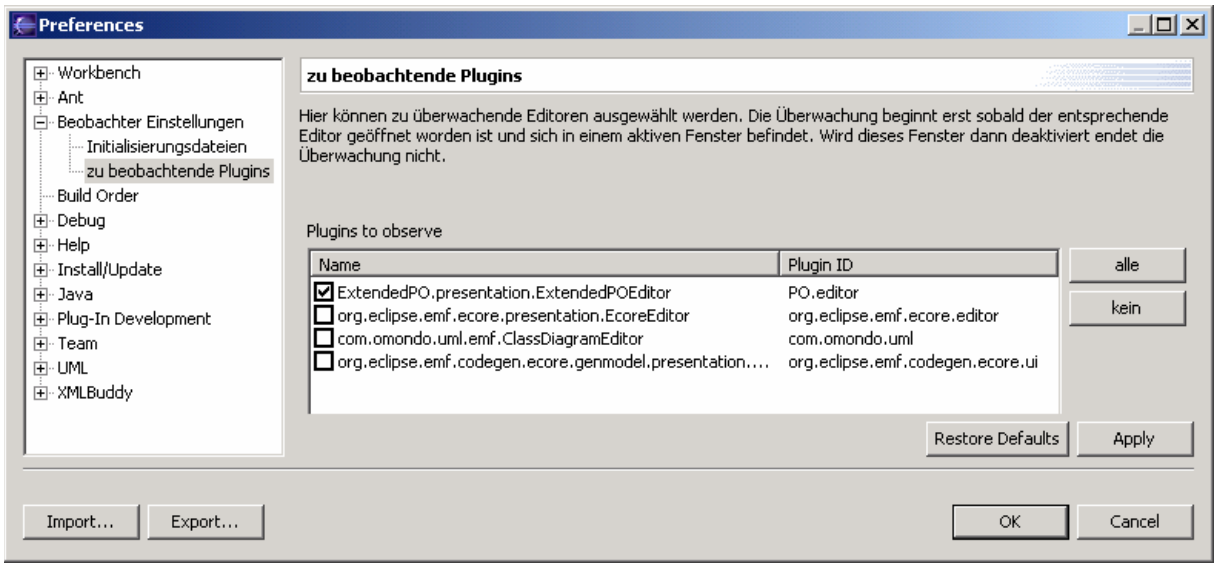

Abbildung 9.3: Einstellung der zu beobachtenden Plugins

Hier wird gleichbedeutend von Plugins wie von Editoren gesprochen. Tatsächlich ist es natürlich auch möglich, dass ein Plugin zwei verschiedene Editoren stellt. Genauer müsste es eigentlich heißen: ‚zu beobachtende Editorentypen'; diese Bezeichnungsweise ist jedoch nicht gewählt worden, um dem Benutzer *nicht* zu viele Details der Implementierung aufzuzwingen.
Die Liste der zur Verfügung stehenden Plugins / Editoren wird direkt vor der Betrachtung des Einstellungs-Dialoges erstellt.
Es wird nun der Frage nachgegangen, wie man an eine Liste der Editoren kommen kann. In dem Kapitel 8.2: EMF-Editoren und Item-Provider beschrieben. Bisher unbeachtet blieb die Tatsache, dass ein Editor auf ein gewisses Ereignis hin instanziert werden muss. Welche Ereignisse das sind, wird vom Plugin durch die Konfigurationsdatei: `plugin.xml`, in die Plugin-Registry eingetragen. Es gibt für jeden Editor einen Eintrag in der Registry, der an dem Erweiterungspunkt: `org.eclipse.ui.editors` ansetzt. Dieser Erweiterungspunkt ist vom Workbench definiert. Für alle Erweiterungen (`extension`) wird nun die realisierende Klasse herausgesucht. Diese Klasse verbirgt sich hinter dem Eintrag: `class=""`. Nur eine begrenzte Gruppe von Editoren ist von Interesse. Es müssen nämlich die Editoren, um observiert werden zu können, ItemProvider zur Verfügung stellen. Den untersuchten Editoren ist nun allen gemein, dass sie die Schnittstelle `IEditingDomainProvider` implementieren. Somit ist eine der Vorraussetzungen für die Eignung der Editoren (Kapitel 9.6) schon definiert. Die gefundene Klasse wird danach überprüft, ob sie, oder eine seiner Superklassen, die Schnittstelle `IEditingDomainProvider` implementiert. Falls dieses der Fall ist wird sie in die List der zur Auswahl stehenden Editoren eingereiht.

Zu den Einstellungen der allgemeinen Art ist wenig zu sagen. Man kann die Einstellung vornehmen, dass Änderungen im Allgemeinen nicht mehr verfolgt werden sollen, und somit werden auch EventHandler für Änderungen an Resourcen ausgetragen. Der Benutzer kann jedoch weiterhin jess-Quellcode ausführen. Welche Editoren aktiv sind und welche geschlossen werden, wird auch weiterhin verfolgt. Es wird jedoch dafür gesorgt, dass kein EventHandler bei deren ItemProvidern eingetragen werden. Deselektiert man lediglich die Einträge aus dem zuvor beschriebenen Einstellungsfenster, so werden selbstverständlich Änderungen an Resourcen weiterhin observiert.

