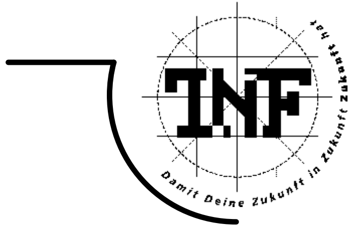
JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis



Coco/R Eclipse Plug-in

BAKKALAUREATSARBEIT
(Projektpraktikum)

zur Erlangung des akademischen Grades

BAKKALAUREUS DER TECHNISCHEN WISSENSCHAFTEN

in der Studienrichtung

INFORMATIK

Angefertigt am *Institut für Systemsoftware*

Betreuung:

Dipl.-Ing. Christian Wimmer
o.Univ.Prof. Dipl.-Ing. Dr. Dr. h.c. Hanspeter Mössenböck

Eingereicht von:

Christian Wressnegger

Linz, September 2006

Zusammenfassung. In dieser Bakkalaureatsarbeit wird die Integration des Werzeugs Coco/R („Compiler Compiler for Recursive Descent“) in das Open-Source Framework Eclipse präsentieren und erläutern. Es wird neben dem entstandenem Plugin und dessen Implementierungskonzepten, auch auf die Architektur des Eclipse Plugin Systems eingegangen.

Abstract. This bachelor thesis describes on the integration of Coco/R („Compiler Compiler for Recursive Descent“) into the open-source framework Eclipse. Beside the implemented plug-in, which was part of the thesis, the Eclipse SDK itself and especially the architecture of it's plug-in system will be discussed and illustrated.

Inhaltsverzeichnis

1 Einleitung 1
  1.1 Coco/R 1
  1.2 Eclipse 1
2 Die Eclipse Platform 3
3 Eclipse Plugin Architektur 5
  3.1 Manifest Dateien 7
  3.2 Extensions 9
  3.3 OSGi Framework 11
4 Coco/R für Java 12
5 Das Coco/R Plugin 14
  5.1 Überblick 14
  5.2 Builder und Nature 14
  5.3 Integrated Development Environment (IDE) 17
    ATG Modell. 17
    Editor. 18
    Content Outline View. 22
    Coco/R Extension Wizard. 24
  5.4 Coco/R Integration 25
6 Ausblick 27
7 Zusammenfassung 28
A Benutzerdokumentation 30
  A.1 Systembeschreibung 30
    Zweck. 30
    Voraussetzungen. 30
    Bedienung. 30
    Datenfluss. 31
    Einschränkungen. 31
  A.2 Installationsanleitung 31
    Offline. 35
  A.3 Bedienungsanleitung 36
    Getting Started. 36
    Erstellen eines neuen Projekts. 37
    Importieren eines bestehenden Projekts. 40
    Navigation. 43

1 Einleitung

Diese Bakkalaureatsarbeit besteht aus zwei Teilen: Zum Einen einer praktische Arbeit in Form eines Eclipse SDK Plugins und zum Anderen aus diesem Dokument, welches vorallem das entstandene Produkt und dessen Umgebung zum Thema hat. Beides kann von der Projekt-Homepage, zu finden unter [Wressnegger, 2006], bezogen werden.
Die beiden Komponenten, welche es im Rahmen dieser Arbeit miteinander zu vereinen galt, waren der Compiler Generator Coco/R und die Eclipse SDK, einem Framework zur Integration unterschiedlichste Software-Projekte.
In weiterer Folge dieser Einleitung wird kurz auf die Entstehungsgeschichte und grundsätzlichen bzw. markantesten Merkmale dieser beiden Projekte eingegangen.

1.1 Coco/R

Coco/R (zu finden unter [Mössenböck et al., 2006]) ist ein Compiler Generator für LL(1) Grammatiken, dessen Ursprung im Compiler-Compiler Coco liegt und in [Rechenberg and Mössenböck, 1985] beschrieben wurde. 1983 begann durch Hanspeter Mössenböck in Zusammenarbeit mit Peter Rechenberg die Entwicklung an der Johannes Kepler Universität Linz.
Mit dem Wechsel von Hanspeter Mössenböck an die ETH Zürich spaltete sich auch der Entwicklungsstrang dieses Projekts und es entstanden als Nachfolger der Coco/R für Oberon (siehe [Mössenböck, 1990a] und [Mössenböck, 1990b]) und Coco-2. (beschrieben in [Dobler and Pirklbauer, 1990])
Bis heute sind unterschiedlichste Varianten und Portierungen des Coco/R für diverse Sprachen entstanden: z.B. Oberon, Modula-2, Pascal, Delphi, C/C++, Java, C#, Icon/Unicon, Ruby, Ada.
Da das Eclipse Projekt vorallem für dessen hervorragende Entwicklungsumgebung für Java bekannt wurde, lag es nahe, sich im Rahmen des Projektpraktikums und dieser schriftlichen Ausarbeitung in erster Linie mit der Coco/R Version für Java zu beschäftigen. Dementsprechend wurde das Plugin darauf ausgelegt und gleichzeitig darauf geachtet, die Tür für andere Sprach-Varianten offen zu halten.

1.2 Eclipse

Die Eclipse Platform entstand als Nachfolger von *IBM Visual Age* an den *IBM Research OTI Labs* und wurde Ende 2001 nach dem ersten Major-Release der von IBM initierten Eclipse Foundation übergeben. Das ursprüngliche Eclipse Consortium bestand zu diesem Zeitpunkt aus Borland, IBM, MERANT, QNX Software Systems, Rational Software, Red Hat, SuSE, TogetherSoft und Webgain, welches heute laut [Eclipse Foundation, 2006a] auf mehr als 80 Mitglieder angewachsen ist.

Seit der Übergabe an die „Community“ werden die Produkte unter freien Lizenzen vertrieben. Während bis Ende 2004 (9. September 2004) ausschließlich die CPL, wie sie unter [IBM, 2002] zu finden ist, verwendet wurde, wechselte man danach in einem längeren Übergangsprozess zu einer eigens auf das Projekt zugeschnittenen Lösung, der „Eclipse Public License“ (EPL – siehe [Eclipse Foundation, 2004]). Die am 18. Juni 2005 erschienene Version 3.1 der Eclipse Platform, wie auch alle darauf folgenden Versionen[1] standen somit komplett unter der ersten Ausgabe dieser Lizenz.

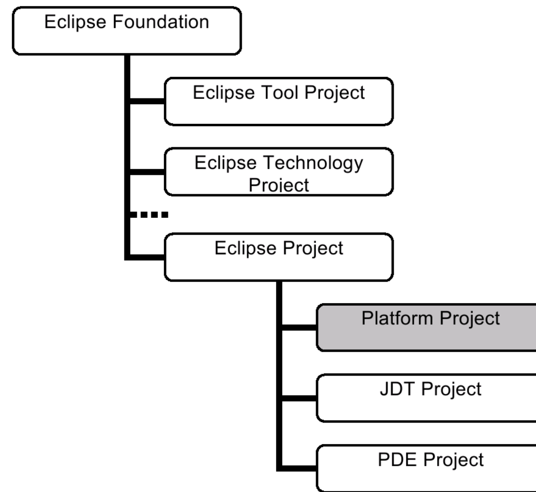


Abbildung 1. Organisatorische Aufteilung der Eclipse Foundation

Das Aushängeschild der Eclipse Foundation, das Eclipse Project unterteilt sich, wie in Abbildung 1 zu sehen, wiederum in 3 Teilbereiche:

- Eclipse Platform.
- Java Development Tools (JDT) Project.
- Plugin Development Environmet (PDE) Project.

In den folgenden Kapiteln wird die Architektur der Eclipse Platform selbst und im speziellen das Plugin System thematisiert werden. Die bekannt gewordenen *Java IDE* oder das *Plugin Development Environment*, sowie die anderen „Groß-Projekte“ der Foundation (*Eclipse Tools Project* etc.), sollen jedoch nicht näher behandelt werden.