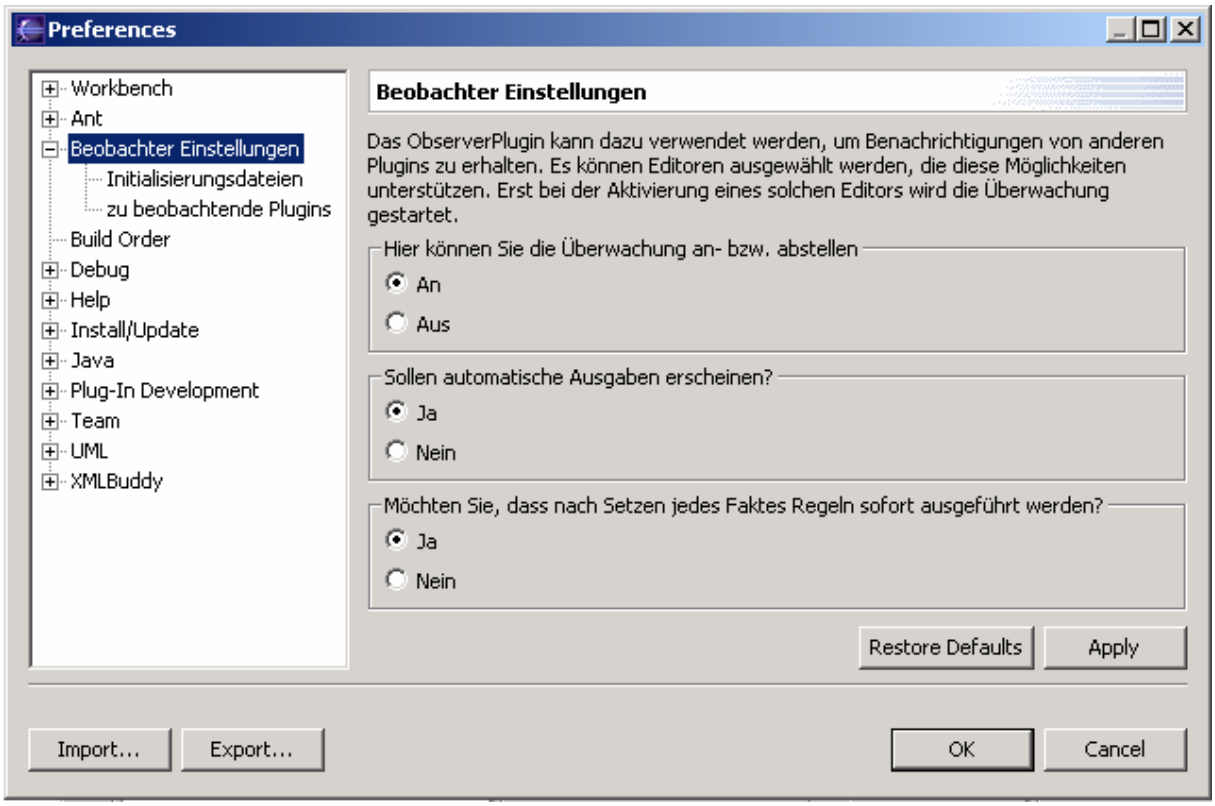

Abbildung 9.4: Allgemeine Einstellungen

Es wurde im Kapitel 5.5: Ausgabe-View schon beschrieben, dass die Ausgabe-Router der `Rete`-Instanz auf die Ausgabe umgeleitet werden. Werden Fakten hinzugefügt, Regeln erstellt, Regeln aktiviert, feuern Regeln, oder werden auch andere Sachen an der `Rete`-Instanz gemacht, kann man Ausgaben dadurch erreichen, dass man den jess-Befehl [`(watch all)`] ausführt. Dieser Befehl, so wie der inverse dazu, sind als Einstellmöglichkeit hier gegeben.
Hat man nun eine Regel der Regelbasis hinzugefügt so passiert erst einmal nichts, wenn ein die Prämisse erfüllender Fakt hinzugefügt wird, außer dass die Regel in die Agenda geschoben wird. Es muss der Befehl [`(run)`] ausgeführt werden, erst dann wird eine Regel gefeuert. Weitere Regeln feuern dann eventuell im Anschluss. Durch die dritte hier gegebene Auswahl kann ein Ausführen des Befehls `run()` nach dem automatischen Setzen eines Faktes erreicht werden.

Der Benutzer ändert die Einstellungen natürlich zur Laufzeit. Diese Änderungen beziehen sich auch alle auf sofort zu übernehmende Eigenschaften des Plugins. Es muss demnach ein Überwachungsmechanismus bereitgestellt werden, der dann reagieren kann.
Die Plugin-Klasse `ObserverPlugin` ist für die Verwaltung systemeigener Resourcen zuständig. Auch die Einstellungen, die als Zeichenketten in dem sogenannten PreferenceStore liegen, gehören dazu. Diese Klasse zur Speicherung der Schlüssel-Wert-Paare fungiert als EventSource und EventManager. Bei einer Änderung wird in diesem Fall der Listener in `ObserverPlugin` informiert. Dieser Listener (`PreferencesPropertyChangeListener`) ruft dann entsprechend der Änderung reglementierende Methoden auf.

9.6 Eignung der Editoren

Es ist an verschiedenen Stellen schon angedeutet worden, welche Vorraussetzungen erfüllt sein müssen, damit Fakten automatisch erzeugt werden können. Zuletzt wurde erläutert, was für das Einstellungsfenster (vgl. Kapitel 9.5: Erweiterung der Einstellung) von Bedeutung ist. Selbstverständlich müssen diese Bedingungen schon mal erfüllt sein, damit der entsprechende Editor überhaupt beachtet wird. Zusätzlich soll noch beschrieben werden, was genau passiert, wenn der Editor gefunden wird und ein EventHandler einzutragen ist.
Am besten wird die Vorgehensweise deutlich, wenn an dieser Stelle der Quellcode betrachtet wird. Alle Annahmen über den Editor werden hier zu erkennen sein. Außerdem wird anschaulich, was als Informationsquelle dient.

```
/**
Hier wird aus einem EditorPart welcher IEditingDomainProvider
implementiert, und dessen EditingDomain eine AdapterFactoryEditingDamain
ist, untersucht.
Falls eine solche AdapterFactory vorhanden ist wird diese hinzugefügt.
Ein bereits vorhandener Editor wird nicht nochmals hinzugefügt.
*/
private void getEditors(IWorkbenchPart part){
  if(part instanceof IEditingDomainProvider)
  {
    EditingDomain domain=((IEditingDomainProvider)part).getEditingDomain();

    if(  editorsToObserve.contains(part.getClass().getName()) &&
         domain != null &&
         domain instanceof AdapterFactoryEditingDomain
      )
    {

//trägt einen Listener in die AdapterFactory ein + Eintrag in Liste

       if(!listener.containsKey(part))
       {

         listener.put(part, new AdapterFactoryListener(
           ((AdapterFactoryEditingDomain)domain).getAdapterFactory(),
            new NotifyChangedListenerFactSetter()));

         updateListeners();
       }
    }
  }
}//getEditors
```

Drei Abfragen sind nun zu betrachten. In der ersten Abfrage wird die bekannte Bedingung überprüft, ob es sich bei dem Editor um einen solchen handelt, welcher die Schnittstelle `IEditingDomainProvider` implementiert. Eigentlich kann noch nicht einmal davon ausgegangen werden, dass der vorliegende `Part` ein Editor ist, es könnte auch eine Ansicht sein. Dieses kann jedoch universell behandelt werden.

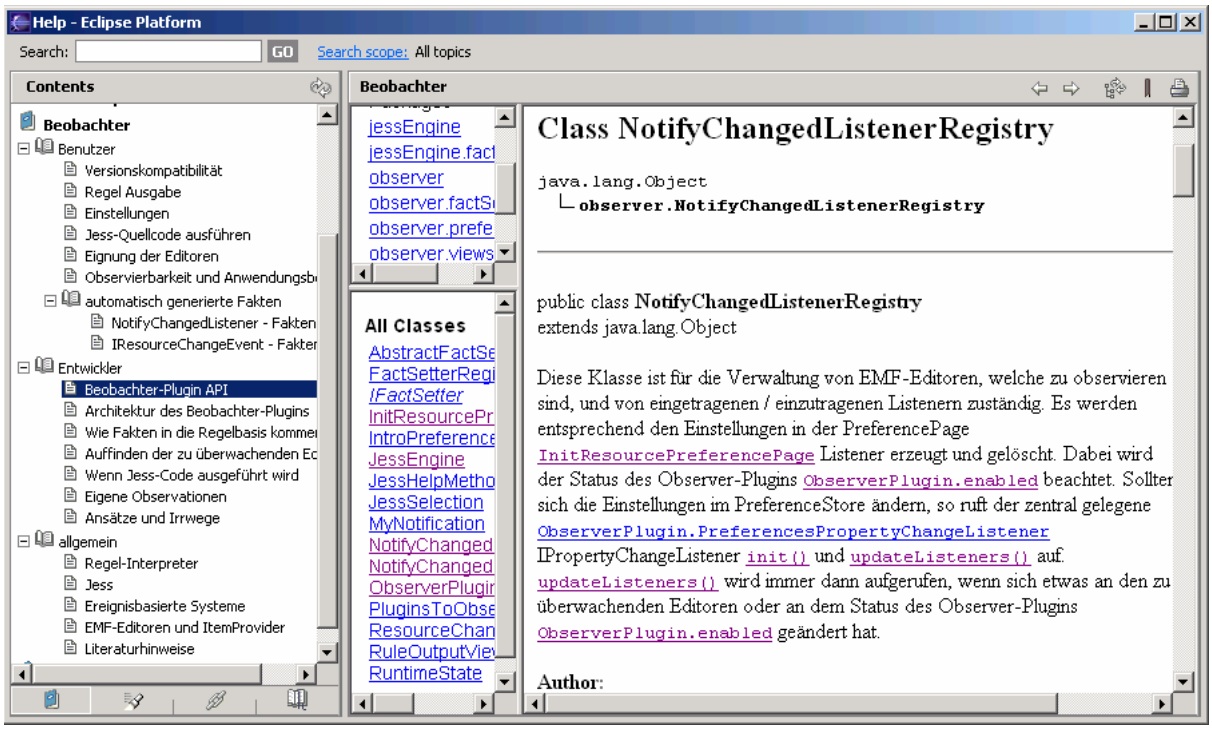

Abbildung 9.5: Hilfe in eclipse

Ein solcher Editor birgt eine `EditingDomain`. In der zweiten Abfrage wird überprüft, ob es sich dabei um eine des Typs `AdapterFactoryEditingDomain` handelt. Eine solche birgt bestimmt ItemProvider, in welche ein Listener eingetragen werden kann. Damit kommt man zur letzten Abfrage, in der überprüft wird, ob der Editor schon in eine vorhandene Liste von ItemProvider-Listener-Paaren eingetragen ist. Zu diesem Zweck wird ein speziell dafür implementiertes Objekt erzeugt, das diese Paare beinhaltet. Dieses Objekt hat die Fähigkeit, sowohl den Listener bei dem ItemProvider ein-, wie auch auszutragen. Diese speziellen Objekte werden nun derart gespeichert, dass ein Verweis zu der entsprechenden Editor-Instanz auch mit dazu gespeichert wird. So kann man in der letzten Abfrage überprüfen, ob für diese Editor-Instanz schon ein Eintrag vorhanden ist.
Wann immer Änderungen an dieser Liste oder an der, die die Einstellungen diesbezüglich speichert, auftreten, ist eine Aktualisierung der Stati (Listener ist eingetragen / nicht eingetragen) vorzunehmen [`updateListeners()`]. Man vergleiche dazu: Kapitel 9.5: Erweiterung der Einstellungen; `PreferencesPropertyChangeListener`.

9.7 Erweiterung der Hilfe

Auf die Hilfe-Dateien wird sehr viel Wert gelegt, da es sich bei diesem Plugin um eines handelt, das nicht intuitiv funktioniert. Der folgende Abschnitt: *Punktweise Erklärung zur Erweiterung der Observation* ist ebenfalls unter dem Punkt: *Eigene Observationen* in der Hilfe zu finden. Die Hilfe soll, genauso wie die Observer - API (a.a.O. S. v), helfen das Plugin zu benutzen, und zu erweitern. Anregungen und (nicht) gangbare Anbindungsmöglichkeiten sollen durch die Hilfe gegeben werden. Abbildung 9.5 zeigt die Observer - API

9.8 Punktweise Erklärung zur Erweiterung der Observation

Durch Erweiterung des Codes ist es möglich, automatisch eigene Fakten in die Regel-Maschine einbringen zu lassen.
Dieses Vorgehen ist zumeist notwendig, da es nicht möglich ist, für alle möglich Plugins, Speicherungsformen in Resourcen, Events... Fakten zu generieren, die allen Anforderungen entsprechen. Es ist ja auch zu beachten, dass eine signifikante Verlangsamung von eclipse vermieden werden muss. Deswegen muss gezielt dort erweitert werden, wo es notwendig ist.

Vorraussetzungen für Erweiterungen:

- Ein vorhandener und lokalisierbarer EventManager
- Kenntnis über die zu implementierende Schnittstelle eines EventHandlers
- Kenntnis über die Struktur des Events

(man vergleiche mit dem Kapitel 6.1: Ereignisbasierte Systeme)

Es sind nun ein paar Klassen zu schreiben und Einträge zu machen:

1. Diejenige Klasse, welche den EventManager ausfindig macht und einen neuen EventHandlerFactSetter einträgt: EventRegistry
2. Einen EventHandlerFactSetter
3. Einen Aufruf zur Initialisierung der EventRegistry aus der Plugin-Klasse `ObserverPlugin` aus.
4. Entsprechende Einträge in den `start()`, `stop()` Methoden der Plugin-Klasse.