[1] Eclipse SDK Version zum Zeitpunkt der Erstellung dieser Arbeit: 3.2

2 Die Eclipse Platform

Für das dritte Major-Release der Eclipse Platform wurde diese komplett umstrukturiert, um das Konzept einer *Rich Client Platform* (RCP) zu implementieren. Spätestens mit diesem Schritt gelang die Distanzierung von der allgemeinen Meinung, bei der Eclipse Platform handle es sich ausschließlich um einen IDE („Integrated Development Environmet“). Auch die offizelle Antwort auf die Frage, was die Eclipse Platform sei, entwickelt sich von *‚The Eclipse Platform is an IDE for anything and for nothing in particular‘*, wie sie in [Eclipse Foundation, 2003] zu finden war, zu *‚Eclipse is a universal Platform for integrating tools‘* (siehe [Eclipse Foundation, 2006a]).

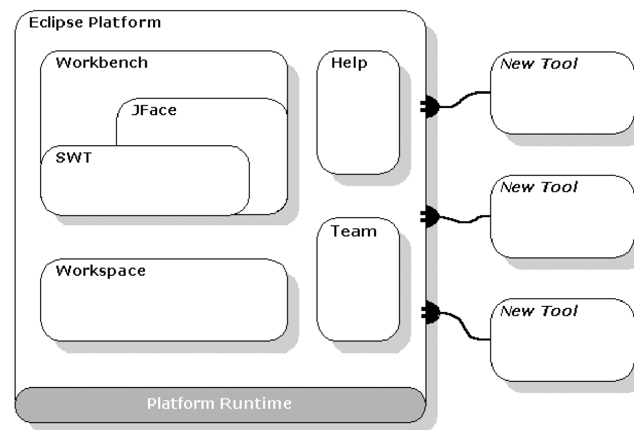


Abbildung 2. Zusammensetzung der Eclipse Platform. Diese Abbildung wurde [Eclipse Foundation, 2006b] entnommen.

Die **Eclipse RCP** ist ein minimales Sub-Set an Plugins, welche für den Aufbau einer einfachen Rich Client Anwendung nötig sind. In Bezug auf Abbildung 2 sind dies:

- Runtime (org.eclipse.core.runtime)
- Workbench (org.eclipse.ui)

und deren Abhängigkeiten.
Es ist allerdings zwischen dem *Runtime Plugin* und der *Platform Runtime* zu unterscheiden. Letztere implementiert die grundlegenden Konzepte des Plugin-System und orientiert sich hierzu am OSGi Framework. (mehr dazu im folgenden Kapitel 3.3)

Sie stellt also den Kernel des Systems dar, welcher die Plugins, die Plugin-Registry sowie *Extensions* und *Extension Points* (siehe Kapitel 3.2) verwaltet, dynamisch Plugins erkennt und bei Bedarf lädt.
Die Runtime ist selbst also (als einziges Element der Eclipse Platform) kein Plugin, lädt aber das Plugin **org.eclipes.core.runtime** zu dessen Unterstützung nach.
Das Starten der Platform übernimmt der sogenannten *Platform Launcher*, eine C Applikation, welche die Java VM aufruft und in Form des Startup-Archivs (`startup.jar`) den Programm-Code der Runtime lädt. Weiters werden über diesen die Rückgabewerte der Java VM ausgewertet und entsprechend als Output aufbereitet bzw.in Log-Dateien geschrieben, oder z.B. bei der Installation eines neuen Plugins zur Laufzeit, die Platform neu gestartet.

Die *Workbench* stellt das User Interface der Eclipse Platform dar. Hierbei kommen zwei Toolkits zur Anwendung:

- *Standard Widget Toolkit (SWT).* Im Gegensatz zu anderen Bibliotheken für graphische Oberflächen, welche für den Einsatz mit Java in Frage kommen, wie z.B. Swing, setzt SWT, wie auch AWT, auf die nativ implementierten Widgets des darunter liegenden Betriebsystems. Sollte ein Element in einer speziellen Umgebung nicht zur Verfügung stehen, wird dieses „selbst“ gezeichnet. Somit werden die Vorteile der erwähnten Alternativen kombiniert, ohne die system-unabhängige API einzubüßen.

- *JFace.* Hierbei handelt es sich um eine zusätzliche Abstraktionsschicht für das SWT, welche darauf aufsetzt, um aus den von dort gelieferten Basiskomponenten komplexere Elemente zusammenzusetzen. Diese umfasst neben Wizards und einzelne dort verwendete Standard-Seiten, Dialoge, etc., vor allem *Viewer* nach dem *Modell-Viewer-Controler* (MVC) Konzept.

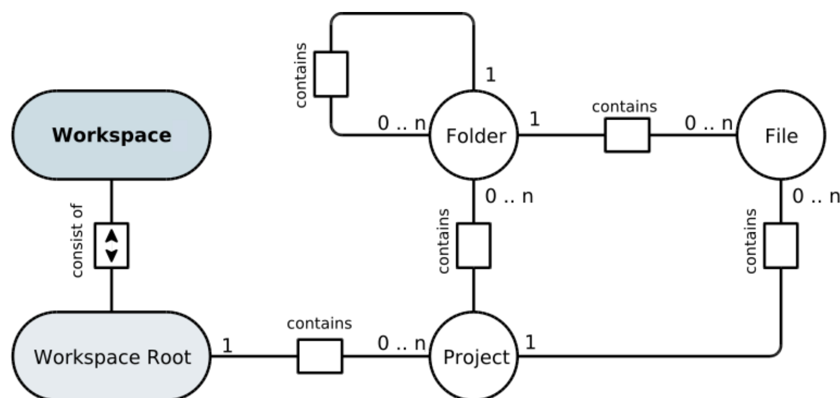


Abbildung 3. Struktur des *Workspace Resource Model.*

Das **Eclipse IDE Framework** selbst setzt nun auf diesem Konzept auf, wie es auch jeder anderen Applikation möglich wäre. Eine entscheidende Erweiterung stellt hier das *Workspace Resource Model* dar, also das der IDE zu Grunde liegende Datenmodell. Jedes Plugin kann nun, im Rahmen einer IDE, auf die Ressourcen des jeweiligen Workspaces zugreifen. Hierbei unterscheidet man zwischen folgende Elementen (siehe Abbildung 3):

- *Workspace Root.* Der gesamte Workspace ist hierarchisch aufgebaut, dementsprechend existiert je Workspace ein Wurzel-Knoten, welcher ausschließlich Projekte enthalten darf.

- *Project.* Ein Projekt umfasst alle Daten und Dateien, welche thematisch zusammen gehören sollen. Es muss allerdings immer zumindest die Datei `.project` enthalten sein, die die beschreibenden Informationen des Projektes enthält.

- *Folder.* Stellt lediglich ein hierarchisches Hilfmittel dar, um innerhalb eines Projektes weitere Abgrenzungen vornehmen zu können.

- *File.* Repräsentiert eine Datei des jeweiligen Projektes (Source-Code, Debug-Information, etc.).

Das Coco/R Plugin stellt zwar im Prinzip eine Erweiterung der JDT dar, operiert allerdings als eine eigenständige IDE für ATG Dateien (näheres dazu im Kapitel 5), welche in der aktuellen Version[2] allerdings per Wizards und Menü-Einträgen an Java Projekte gebunden wird. Insofern setzt dieses Plugin genau auf dieser Ebene auf, dem *Eclipse IDE Framework*.

3 Eclipse Plugin Architektur

Die folgenden Ausführungen und Erläuterungen orientieren sich hauptsächlich an [Bolour, 2003], [Gamma and Beck, 2004], [Clayberg and Rubel, 2004] sowie [Daum, 2006].

Seit der dritte Version der Eclipse SDK implementiert diese das OSGi Framework (mehr dazu im Kapitel 3.3), sodass Plugins durch OSGi Bundles dargestellt werden. Ein Plugin stellt also in Bezug auf die Eclipse Workbench, eine bestimmte Art eines Dienstes dar.
Hierzu muss jedes Runtime-Plugin die Klasse **org.eclipse.core.runtime.Plugin** erweitern, welche die grundlegenden Funktionen bereitstellt und seit 3.0 selbst das Interface **org.osgi.framework.BundleActivator** implementiert. Darin werden die beiden Methoden zum Starten bzw. Stoppen des Plugins bzw. der Bundles vorgeschrieben.

[2] Coco/R Eclipse Plugin Version zum Zeitpunkt der Erstellung dieser Arbeit: 0.1

Wenn man allerdings gemeinhin von Plugins spricht, so wird oft der Begriff eines „Features“ mit dem des „Plugins“ gleichgestellt. Diese sind allerdings für ein näheres Verständnis der Plugin Architektur, ebenso wie von dem Konstrukt eines „Fragments“ unbedingt zu trennen.

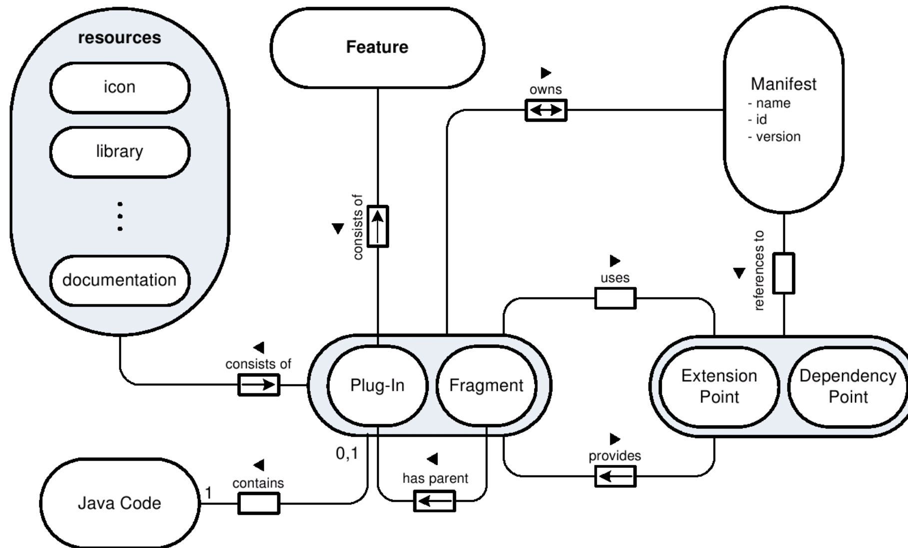


Abbildung 4. Struktureller Zusammenhang der einzelnen Teile eines „Plugins“. Diese Abbildung wurde [Perscheid, 2005] entnommen.

Wie auf Abbildung 4 zu sehen, besteht ein Feature aus einem oder mehreren Plugins, welche jeweils wiederum mehrere Fragmente unter sich vereinen können. Diese Abstraktion existiert, um größere Projekte, bestehend aus mehreren einzelnen Plugins, von einfachen Plugins zu trennen bzw. deren Handhabbarkeit und somit die Installation zu erleichtern.
Fragmente besitzen keine eigene „Plugin-Klasse“ (also eine Klasse, die, wie zuvor erwähnt, `org.eclipse.core.runtime.Plugin` erweitert) und ist somit ohne einem *Host-Plugin* nicht lauffähig. Der Vorteil besteht darin, dass das Fragment nach der Installation zwar zum Namensraum des Host-Plugins gehört, sodass dieses auf die Funktionalität des Fragments zugreifen, jedoch vollkommen autonom entwickelt oder auch aktualisiert werden kann.
Ein Fragment wird von der Eclipse Runtime zwar logisch, allerdings nicht physisch einem Plugin zugeordnet. Insofern stellt ein Fragment für das Host-Plugin lediglich eine optionale Erweiterung dar, wird also für ein korrektes Funktionieren nicht unbedingt gebraucht. Ein häufiges Einsatzgebiet stellen hier z.B. Spracherweiterungen dar.

Jedes dieser 3 Elemente wird von zumindest einer `Manifest`-Datei (siehe Folgekapitel 3.1) beschrieben. Dies hat den Zweck, Plugins z.B. im User Interface zu repräsentieren bzw. es später zu laden und somit unabhängig vom Quellcode darstellen zu können. In diesem Zusammenhang werden wir im Kapitel 3.2 auf *Extensions* und *Extension-Points* zu sprechen kommen.
Erst bei Bedarf, also wenn das Plugin angefordert wird, sei es direkt oder als Abhängigkeit eines zweiten Plugins, werden die Java-Klassen gelesen und ausgeführt. Dieses Vorgehen wird als *Lazy Loading* bezeichnet und zieht sich gemeinsam mit dem Konzept des Plugins durch das komplette Eclipse SDK.

3.1 Manifest Dateien

Man unterscheidet seit der OSGi Implementierung von Eclipse zwischen zwei unterschiedlichen Arten von Manifest-Dateien:

- Eclipse Platform Manifest
- OSGi Bundle Manifest

Letzteres beschreibt das Plugin und dessen Verhalten in Bezug auf die Umgebung in der es eingefügt werden soll. Dazu gehört die Versions-Nummer des Plugins und der Manifest-Spezifikation, exportierte bzw. benötigte Bundles u.ä. Ein Beispiel für solch ein Manifest ist in Listing 1 gegeben.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Coco/R Builder
Bundle-SymbolicName: at.ssw.coco.builder;singleton:=true
Bundle-Version: 0.0.6
Bundle-Activator: at.ssw.coco.builder.BuilderActivator
Bundle-Localization: plugin
Require-Bundle: org.eclipse.ui,
 org.eclipse.core.runtime,
 at.ssw.coco.core,
 org.eclipse.core.resources,
 org.eclipse.ui.console,
 org.eclipse.jdt.core
Eclipse-LazyStart: true
Export-Package: at.ssw.coco.builder.nature
Bundle-Vendor: ssw.jku.at/ madebywR.org
```

Listing 1. OSGi Bundle Manifest Datei am Beispiel des `at.ssw.coco.builder` Plugins

Der Umfang der Eclipse Platform Manifest Dateien hat sich mit dem OSGi Framework dementsprechend reduziert, sodass in der aktuellen Version darin nur noch die Extensions und Extension-Points (siehe Kapitel 3.2) zu finden sind.
In Form und Struktur unterscheiden sich hier Plugins nicht von Fragmenten, lediglich das „Root-Keyword“ ist ein anderes. Listing 2 zeigt anhand der offiziellen *Document Type Definition* (DTD) den genauen Aufbau eines Plugin bzw. Fragment Manifests.

```
<?xml encoding="US-ASCII"?>
<?eclipse version="3.2"?>

<!ELEMENT plugin (extension-point*, extension*)>

<!ELEMENT fragment (extension-point*, extension*)>

<!ELEMENT extension-point EMPTY>
<!ATTLIST extension-point
 name                  CDATA #REQUIRED
 id                    CDATA #REQUIRED
 schema                CDATA #IMPLIED
>

<!ELEMENT extension ANY>
<!ATTLIST extension
 point                 CDATA #REQUIRED
 name                  CDATA #IMPLIED
 id                    CDATA #IMPLIED
>
```

Listing 2. Die *Document Type Definition* für Plugin bzw. Fragment Beschreibungen. (Die Daten der plugin.dtd wurden [Eclipse Foundation, 2006c] entnommen.)

Der Aufbau des Feature Manifests, wie in Listing 3 dargestellt, ist, aufgrund der komplett unterschiedlichen Anforderungen, selbstverständlich anders. Zunächst wird das Feature selbst betitelt und beschrieben, danach folgen 3 unstrukturierte Blöcke: Beschreibung, Copyright- und Lizenzvermerk. Weiters enthält eine `feature.xml` die *Requirements* und zumindest einen Verweis auf ein Plugin mit den zumindest den geforderten Metainformationen (*id, version*).