   Optional:

5. Eine Preference-Seite (Einstellungsmöglichkeiten)
6. Eintrag in der `plugin.xml` für die Preference-Seite
7. Erweiterung des PreferenceChangeListener in `ObserverPlugin`
8. Hilfe-Seite
9. Eintrag in der `toc.xml` für die Hilfe-Seite

9.8.1 Eintrag in EventManager

Zu 1.,3.,4.:

In dem Paket `observer` ist eine Klasse zu ergänzen. Diese Klasse muss den EventManager ausfindig machen, bei dem man sich für ein bestimmtes Ereignis eintragen kann. Selbstverständlich muss diese Klasse auch instanziert werden. Dieses soll aus der Plugin-Klasse `ObserverPlugin` heraus passieren. Bereits implementiert ist die Klasse `NotifyChangedListenerRegistry`. Sie kann hier auch als Vorlage dienen. Eine abstrakte Klasse ist nicht vorgegeben, da der Aufbau dieser Klasse beliebig ist, so wie auch ihre Verwendung, Aktivierung und Deaktivierung. Da das gesamte Plugin über die

Preference-Seiten ein- und ausgestellt werden kann, sind die Methoden `start()` und `stop()` in der Plugin-Klasse implementiert. Es ist zu untersuchen, ob in so einem Fall in der neuen EventRegistry eine Aktualisierung auszuführen ist.

9.8.2 EventHandlerFactSetter

Zu 2.:

Ein EventHandlerFactSetter hat nun zwei Aufgaben: Zum einen muss er die notwendige Schnittstelle implementieren, um von dem EventManager eingetragen zu werden, und zum anderen ist die Schnittstelle `IFactSetter`, aus dem Paket `jessEngine.factSetter` zu implementieren. Diese FactSetter-Schnittstelle kann auch durch die Erweiterung der Klasse `AbstractFactSetter` übernommen werden.
Ein FactSetter muss zu seiner Fähigkeit, einen `jess.Fact` zu erzeugen, auch ein `jess.Deftemplate` erstellen können, das zu dem `Fact` passt. Aus diesem Grund ist die Methode: `Deftemplate getTemplate(Rete engine) throws JessException` zu implementieren.
Man vergleiche hierzu die Klasse: `NotifyChangedListenerFactSetter` .
Da die zu implementierende Klasse auch ein EventHandler ist, hat sie eine Methode, die auf Events reagiert. Diese Methode wird nun verwendet um einen `Fact` zu erzeugen. Zur Erzeugung eines Faktes holt sich die Methode von der Klasse `jessEngine.JessEngine` einen neuen leeren `Fact`. Hier an dieser Stelle wird nun von der Klasse `jessEngine.factSetter.FactSetterRegistry` überprüft, ob die aktive Regel-Maschine solch einen Fakt mit dem übergebenen Namen kennt. Falls nicht, wird bei der nun neu geschriebenen Klasse die Methode `getTemplate (Rete engine)` aufgerufen und, falls notwendig, ein neues `jess.Deftemplate` angelegt. Damit die FactSetterRegistry diese Funktion auch ausführen kann, muss sich die neue FactSetter-Klasse zuvor bei ihr registriert haben. Jeder FactSetter muss nun auf die Methode `getName()` eine andere Zeichenkette zurückliefern. Ansonsten kann die Namen-FactSetter-Zuordnung nicht funktionieren. Der `AbstractFactSetter` hat einen Konstruktor, der für die Registrierung sorgt. Erweitert man diese Klasse, muss man in dem Konstruktor `super()` aufrufen. Man kann natürlich auch andernorts eine Registrierung vornehmen.
Bei der Erstellung eines Faktes ist darauf zu achten, dass alle eingefügten Objekte die Schnittstelle `Serializable` implementieren, da sonst der Laufzeitzustand nicht gesichert werden kann. Diese Sicherung bricht bei einer Zuwiderhandlung komplett ab und wirkt sich somit nicht nur auf den selbst programmierten Teil aus.

9.8.3 Preference-Erweiterung

Zu 5., 6., 7.:

Das Anlegen ein Preference-Seite ist natürlich nicht verpflichtend. Auch soll hier nicht erklärt werden, wie man eine solche Seite anlegt, sondern lediglich darauf verwiesen werden, wie es bereits gemacht wurde. Die Klassen zur Verwirklichung einer Preference-Seite sind in dem Paket `observer.preferences`. Welche Editoren beobachtet werden sollen, kann man mit Hilfe der Klasse `PluginsToObservePreferencePage` einstellen. Diese Klasse spezialisiert `PreferencePage`. Einstellungen werden als Zeichenkette im PreferenceStore als Schlüssel-Wert-Paar gespeichert (siehe Methode: `performOK()`). An diese Werte kommt man von anderen Klassen aus, indem man die Methode `ObserverPlugin.getDefault().getPreferenceStore()` aufruft. Zur Erstellung einer solchen Preference-Seite sei hier auch auf [JEmE2] und [JDGtoE], S. I verwiesen.
Selbstverständlich sind dort auch die notwendigen Ergänzungen für die `plugin.xml` aufgezeigt. Man kann aber auch den folgenden Code Teil einfach anpassen und hinzufügen:

```
<extension point="org.eclipse.ui.preferencePages">
 <page
  name="zu beobachtende Plugins"
  category="observer.preferences.preferencePage.intro"
  class="observer.preferences.PluginsToObservePreferencePage"
  id="observer.preferences.preferencePage.pluginsToObserve">
 </page>
</extension>
```

Hier braucht man bei `class` nur den neuen Namen der implementierenden Klasse einzutragen und die `id` beliebig (jedoch einzigartig) zu verändern.
Wenn jetzt Einstellungen vorgenommen werden, so muss man ja darüber informiert werden, um darauf reagieren zu können. Sollte man nur zur Startzeit an den Einstellungen interessiert sein, braucht man folgendes natürlich nicht.
In der Plugin-Klasse `ObserverPlugin` ist ein EventHandler für diese Zwecke vorhanden: `PreferencesPropertyChangeListener`. Man kann hier in der Methode `propertyChange(PropertyChangeEvent event)` nach der Überprüfung, ob es sich bei dem geänderten Wert um denjenigen zu dem selbst definierten Schlüssel (bekommt man mit `event.getProperty()`) handelt, einen entsprechenden Methodenaufruf machen.