```
<?xml version="1.0" encoding="UTF-8"?>
<feature
      id="at.ssw.coco"
      label="Coco\R Eclipse Plugin"
      version="0.1"
      provider-name="ssw.jku.at/ madebywR.org">

  <description url="http://description"> the description </description>

  <copyright url="http://copyright"> the copyright text </copyright>

  <license url="http://license"> the license text </license>

  <requires>
    <import plugin="org.eclipse.ui"/>
    <!-- more requirements here -->
  </requires>

  <plugin
      id="at.ssw.coco.builder"
      download-size="0"
      install-size="0"
      version="0.0.4"
      unpack="false"/>

  <!-- more plugin references here -->
</feature>
```

Listing 3. Ein minimales Eclipse Platform Feature Manifest

3.2 Extensions

Unter einer Extension versteht man eine Erweiterung eines Plugins um zusätzliche Elemente bzw. Funktionen. Das beste Beispiel ist hier das Eclipse Workbench Plugin (**org.eclipse.ui**), wobei das Konzept natürlich nicht auf das User Interface beschränkt ist. Jedes Plugin kann sogenannte *Extension Points* definieren, welche als Verbindungs-Schnittstelle zur Extension dienen.

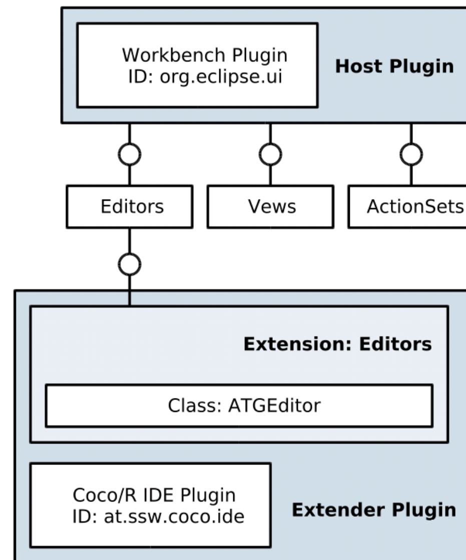


Abbildung 5. Teilnehmer einer Plugin Extension. Hier wird das Workbench UI um einen Editor erweitert.

Für den weiteren Verlauf sollten zunächst noch einige Begrifflichkeiten geklärt werden. „Extension" und „Extension Points" gehören zur Standard-Terminologie, wenn man sich mit Eclipse auseinandersetzt. Für eine bessere Verständlichkeit werde in den kommenden Ausführungen analog zu [Bolour, 2003] folgende Begriffe verwendet:

– *Host Plugin.* Jenes Plugin, dass einen oder mehrere Extension Points zur Verfügung stellt.

– *Extender Plugin.* Die Extension an sich, also das Plugin, welches die Extension Points des Host Plugins verwendet.

– *Callback Object.* Das Objekt, welches zur Kommunikation zwischen den beiden Plugins dient.

In Abbildung 5 sind alle diese Komponenten einer Extension präsent, um deren Zusammenspiel zu verdeutlichen. In diesem Fall wird die Eclipse Workbench

erweitert, von der hier allerdings lediglich 3 der häufigsten Extension-Points angeführt wurden. Das Plugin `org.eclipse.ui` stellt also das *Host Plugin* dar und wird durch das *Extender Plugin* `at.ssw.coco.ide` um einen Editor für ATG Dateien ergänzt. Das *Callback Object* der Extension wird hierbei durch die Klasse `ATGEditor` implementiert.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
  <!-- Workbench extension points -->
  <extension-point id="editors" name="%ExtPoint.editors"
                   schema="schema/editors.exsd"/>
  <extension-point id="views" name="%ExtPoint.views"
                   schema="schema/views.exsd"/>
  <extension-point id="actionSets" name="%ExtPoint.actionSets"
                   schema="schema/actionSets.exsd"/>

   <!-- more extension point declarations here -->
</feature>
```

Listing 4. Ausschnitt aus dem Manifest von `org.eclipse.ui`

In den Manifest Dateien, welche in Kapitel 3.1 beschrieben wurden, äußert sich dies auf der Seite des Host Plugins durch die Definition der Extension Points in Form eines einfachen XML Tags (siehe Listing 4).
Auf Seiten des Extender Plugins fällt dies durch Details wie eine Beschreibung, Icons oder einem Tooltip-Text etwas ausführlich aus. (Listing 5)

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
  <extension
       id="ATGEditor"
       name="ATG Editor"
       point="org.eclipse.ui.editors">
    <editor
         name="ATG Editor"
         extensions="atg"
         icon="icons/file.gif"
         contributorClass=
           "org.eclipse.ui.texteditor.BasicTextEditorActionContributor"
         class="at.ssw.coco.ide.editor.ATGEditor"
         id="padre.editors.ATGEditor">
    </editor>
  </extension>

  <!-- more extensions here -->
<\plugin>
```

Listing 5. Ausschnitt aus dem Manifest von `org.eclipse.ui`

Callback Objekte sind im Gegensatz zu den beiden anderen Komponenten einfache Java Klassen, welche ein vom Host Plugin vorgegebenes Interface implementieren. Die „Callback Klasse“ selbst ist aber Teil des Extender Plugins und wird bei Auftreten eines bestimmten Ereignisses vom Host Plugin aufgerufen.

Auf die unterschiedlichen Extensions und Extension-Points, welche im Rahmen der Implementierung der pratischen Arbeit verwendet wurde, wird in den folgenden Kapiteln eingegangen, wenn es darum geht, das Coco/R Plugin zu dokumentieren.
Für einen tieferen Einblick, vor allem in die programmatische Verarbeitung von Extensions, möchte ich besonders [Bolour, 2003] empfehlen.

3.3 OSGi Framework

Das OSGi Framework ist Teil der von der *Open Services Gateway Initiative* entwickelten OSGi Service Platform. Dabei handelt es sich um eine auf Java aufsetzende, *„standardiesierte, komponenten-orientierte Umgebung für vernetzte Dienste"*. Ähnlich wie z.B. auch bei der J2EE-Spezifikation wird ein Rahmen aus Richtlinien und Schnittstellen bereitgestellt, um eine variabel konfigurierbare Umgebung aus Diensten zu schaffen.
Speziell das Hinzufügen bzw. Entfernen von Komponenten zur Laufzeit, ohne das System anzuhalten und neu zu starten, steht hier im Vordergrund. Die OSGi weist hierbei vor allem in die Richtung Mobile and Embedded Devices, das Framework findet allerdings auch Anwendung auf Desktops (wie es Eclipse beweist) und High-End Servern.

Das Framework setzt sich aus folgenden Ebenen zusammen:

- *Execution Environment.* Die Java VM bzw. allgemein eine Java Umgebung, auf welcher das Framework aufsetzt.
- *Modules.* Das OSGi Framework erweitert die Möglichkeiten des *Execution Environments* um Modularisierung und ermöglicht so die Entwicklung verteilter, mehrschichtiger Anwendungen. Auf dieser Ebene werden also die Richtlinien zum Laden der einzelnen Programmteile (Ressourcen, Klassen, etc) festgelegt.
- *Life Cycle Managment.* Entspricht der dynamischen Verwaltung der einzelnen Bundles. (De-/Installieren, Aktualisieren, Starten, Stoppen der Bundles)
- *Service Registry.* Die Registry verwaltet nun unter Beachtung der dynamischen Anforderung die Existenz, Interaktion, etc. der Bundles im System und untereinander. In einer statischen Umgebung entspricht dies dem Design Pattern Listener.

Eine weitere Schicht, das *Security System*, umspannt diese 4 Ebenen, um eine sichere Kommunikation zwischen den Bundles und eine feinere Zugriffskontrolle auf selbige und deren Ressourcen zu ermöglichen.

4 Coco/R für Java

Coco/R erzeugt anhand einer **attributierten Grammatik** einen als endlichen Automaten implementierten Scanner und einen nach rekursivem Abstieg arbeitenden LL(1) Parser.

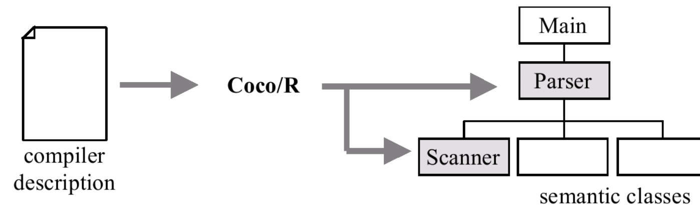


Abbildung 6. Ein- und Ausgabe von Coco/R. (Die Abbildung stammt aus [Mössenböck, 2005])

Die Beschreibung des zu erstellenden Compilers befindet sich in einer sogenannten ATG-Datei, dessen Endung bzw. Bezeichnung das Akronym für „Attributed Grammar“ darstellt.
Listing 6 zeigt die Grundstruktur einer solchen Datei, wobei man sich bei der Beschreibung des Compilers der Extended Backus-Naur Form (EBNF – siehe [Wirth, 1977]) bedient hat. Dementsprechend werden Terminalsymbole klein und Nonterminalsymbole groß geschrieben.
Die Platzhalter *[Imports]* und *[GlobalFieldsAndMethods]* stehen in diesem Fall für beliebigen Java Code bzw. allgemein für eine beliebige, unterstützte Sprache. Wie in der Einleitung erwähnt wurde, existieren bis heute mehr als 10 Varianten für unterschiedlichste Programmiersprachen. Diese Platzhalter sind jedoch, wie die Namen bereits ausdrücken, für importierte Klassen/Pakete (z.B. `import java.io.*;`) und globale Felder oder Methoden vorgesehen.

```
Coco =
  [Imports]
  "COMPILER" ident
  [GlobalFieldsAndMethods]

  /* Scanner Specification */
  ["IGNORECASE"]
  ["CHARACTERS" {SetDecl}]
  ["TOKENS" {TokenDecl}]
  ["PRAGMAS" {PragmaDecl}]
  {CommentDecl}
  {WhiteSpaceDecl}.

  /* Parser Specification */
  "PRODUCTIONS" {Production}.

  "END" ident '.'
  .
```

Listing 6. Struktur einer ATG Datei

Weiters ist darauf zu achten, dass die Identifier nach den Schlüsselwörtern **COMPILER** und **END** identisch sind, da diese den Namen des Compilers bzw. der dazugehörigen Methode angeben.
Mit Listing 7 ist hier nocheinmal ein kleiner Einblick gegeben, wie eine solche Compiler Beschreibung in dessen Grundzügen aussehen kann. Ein wesentlich ausführlicheres Beispiel liegt den auf der Projekt Homepage des Coco/R erhältlichen Source Distributionen bei. Dabei handelt es sich um die Grammatik für Coco/R selbst.
Für einen detailliertere Einführung und Erklärung der Hintergründe, sowie der Bedienung von Coco/R wird an dieser Stelle an [Mössenböck, 2005] verwiese, welches auch für dieses Kapitel als Quelle diente.

```
COMPILER Coco

  /* public/protected/private fields */

/*----------------------------------------------------------------*/

CHARACTERS
  letter    = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".
  digit     = "0123456789".
  cr        = '\r'.
  lf        = '\n'.
  tab       = '\t'.
  stringCh  = ANY - '"' - '\\' - cr - lf.
  charCh    = ANY - '\'' - '\\' - cr - lf.
  printable = '\u0020' .. '\u007e'.
  hex       = "0123456789abcdef".

TOKENS
  ident     = letter { letter | digit }.
  number    = digit { digit }.
  string    = '"' { stringCh | '\\' printable } '"'.
  char      = '\'' ( charCh | '\\' printable { hex } ) '\''.

COMMENTS FROM "/*" TO "*/" NESTED
COMMENTS FROM "//" TO cr lf

IGNORE cr + lf + tab

/*----------------------------------------------------------------*/

PRODUCTIONS

Coco
=
  "Coco/R" "Development" "Tools"         (. System.out.println(t.val); .)
.

END Coco.
```

Listing 7. Beispiel ATG, wie sie z.B. als Template im Plugin `at.ssw.coco.core` verwendet wird.

5 Das Coco/R Plugin

In diesem Kapitel werden nun die Interna des Coco/R Plugins präsentiert und vor allem die Verbindungen zur Integration, speziell auch in Bezug auf die Extension Points, hergestellt.

5.1 Überblick

Das Coco/R Plugin bzw. das implementierte Feature, besteht, exkl. der Projekte des Features selbst und der Update Site, aus 4 Teilen bzw. Plugins:

- *at.ssw.coco.branding.* Das Branding Plugin dient dazu, das Projekt „als Ganzes“ zu beschreiben und es vor allem in der Eclipse SDK als Feature zu repräsentieren.

- *at.ssw.coco.builder.* Getrennt vom User Interface und dem Compiler Generator selbst, wird der Umgang mit Coco/R bzw. dessen Integration in die IDE gehandhabt.

- *at.ssw.coco.core.* Dieses Plugin umfasst alle Coco/R spezifischen Daten, sprich alles was von dem „Dritt-Anbieter“ bereitgestellt wird und für die Integration in die Eclipes SDK von Nöten ist. Dies umfasst neben dem JAR-Archiv für den Generator selbst, vor allem die Frame-Dateien (das Grundgerüst für den zu erstellenden Scanner und Parser), sowie das Template für ein neues Coco/R unterstütztes Projekt. Dies soll das Updaten des Coco/R erleichtern und von der restlichen Funktionalität physisch sowie logisch trennen.

- *at.ssw.coco.ide.* Den größten Teil des Features stellt die IDE dar, welche das User Interface für eine einfache Handhabbarkeit des Coco/R zur Verfügung stellt. Darunter befinden sich neben dem Editor diverse Eclipse Views und vor allem das dahinter stehende Datenmodell des Compilers.

In der folgenden Ausführung wird lediglich auf 2 dieser 4 Plugins näher eingegangen, welche den überwiegenden Teil der Zeit der Implementierung in Anspruch genommen haben. Dabei handelt es sich zum Einen um *at.ssw.coco.builder*, welches, wie der Name schon sagt, den *Builder* und die dazugehörige *Nature* enthält (mehr dazu im Kapitel 5.2) und zum Anderen die IDE (Kapitel 5.3), also *at.ssw.coco.ide*.

5.2 Builder und Nature

Der Sinn eines *Project Natures* ist es, ein gegebenes Projekt an eine gewissen Funktionalität, ein externes Programm, ein Plugin einen Builder o.ä. zu binden bzw. es zu markieren. Im Fall des Coco/R Plugins wird so die Verbindung über das Plugin, *at.ssw.coco.builder* und dem dort implementierten Builder zu Coco/R hergestellt.

Es gibt zwei unterschiedliche Arten von Natures bzw. Randbedingungen, welche das Verhalten eines Natures beeinflussen:

- *one-of.* Mit dieser Bedingung wird sichergestellt, dass lediglich ein Nature der „selben Art“ für ein Projekt gesetzt werden kann. Mit „Art“ ist hier die Zugehörigkeit zu einem *Meta-Typ* gemeint, damit zwei oder mehrere Natures sich nicht stören, welche die selbe Aufgabe unterschiedlich erfüllen (z.B. die Funktionalität des gleichen Compiler Generators zur Verfügung zu stellen) und sich so gegenseitig beeinflussen würden.

- *require.* Hiermit wird bezweckt, dass eine Nature nur gesetzt werden kann, wenn die Nature, welche als „required“ angegeben wurde, bereits im Projekt vorhanden ist.

Die Deklaration einer solchen Nature ist mit Listing 8 gegeben.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
   <extension
      id="CocoNature"
      name="Coco/R Nature"
      point="org.eclipse.core.resources.natures">
      <runtime>
         <run class="at.ssw.coco.builder.nature.CocoNature"/>
      </runtime>
      <builder id="at.ssw.coco.builder.CocoBuilder"/>
   </extension>
   <extension id="CocoBuilder" name="Coco Builder"
      point="org.eclipse.core.resources.builders">
      <builder
         hasNature="true"
         isConfigurable="true">
         <run class="at.ssw.coco.builder.CocoBuilder">
            <parameter name="optimize" value="true" />
            <parameter name="comment"
               value="Builds the Coco/R Parser &amp; Scanner" />
         </run>
      </builder>
   </extension>
</plugin>
```