9.8.4 Hilfe-Erweiterung

Zu 8.,9.:

Eine Hilfe-Seite ist im HTML-Format zu schreiben. Es können auch Grafiken oder `*.pdf`-eiten eingebunden werden. Das Inhaltsverzeichnis ist in der Datei `toc.xml`, welche sich im Wurzelverzeichnis befindet, zu definieren. Werden eigene Hilfe-Seiten erstellt, muss dort ein Link hinzugefügt werden.

```
<topic label="X" href="html/developer_guide/event-based.html" />
```

ist ein solcher Eintrag. Unter dem Attribut `label` findet man den sichtbaren Text, und `href` bezeichnet einen Verweis von dem Wurzelverzeichnis aus.

Kapitel 10

Funktionale Erweiterung durch Suchanfragen

Eine weitere Einsatzmöglichkeit für die regelgesteuerte Manipulation von Daten kann aus Schaffung von einheitlichen Datenstrukturen für beliebige Quellen resultieren. Wenn eine solche einheitliche, vielleicht auch zur Laufzeit analysierbare Datenstruktur vorhanden ist, kann die Regelmaschine auch dazu angehalten werden, sich einen Überblick über den momentanen Zustand von beliebigen Konstrukten zu verschaffen. Momentan kann solch ein Zustand nur in so weit erfasst werden, als man von Anfang an alle Veränderungen beobachtet. Selbstverständlich kann man, bei bekannter Speicherungsart oder bei uniformen Zugriffsmechanismen, auch eine Analyse der Struktur und somit des momentanen Zustandes durchführen. Bislang ist jedoch eine genaue Kenntnis über die Zielstruktur zur Programmierzeit notwendig.
In EMF ist bereits ein Mechanismus vorhanden, der das Laden von EObjects aus `*.xml-Resourcen` unterstützt. Man muss sich in diesem Fall genau mit der Speicherstruktur auskennen, was hier zwar bei der Standardimplementierung in EMF noch möglich ist, durch die Laufzeitanalyse von EMF-Modelldaten. Bei einer Abänderung des Speichermechanismus' jedoch unmöglich wird.

Zur Identifizierung der gespeicherten EObjects wird die Position der Elemente innerhalb der Ressource benutzt (Beispiel: `//@customers.1`). Dieses ist leider bei mehreren beteiligten Clients ein Problem. Wenn einer ein Element löscht, verschieben sich die Positionen der anderen Elemente. Zwar könnte man solche Löschaktionen mitverfolgen, bei einem nicht deterministischen Automaten, wie der Regel-Maschine, ist das jedoch nicht möglich, da die Abarbeitung der Regeln, wenn neue Fakten hinzukommen, in beliebiger Reihenfolge geschieht. Eine Identifikation der serialisierten Objekte muss für so ein System mit einer Regel-Maschine anders aufgebaut sein.

Dazu wird exemplarisch eine Analyse von einem Teilprojekt von EMF durchgeführt. Dieses Teilprojekt, SDO, stellt einheitliche Datenzugriffe mit beliebigen Anfragesprachen zur Verfügung.

10.1 Vorstellung von SDO (Service Data Objects)

Ein von IBM und BEA Systems begonnenes Projekt verspricht interessant zu werden.

10.1.1 Ziele von SDO

- SDO soll sowohl statische wie auch dynamische DatenAPIs vereinen, da statische einfach, jedoch nicht immer ausreichend, sind und dynamische DatenAPIs genau diese Lücke füllen. Unter statischen DatenAPIs zählen Zugriffsarten festen Typs:
  `getName():String`
  anders dagegen wie bei JDBC: `resultSet.getString(„name“)`.

- Unterstützung von Tools und Frameworks:

  -Gewährleistung eines einheitlichen Datenzugriffs
  -dynamische APIs
  -Strukturen zur Laufzeit analysierbar.

- Unterstützung für nicht verbundene Daten:

  Zum Beispiel.: Web-Anwendung: Daten holen, Bearbeiten, Zurückspeichern.
  Beste Möglichkeit für die meisten Anwendungen scheint 'optimistic concurrency semantics' zu sein;
  Wenn ein Client eine Veränderung durchführt und derweil ein anderer Client eine Änderung auf den gleichen Daten-Strukturen vollendet hat, wird das Zurückschreiben des zuerst genannten zurückgewiesen.

- Unterstützung bei der Erstellung von eigenen Daten-Zugriffs-Schichten

- Trennung von Daten-Zugriffs-Schichten-Code und Anwendungs-Code

10.1.2 Zur Architektur

SDO besteht aus einer Komponenten-Architektur. Der Kern ist allgemein gefasst und setzt keine bestimmte Anfragesprache oder Datenquelle voraus. Vieles kann benutzt werden: XPath, XQuery, XML-Datenquellen etc. Der Kern definiert Data Objects, Data Graphs und eine Metadata-API.

Es wird das Konzept der '*disconnected data graphs*' verwirklicht. Ein Client bekommt also einen Graphen mit Daten, kann diesen verändern und diesen dann wieder zurückschreiben. Sollten Teile des Graphen von anderer Seite mittlerweile verändert worden sein, so wird das Zurückschreiben abgewiesen.

Anbindung an Datenquellen geschieht über einen Data Mediator Service. Diese Verbindungen vom Client zum Service werden immer wieder neu aufgebaut und getrennt, wenn Daten gelesen oder geschrieben werden.
Der Data Mediator Service erstellt die Data Objects und Graphen und ist auch für das Zurückschreiben in die Datenquellen zuständig.