Listing 8. Eclipse Manifest von at.ssw.coco.builder

Wenn eine Nature einem Projekt hinzugefügt wird, wird dessen `configure()` Methode aufgerufen, was ein gewisses Life-Cycle Management zulässt. Ein typischer Anwendungsfall, welcher auch bei diesem Plugin zu tragen kommt, ist es, den dazugehörigen Builder zu setzen. Wichtig ist, dass innerhalb der Extension für die Nature der Builder mit dessen ID angeführt wird (siehe Listing 8), da dieser sonst nicht aufgerufen werden kann. Auch für das Entfernen eines Natures ist eine entsprechende Methode vorgesehen: `deconfigure()`.
Ein *Builder* ist jene Klasse, welche aufgerufen wird, sobald sich eine Ressource geändert hat, um diese Änderungen entsprechend umzusetzen. Im Fall des

Coco/R Plugins bedeutet dies, den passenden Scanner und Parser zu der attributierten Grammatik zu generieren.
Das Eclipse IDE Framework unterscheidet zwischen 3 Arten von Builds:

– *Incremental build.* Hierbei geht es darum, immer nur jene Teile eines Projekts zu verarbeiten, welche auch wirklich eine Änderung erfahren haben. Hierzu wird dem Builder ein sogenanntes *Resource Delta* übergeben, welches eine hierarchische Beschreibung aller Änderungen des Projekts enthält. Ein solches *Incremental build* wird durch einen expliziten Aufruf (`Build All` oder `Build Project`) getriggert.

– *Auto build.* Dies unterscheidet sich von dem *Incremental build* nur in der Art des Aufrufs, welcher hier, automatisch, also ohne Zutun des Benutzers, erfolgt.

– *Full build.* In diesen Fall werden alle betroffenen Ressourcen bearbeitet, unabhängig davon, ob diese im Vergleich zum letzten Durchlauf verändert wurden oder nicht. Eine solche Situation tritt auf, wenn z.B. auf Grund von Performanz-Überlegungen (Kosten der Speicherung im Verhältnis zu den Kosten für das Durchführen eines *Full build*s) oder durch einen Aufruf mit dem Parameter `clean` das *Resource Delta* gelöscht wurde.

Für jede dieser Build-Varianten muss für gewöhnlich ein `BuildVisitor` (vgl. Design Pattern Visitor) implementiert werden. Je nach Anforderung des Builds wird hierfür eines der folgenden beiden Interfaces verwendet, welche die Art der Übergabe von Resource-Daten regeln: `IResourceVisitor` und `IResourceDeltaVisitor`.
Die Implementierung des Coco/R Plugins unterscheidet jedoch nicht zwischen der Verarbeitung der Ressourcen der unterschiedlichen „Build"-Varianten. Der Grund hierfür ist, dass ATG Dateien eigenständige Beschreibungen eines Compilers darstellen und somit keine Verknüpfungen zu anderen Dateien berücksichtigt werden müssen. Insofern teilen sich alle den selben `BuildVisitor`, um die vorhandenen Daten zu verarbeiten, wobei im Fall eines *Incremental Build*s das *Resource Delta* auf eine einfach Ressource abgebildet wird.
Neben dem reinen Übermitteln der Dateien, welche für einen weiterhin aktuellen „Source-Tree" verarbeitet werden müssen, und den entsprechenden Parametern für den Aufruf des Compiler Generators, ist der Builder in diesem Fall außerdem für den Output verantwortlich. Dies bedeutet hier, die eventuellen Fehlermeldungen des Coco/R in entsprechende Marker (`IMarker.PROBLEM`) umzusetzen. Mehr zur Integration des Coco/R ist im Kapitel 5.4 nachzulesen.

Dieses Kapitel hat sich vor allem der Information aus [Arthorne, 2004], [Freier, 2005] und [Friese, 2004] bedient.

5.3 Integrated Development Environment (IDE)

Den eindeutig komplexeren und umfangreicheren Teil des Coco/R Plugins stellt die IDE dar, welche sich im wesentlichen auf folgende Elemente beschränkt:

- ATG Modell
- Editor
- Content Outline View
- Coco/R Extension Wizard

ATG Modell. Für das Modell der attributierten Grammatik wurde intern wiederum Coco/R verwendet. Dies erforderte allerdings einige Änderungen an den Frame Dateien, da diese so wie sie dem Coco/R beiliegen in erste Linie dafür ausgelegt sind, eigenständige Scanner/Parser Paare zu generieren. So war es z.B. nötig den verwendeten Zeichen-Buffer so anzupassen, dass dieser mit dem von der Eclipse SDK verwendeten Datenmodell zurecht kommt.
Im Verlauf der Entwicklung dieses Plugins konnten einige Ecken, welche eine Integration erschwerten, in Zusammenarbeit mit den Betreuern des Coco/R Projekts beseitigt werden.
Mehr bzgl. der Problematik, Coco/R zu integrieren, sowohl in die Eclipse SDK wie auch in andere Programme ist im Kapitel 5.4 nachzulesen.

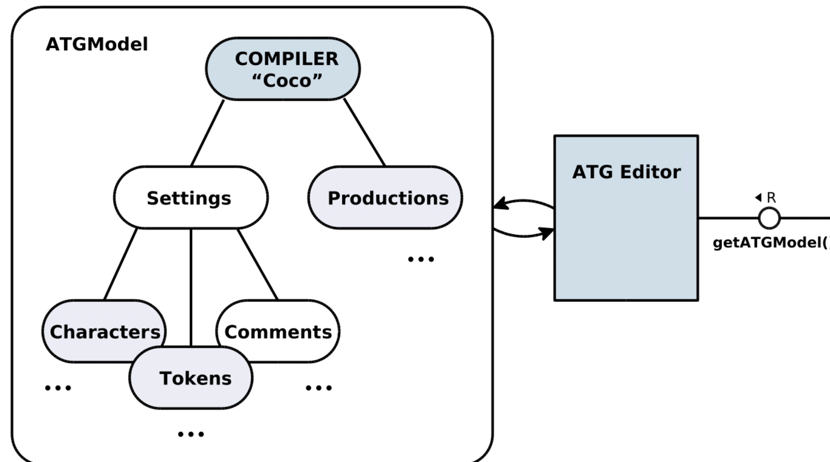


Abbildung 7. ATG Modell

Die Repräsentation der ATG selbst wurde als Baum, wie er in Abbildung 7 unter anderem dargestellt ist, realisiert. Dieses Modell wird sowohl für die *Content Outline View*, als auch die erweiterten Funktionen des Editors, wie *Folding*, *Link Navigation*, etc. benötigt.

Editor. Wenn zuvor von der Komplexität der IDE gesprochen wurde, so bezog sich dies vor allem auf den Editor. Dieser bietet unterschiedlichste Funktionalitäten, um die Handhab- und Bearbeitbarkeit der attributierten Grammatiken zu verbessern:

- Syntax-Highlighting
- Folding
- Hyperlink Navigation
- Keyword Completion / Content Assistant
- sowie etliche kleinere Beiträge

Um all diese Merkmale zu implementieren, muss in der Editor Klasse zunächste die `SourceViewerConfiguration` gesetzt werden, welche in Form von überschriebenen Methoden die angepassten und konfigurierten Einzelteile für die Gestaltung zurückgeben.

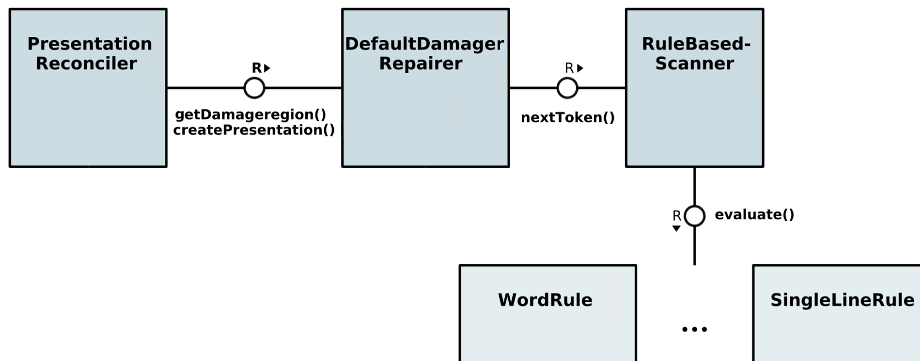


Abbildung 8. Syntax-Highlighting.