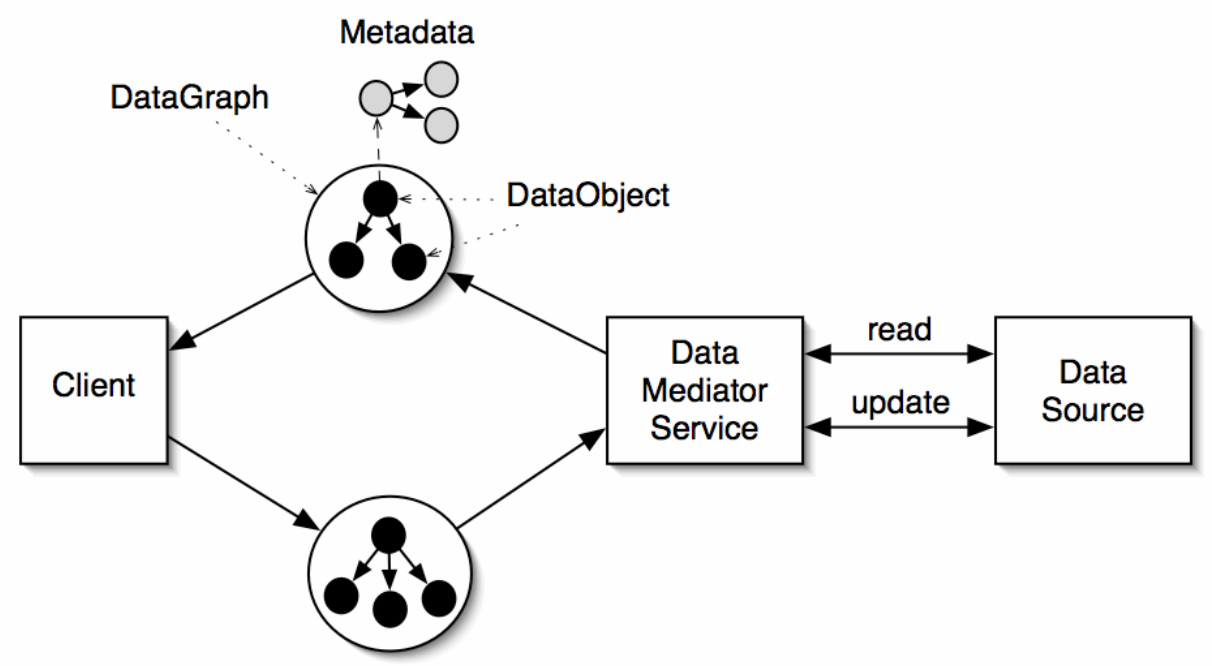

Abbildung 10.1: SDO Architektur, Quelle: SDO Whitepaper

So ein Data Mediator Service (vgl. Abbildung 10.1) kann für jede beliebige Datenquellenart oder Kombinationen aus mehreren geschrieben werden. Sie werden zusätzlich zu der Kern-Komponente benötigt.

Data Objects halten die gewünschten Werte, Referenzen auf andere Data Objects und Referenzen auf Metadaten (Metadata). Durch Metadaten kann eine Laufzeitanalyse der Daten und des Daten Graphen durchgeführt werden. Der Daten Graph (DataGraph) beinhaltet die Struktur der Data Objects sowie alle Änderungen, auch wenn neue Elemente hinzukommen, oder vorhandene gelöscht werden. Hier wird also eine Identifizierung der Elemente in dem Data Mediator vorgenommen. Es werden vom Client einfache XPath-Anweisungen genutzt, um Data Objects innerhalb des eigenen Graphen zu identifizieren.

10.2 Untersuchung der Eignung für Regel-Maschinen

Inwiefern die angestrebten Ziele der Entwickler jedoch verwirklicht werden, und wie einfach die Benutzung letztendlich wird, kann man leider noch nicht abschätzen.

Abgesehen davon hat der Data Mediator Service den Überblick über Veränderungen an den Datenquellen. Mit Hilfe der optimistischen Nebenläufigkeits-Semantik (optimistic concurrency semantics) können Identifikations-Probleme, die durch die nicht deterministische Abarbeitung von Regeln entsteht, umgangen werden.

Regel-Maschinen könnten durch die Möglichkeit der Analyse von Daten zur Laufzeit auch dazu benutzt werden, Bedingungen an die Daten und Datenstrukturen hinzuzufügen.

Ist also auch eine semantische Erweiterung der Datendefinitionen durch Regeln denkbar?

Solange nur Aktionen, nicht Prämissen, Nutzen aus Anfragen an außerhalb liegende Objekte ziehen, ist das tatsächlich eine Erweiterung der Funktionalität.

Leider ist eine der Voraussetzungen für die Effizienz von Regel-Maschinen, dass eine schnelle Bearbeitung der zu evaluierenden Prämissen möglich ist.
Wenn Objekte, auf denen Anfragen ausgeführt werden, außerhalb von der Regelmaschine residieren, dann müssen für die Evaluation von Prämissen bei *jedem* Zyklus Anfragen an außerhalb liegende Einheiten gestellt werden. Ein erheblicher Geschwindigkeitsvorteil der Regelmaschinen geht dadurch verloren; denn wenn es zur Prüfung der Voraussetzungen einer Regel nur lokal gehaltener Fakten bedarf, so müssen diese Prämissen nur dann geprüft werden, wenn sich ein beteiligter Fakt ändert. Über Änderungen wird man hier jedoch nicht informiert.
Dennoch muss eine Regelmaschine eine gemeinsame Teilbedingung mehrerer Prämissen nur einmal pro Zyklus, wenn überhaupt, evaluieren.
Die Benutzung einer Regelmaschine ist für den oben genannten Fall also fragwürdig und deren Effizienz im Einzelfall genau zu überprüfen.

Kapitel 11

Schlussbetrachtung

Man kann die Frage, ob es sich lohne, andere Plugins zu observieren, nicht einfach mit ja oder nein beantworten, vielmehr müssen von Fall zu Fall Entscheidungen getroffen werden. Der folgende Abschnitt schließt die Studienarbeit mit Erklärungen zur Eignung ab.

11.1 Observierbarkeit und Limitationen

Es sind im Prinzip 4 Fälle zu unterscheiden:

Ziel ist	Eigenes Plugin	Fremdes Plugin
EMF - generiert	• Benutzbarkeit der vorhandenen Observation • Adaption an Objekte auch direkt implementierbar • Beliebig erweiterbar	• Benutzbarkeit der vorhandenen Observation • Falls andere EventManager und EventSource vorhanden sind erweiterbar
Nicht EMF – generiert	• Vorhandener Observations-mechanismus so nicht nutzbar • Erstellen eigener EventManager und EventSource • Erstellen zusätzlicher Klassen im Observer-Plugin notwendig	• Vorhandener Observations-mechanismus so nicht nutzbar • EventManager und EventSource müssen vorhanden sein, ansonsten nicht observierbar • Erstellen zusätzlicher Klassen im Observer-Plugin notwendig

Abbildung 11.1: Fallunterscheidung für Observierbarkeit

Eigene Plugins können immer observiert werden. Es muss lediglich ein EventManager vorhanden sein, an dem ein Listener registriert werden kann. In den Kapiteln Ereignisbasierte Systeme (6.1) und Punktweise Erklärung zur Erweiterung der Observation (9.8) können Informationen dazu gefunden werden.
Wenn das zu observierende Plugin von fremder Hand geschrieben wurde und man keinen Zugriff auf den Quellcode hat oder diesen nicht ändern möchte, gibt es zwei Möglichkeiten. Entweder ist es ein EMF-generiertes Plugin und erfüllt die beschriebenen Bedingungen (vgl. Kapitel 9.6: Eignung der Editoren), oder es ist irgend ein anderer erreichbarer Benachrichtigungsmechanismus vorhanden. In letzterem Fall kann das Observer-Plugin wie beschrieben erweitert werden.