Für das *Syntax-Highlighting*, dessen strukturellen Zusammenhang in Abbildung 8 dargestellt ist, wird ein `PresentationReconciler` konfiguriert und zurückgegeben. Ein solcher Reconciler (engl. to reconcile: wieder zusammenbringen, in Einklang bringen) besteht aus einem *Damager* und einem *Repairer*, welche gemeinsam dafür sorgen, dass die Farbgebung des Syntax-Highlightings auch nach einer Veränderung des Textes korrekt gegeben ist. Der Damager ermittelt die geänderten Textpositionen und der Repairer erneuert die Farbgebung in diesen Regionen.
In Listing 9 ist jener Teil der `SourceViewerConfiguration` des ATG Editors beschrieben. Man kann erkennen, dass die Default-Implementierung einer Damager-Repairer Kombination ein Scanner, welcher die einzelnen Teil der attributierten Grammatik erkennt und betitelt, übergeben wird.

```
public IPresentationReconciler
getPresentationReconciler(ISourceViewer sourceViewer) {

  PresentationReconciler reconciler = new PresentationReconciler();

  DefaultDamagerRepairer dr;
  dr = new DefaultDamagerRepairer(getATGScanner());
  reconciler.setDamager(dr, IDocument.DEFAULT_CONTENT_TYPE);
  reconciler.setRepairer(dr, IDocument.DEFAULT_CONTENT_TYPE);

  dr = new DefaultDamagerRepairer(getCodeScanner());
  reconciler.setDamager(dr, ATGPartitionScanner.ATG_INLINE_CODE);
  reconciler.setRepairer(dr, ATGPartitionScanner.ATG_INLINE_CODE);

  // ...

  return reconciler;
}
```

Listing 9. Ausschnitt der SourceViewerConfiguration des ATG Editors: `IPresentationReconciler getPresentationReconciler(ISourceViewer)`

Für den Scanner der ATG Struktur musste etwas tiefer in dessen Arbeitsweise eingegriffen werden. Dies lag daran, dass vor allem der erste Teil einer attributierten Grammatik, wie sie Coco/R verwendet (globaler Java Quellcode), der Unterstützung durch Syntax-Highlighting bedarf, aber keinen fix definierten Endpunkt besitzt (vgl. Listing 6 auf Seite 12). Diese Region kann sich, je nach Präsenz der einzelnen Scanner-Definitionen, vom „Header" des Compiler (`COMPILER ident`) bis maximal zum Beginn der Produktionen für die einzelnen nonterminal Symbole erstrecken.
Der Partitions-Scanner setzt zwar ebenfalls auf einen `RuleBasedScanner`, wie er auch in Abbildung 8 angedeutet wurde, allerdings muss die Ausgabe der Token ein weiteres Mal, direkt in der Mehtode `nextToken()`, gefiltert werden, um die oben beschriebene Funktionalität zu gewährleisten. Entsprechend der Änderungen, welche eine Anpassung des Syntax-Highlightings nötig machen, muss das zuvor angesprochene Modell der attributierten Grammatik erneuert werden.

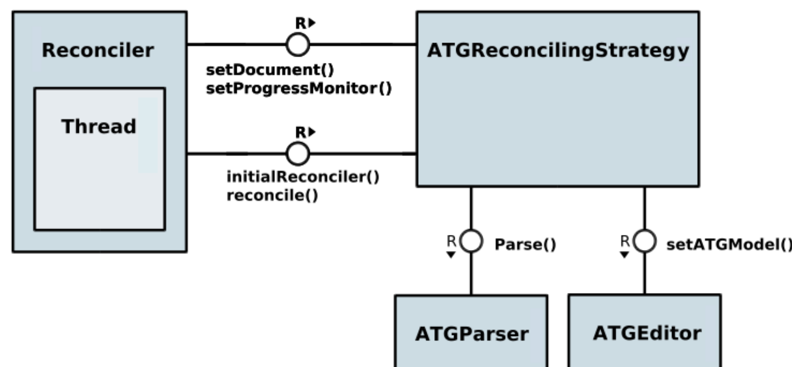


Abbildung 9. Abgleich des Modells der attributierten Grammatik.

Hierzu wird ein weiterer Reconciler implementiert, welcher sich der vom Text-Framework gebotenen „Reconciler-Infrastruktur“ bedient. Diese macht es möglich, das zeitaufwändige Parsen der ATG Struktur bzw. das Erneuern der modellabhängigen Teile der IDE, in einem separaten Hintergrund-Thread zu erledigen. Um die Effizienz weiter zu steigern, startet dieser Thread erst, wenn eine bestimmte Zeit lang keine Änderung an dem Dokument vorgenommen wurde.
Dies verhindert, dass pro Tastendruck ein neuer Durchlauf gestartet wird, die Änderung für den Betrachter bzw. Nutzer der IDE aber dennoch subjektiv gleichzeitig in jedem Element übernommen werden kann. Abbildung 9 zeigt den Zusammenhang der Reconciler-Infrastruktur und dem ATG Modell.
Das zentrale Element dieses Gebildes stellt die *ReconcilerStrategy* dar, welche in diesem Fall, die beiden Interfaces `IReconcilingStrategy` und `IReconcilingStrategyExtension` implementiert. Das Extension-Interface erweitert hierbei die unterschiedlichen reconcile-Schnittstellen um die Möglichkeit, einen *Progressmonitor* zu setzen bzw. vor jeder Dokumentänderung den Reconciler zu initialisieren.

```
private void reconcile() {
  fATGModel = new ATG(fDocument);
  if (fATGModel == null) {
      return;
  }
  Shell shell = fEditor.getSite().getShell();
  shell.getDisplay().asyncExec(new Runnable() {
    public void run() {
      fEditor.setATGModel(fATGModel);
    }
  });
  fFoldingStructureProvider
    .updateFoldingRegions(fATGModel.getElements());
}
```

Listing 10. Ausschnitt der ReconcilerStrategy des ATG Editors:
`void reconcile()`

Das Coco/R Plugin macht davon allerdings nur begrenzt Gebrauch und ruft sowohl bei einem initialen als auch jedem weiteren Aufruf die selbe private Methode auf, welche das Erstellen des ATG Modells übernimmt (siehe Listing 10). Angewandt wird diese *ReconcilerStrategy* auf einen `MonoReconciler`, der, seinem Name entsprechend, nur **eine** solche Strategie umsetzen kann. Dieser wird, wenn angefordert, in der *SourceViewerConfiguration* des ATG Editors erzeugt und zur Verfügung gestellt. (siehe Listing 11)

```
public IReconciler getReconciler(ISourceViewer sourceViewer) {
  ATGReconcilerStrategy strategy = new ATGReconcilerStrategy(fEditor);
  MonoReconciler reconciler = new MonoReconciler(strategy, false);
  reconciler.setProgressMonitor(new NullProgressMonitor());
  reconciler.setDelay(500);
  return reconciler;
}
```

Listing 11. Ausschnitt der SourceViewerConfiguration des ATG Editors:
`IReconciler getReconciler(ISourceViewer)`

Wie in Listing 10 bereits zu sehen war, werden bei der gleichen Gelegenheit auch die Regionen und Segmente der Folding Datenstruktur erneuert.
Folding dient, dazu mehrzeilige Fragmente eines Textes, welche einen thematisch abgeschlossen Block bilden, **optional** zu einer Zeile zusammenzufassen. Um dies zu erreichen, wird die Mehtode `createPartControl` des Editors überschrieben, damit diese die Support Klasse des „Projection Viewers“ erzeugt und installiert. (Siehe Listing 12)

```
public void createPartControl(Composite parent) {
  super.createPartControl(parent);

  ProjectionViewer projectionViewer;
  projectionViewer = (ProjectionViewer)getSourceViewer();

  fProjectionSupport =
    new ProjectionSupport(
      projectionViewer,
      getAnnotationAccess(),
      getSharedColors()
    );

  fProjectionSupport.install();
  projectionViewer.doOperation(ProjectionViewer.TOGGLE);
}
```