11.2 Nachteile des Event - Mechanismus

Benutzt man die vorhandenen Observationsmechanismen, muss man sich darüber im Klaren sein, dass nur von denjenigen Events, welche von den Item-Provider weitergeleitet werden, auch Notiz genommen wird. Im Kapitel 8.2: EMF-Editoren und Item-Provider ist dieses Verhalten bereits erwähnt worden. Sollten nicht alle notwendigen Ereignisse weitergeleitet werden, ist im konkreten Fall darüber nachzudenken, eigene ItemProvider zu schreiben und diese dann denen des zu observierenden Plugins hinzuzufügen. Das könnte auch gemacht werden, wenn es sich um ein fremdes Plugin handeln sollte, da eine Referenz auf die ItemProvider vorhanden ist. Diese ItemProvider sind in Form von Adapterfabriken in einer Vereinigung solcher zusammengefasst. Man kann dieser Vereinigung, die sich auch wie eine einzelne Adapterfabrik verhalten kann, noch weitere Adapterfabriken hinzufügen. In EMF 2.** sind jetzt noch neue `ItemProviderAdapterFactory` – Klassen hinzugekommen, welche hoffen lassen, einfache Implementierungen hinzufügen zu können. Diese Vorgehensweise ist jedoch aus Zeitgründen noch nicht weiter verfolgt worden.

11.3 Beobachtung von Zustandsänderungen: nicht von Zuständen

Ein wesentlicher Punkt ist, dass bei dieser Art der Implementierung, nämlich durch die Beobachtung von Zustandsänderungen, nicht zwingend der aktuelle Zustand konsistent repräsentiert wird. Es ist zwar bei geschickter Implementierung so, dass der Initialzustand eines beliebigen Objektes erfasst wird. Dann folgend werden alle Veränderungen verfolgt. Es handelt sich hier demnach um den aktuellen Zustand des Objektes, wie er in dem Regelinterpreter erscheint. Sollte allerdings der Regelinterpreter von der Erstellung des Objektes keine Notiz nehmen, oder eine der Zustandsänderungen nicht mitbekommen, so spiegelt die Regel-Maschine einen falschen Zustand wieder. Alle Problematiken der zweifachen Repräsentation von Zuständen greifen hier. So können auch Original-Objekte mehrfach geändert werden und die Änderungen werden in fascher Reihenfolge auf die Kopien übertragen.
Pluginübergreifende Abhilfe könnte man schaffen, wenn man die Änderungen an Objekten über eine zentrale Instanz der Laufzeitumgebung abwickeln würde. So etwas ist leider in dem Kern von eclipse nicht vorgesehen. Man vergleiche hierzu auch [Ctoe].
Mit der vorhandenen Implementierung der Observation kann der Zustand von Objekten auch bedingt beobachtet werden. Der Schreiber der Regeln muss nun *aktiv* nach Zuständen der Objekte fragen, auf welche ein Verweis vorliegt - natürlich kann dieses nicht geschehen wenn kein solcher vorhanden ist.

11.4 Veranschaulichung der Problematik

Zur Veranschaulichung der Einsetzbarkeit und von Problemen soll hier ein Beispiel genannt werden. In der schon öfter erwähnten Objektstruktur von Zulieferern, Kunden, Bestellungen und Waren soll eine Überprüfung vorgenommen werde, ob die Bezeichnungen der Waren

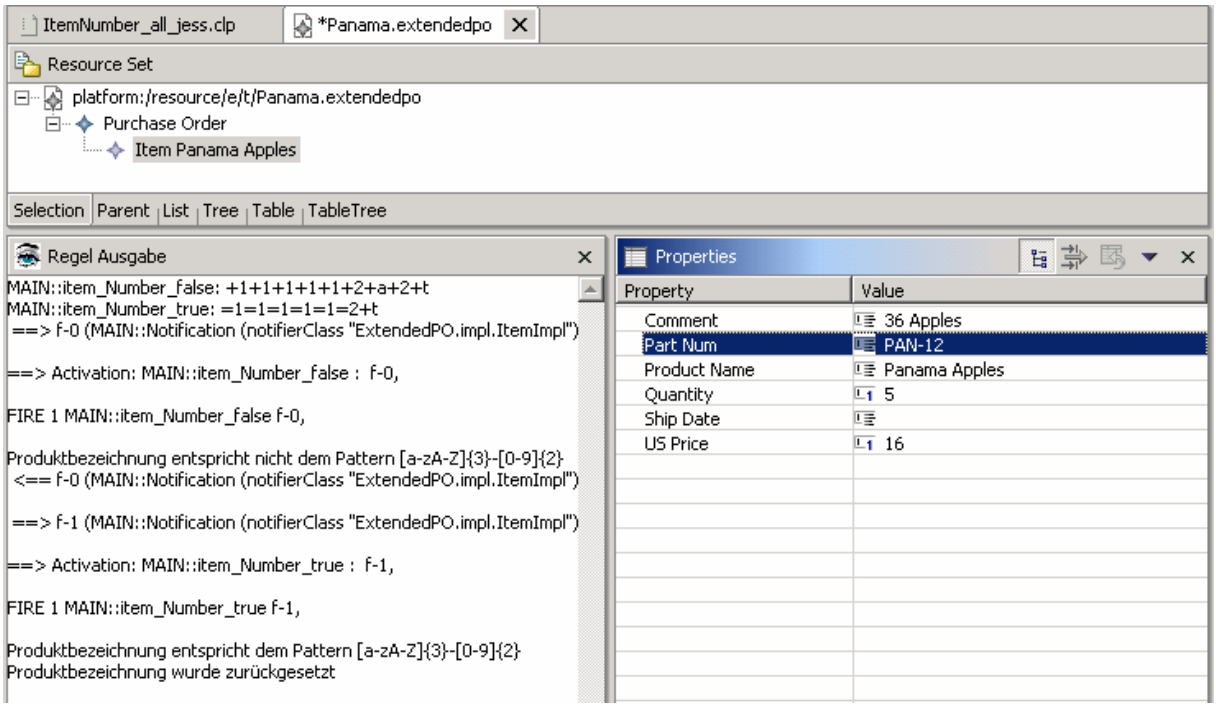

Abbildung 11.2: Anwendungsbeispiel Produktbezeichnung

einer bestimmten Norm entsprechen. Normalerweise ist so etwas im Quellcode des implementierenden Plugins zu finden. Man stelle sich nun vor, dass man keinen Zugang zum Quellcode habe, oder vielleicht auch nur nicht weiß, wo eine Ergänzung gegebenenfalls zu machen wäre. Nun kommt das Observer-Plugin wie gerufen.
Mit Hilfe von einer Regel kann nun die Erfüllung der Norm kontrolliert werden. Falls die Norm verletzt wurde, kann der vorherige Zustand wieder hergestellt werden.

```
(defrule item_Number_false
      ?itemFactW <-
            (Notification
                  (notifierClass "ExtendedPO.impl.ItemImpl")
                  (featureId 5)
                  (newValue ?nValue)
                  (serNot ?myNotification)
            )
      (not (test (call ?nValue matches "[a-zA-Z]{3}-[0-9]{2}")))
      =>
      (printout t  "Produktbezeichnung entspricht nicht dem Pattern [a-zA-
Z]{3}-[0-9]{2}")
      (retract ?itemFactW)
      (bind ?notifierObject (call ?myNotification getNotifier))
      (bind ?oldValue (call ?myNotification getOldValue))
      (bind ?feature (call ?myNotification getFeature))
      (call ?notifierObject eSet ?feature ?oldValue)
      (printout t  "Produktbezeichnung wurde zurückgesetzt")
)
```

Sollte nun eine Produktbezeichnung eingegeben werden, die anders ist als: drei Buchstaben gefolgt von zwei Ziffern, so wird der vorherige Wert wieder hergestellt.

Für einen geübten Regelschreiber ist das kein Problem und sehr nützlich, zumal man nun außerhalb des eigentlichen Plugins die Geschäftsregeln definieren kann. Für ungeübte kann das jedoch auch zu fatalen Abstürzen führen. Man stelle sich vor, ein Produkt wird neu erstellt. Dann entspricht die Produktbezeichnung wahrscheinlich nicht der gewünschten

Spezifikation. Es wird also bei einer Änderung der Produktbezeichnung, so denn die Änderung eine falsche ist, der vorherige falsche Wert wieder eingesetzt. Ein neuer Fakt mit dem neuen (alten) Wert wird gesetzt, und die gleiche Regel feuert wieder. Somit ist eine Endlosschleife entstanden. Es sind ähnlich Situationen aus anderen Gründen denkbar: was passiert, wenn eclipse vor dem Feuern der Regel geschlossen wird? Referenzen zu der EventSource gehen verloren, auch wenn der Laufzeitzustand gesichert wurde.

Soll bei richtiger Eingabe der Produktnummer eine Aktion ausgeführt werden, so muss eine separate Regel definiert werde, ein Konstrukt wie `if()... else...` kann hier nicht Verwendung finden. Man muss nun, falls sich die Art der Produktbezeichnung ändern soll, beide Regeln identifizieren und entsprechend abändern.
Es wird demnach deutlich, dass eine sehr genaue Programmierung in jess notwendig ist. Fatal ist dabei, dass man auf die Werkzeuge, die einem zur Verfügung stehen, wenn man zum Beispiel in Java programmiert, verzichten muss (JUnit, Debug, automatische Übersetzen ...).

Eine mögliche Anwendung für dieses Plugin ist die Unterstützung eines (möglicherweise grafischen) Editors, der mitunter Regeln definiert. Das Observer-Plugin könnte helfen, eine Trennung von Entitäten und Regeln zu erreichen. Entscheidender Vorteil ist bei einer solchen Verwendung, dass durch die automatische Erstellung von Regeln Fehler durch Benutzereingaben vermieden werden könnten.

Falls ein Plugin unter eigener Kontrolle geschrieben wird, kann die Fähigkeit, von jess JavaBeans einzubinden helfen. In dem Kapitel 3.2: Shadow Facts finden sich Informationen darüber.

Es ist in den vorangegangenen Betrachtungen deutlich geworden, dass keine generelle Empfehlung für die Verwendung einer solchen Observation gegeben werden kann. Eine individuelle Analyse aufgrund der zuvor besprochenden Punkte muss zeigen, ob die Verwendung des Plugins einen Vorteil bringt.

Literaturverzeichnis

[JDGtoE] Java Developer's Guide to ECLIPSE
Sherry Shavor, Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, Pat McCarthy
Verlag: Addison Wesley

[Ctoe] Contributing to eclipse, Principles, Pattern and Plug-Ins
Erich Gamma, Kent Beck
Verlag: Addison Wesley

[eMF] eclipse Modeling Framework
Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, Timothy J. Grose
Verlag: Addison Wesley

[JEmE2] Java-Entwicklung mit Eclipse 2
Berthold Daum
Verlag: dpunkt

[JIA] Jess in action
Ernest Friedman Hill
Verlag: Manning

[SWArch] Software-Architectures – Script des Fachbereichs Softwaresysteme der Teschnischen Universität Hamburg Harburg, Stand 2003
Hans-Werner Sehring, Miguel Garcia
Erhältlich: www.sts.tu-harburg.de

[OAPI] Java – Dokumentation des Observer-Plugins

[Marti] Vorlesungsmaterialien zur Produktionssystemen
von R. Marti
Technische Hochschule Zürich
http://www.inf.ethz.ch/~domnitch/teaching/ss2002_wbs/script/prodregeln.pdf

[NGDP] Next-Generation Data Programming: Service Data Objects
A Joint Whitepaper with IBM and BEA
http://www.eclipse.org/emf/ documentation => whitepaper

[IinUML] Rule-Based Detection of Inconsistency in UML Models
W. Liu, S. M. Easterbrook and J. Mylopoulos
Presented at the Workshop on Consistency Problems in UML-Based Software Development, at the Fifth International Conference on the Unified Modeling Language, Dresden, Germany, October 1, 2002.
http://www.cs.toronto.edu/~wl/papers/2002/uml02wl.18.pdf