Listing 12. Erstellen des „Projection Supports“ um Folding im ATG Editor zu ermöglichen: `void createPartControl(Composite)`

Ein `ProjectionViewer` stellt die JFace Implementierung eines Anzeige-Elements für einen beliebigen Text, in den meisten Fällen, wie auch in diesem, allerdings Quellcode, dar. Im Gegensatz zu anderen Klassen dieses Typs unterstützt dieser jedoch mehrere sichtbare Ebenen (Projektionen), welche unter anderem dazu verwendet werden können, um Folding zu realisieren.
Hierzu muss das `AnnotationModel` des `ProjectionViewer`s bearbeitet bzw. erstellt werden. Für den ATG Editor des Coco/R Plugins wird jedoch bei jeder Änderung der Struktur dieses Modell komplett neu aufgebaut, anstatt die Möglichkeit zu nutzen, es immer wieder nach, durch die Bearbeitung des Text-Dokuments „zerstörten“ Regionen zu durchsuchen und diese neu zu erstellen. Ähnlich wurde dies auch schon bei den Partitionen des Syntax-Highlightings gelöst, was in diesen Maßstäben für die Performanz keine gravierende Nachteile bewirkt.

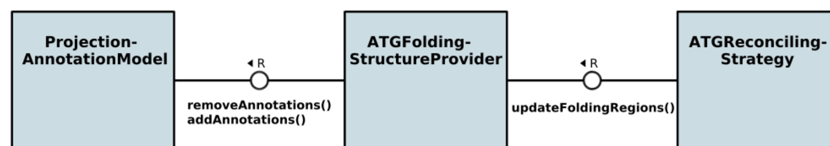


Abbildung 10. Berarbeiten der Regionen/Segmente für die Folding-Funktionalität von dem Reconciler aus.

Der Zusammenhang des Reconcilers bzw. der `ReconcilerStrategy` und dem `AnnotationModel` für das Folding wurde nochmals in Abbildung 10 dargestellt. Verhältnismäßig unkompliziert stellt sich der Zusammenhang der Komponenten für die *Hyperlink Navigation* und den *Content Assistent* dar.
Hierzu werden wiederum in der *SourceViewerConfiguration* die dafür vorgesehenen Methoden überschrieben (siehe Listing 13), und somit diese beiden Funktionen konfiguriert.

```
public IContentAssistant
getContentAssistant(ISourceViewer sourceViewer) {

  ContentAssistant assistant = new ContentAssistant();
  IContentAssistProcessor processor =
      new ATGContentAssistProcessor(fEditor);

  assistant.setContentAssistProcessor(
    processor,
    IDocument.DEFAULT_CONTENT_TYPE
  );
  return assistant;
}


public IHyperlinkDetector[]
getHyperlinkDetectors(ISourceViewer sourceViewer) {

  if (fHyperlinkDetectors == null) {
    fHyperlinkDetectors =
        new IHyperlinkDetector[] { new ATGHyperlinkDetector(fEditor) };
  }
  return fHyperlinkDetectors;
}
```

Listing 13. Ausschnitt der SourceViewerConfiguration des ATG Editors:
`IContentAssistant getContentAssistant(ISourceViewer)`

Alles was z.B. für die *Hyperlink Navigation* von Nöten ist, ist die Angabe eines Tupels aus Namen und Ziel des Links, für eine der Funktion übergebenen Cursor Position. Während für den Namen das Text-Dokument, welches die attributierte Grammatik darstellt, herangezogen wird, ermittelt man die Ziel Position (wohin der Cursor durch den Link springen soll) anhand des ATG Modells, welches die entsprechenden Positionen bereits verwaltet.
Ebenso dient das ATG Modell dem *Content Assistent*, der wie auch der `HyperlinkDetector` über eine Funkion die Cursor Position übermittelt bekommt. Um den entsprechenden Eintrag eines Keywords bzw. des Namens einer Produktion im Modell suchen zu können, muss zunächst der bereits getippte Prefix des Wortes ermittelt werden. Die Suche selbst gestaltet sich als einfaches, rekursives Traversieren des Baums, welcher die attributierte Grammatik intern darstellt (vgl. Abbildung 7 auf Seite 17).

Content Outline View. Wie so viele Elemente des User Interfaces der Eclipse SDK basiert auch die *Content Outline View* auf einem *Tree Viewer* des JFace Toolkits. Darum sind viele der hier angesprochenen Vorgehensweisen dementsprechend allgemein auf diese Komponente anwendbar.

Für diese View existiert bereits ein vorgefertigtes Template, welches es zu erweitern gilt: `ContentOutlinePage`. Die Verbindung zum User Interface wird über die Methode `getAdapter` des ATG Editors erreicht, und setzt so das Prinzip der losen Bindung innerhalb der Eclipse SDK fort. (siehe Listing 14)

```
public Object getAdapter(Class required) {
  if (IContentOutlinePage.class.equals(required)) {
      if (fOutlinePage == null) {
    fOutlinePage = new ATGContentOutlinePage(this);
    if (getEditorInput() != null)
       fOutlinePage.setInput(getEditorInput());
      }
      return fOutlinePage;
  }

  // ...

  return super.getAdapter(required);
}
```

Listing 14. Ausschnitt der Konfiguration der Adapter des ATG Editors: `Object getAdapter(Class)`

Die abgeleitete *Content Outline Page* implementiert nun den zuvor erwähnten *Tree Viewer*, für den man zumindest einen `ContentProvider` angeben muss, um diesen zu befüllen. Soll auch dessen Aussehen verändert bzw. angepasst werden (Icons an den Verzweigungen des Baums, etc), ist ein `LabelProvider` zu setzen. (siehe Listing 15)

```
public void createControl(Composite parent) {
  super.createControl(parent);

  TreeViewer viewer = getTreeViewer();
  viewer.setContentProvider(new ATGContentProvider());
  viewer.setLabelProvider(new ATGLabelProvider());
  viewer.setAutoExpandLevel(AbstractTreeViewer.ALL_LEVELS);
  viewer.addSelectionChangedListener(this);

  setTreeViewerInput();
}

private void setTreeViewerInput() {
  getTreeViewer().setInput(fEditor.getATGModel());
}
```

Listing 15. Ausschnitt der Content Outline Page: Erstellen und Konfigurieren des *TreeViewers*: `void createControl(Composite)`

Sinn und Zweck einer solchen *Content Outline View* ist es, den Inhalt jener Datei, welche gerade bearbeitet wird, anzuzeigen. Da diese, wie zuvor schon beschrieben, bereits im ATG Modell verarbeitet und hinterlegt ist, müssen sich die beiden „Provider-Klassen“ an dieser Stelle nur noch aus dem vorhandenen Bestand bedienen.

Coco/R Extension Wizard. Um einen Wizard dieser Art zu erstellen, werden wiederum *Extension-Points* erweitert:

– *org.eclipse.ui.newWizards*. Für die Integration des ‚New'-Wizard inkl. dessen Eintrag in die Kategorie der neu erstellbaren Projekte.

– *org.eclipse.ui.actionSet*. Um den Wizard von der Toolbar aus aufrufen zu können.

Ein *Extension Point* dient also, wie auch in den vorherigen Kapiteln erwähnt, der Integration in das vorhandene Framework. Somit ist die Extension von `org.eclipse.ui.newWizards` hier auch **nicht** für den eigentlichen Wizard zuständig, sondern nur für die Art und Weise dessen Aufrufs.
Erstellt wird der Wizard mit der Klasse `at.ssw.coco.ide.wizard.CocoWizard`, welche für diesen Zweck im Plugin Manifest vermerkt wurde (siehe Listing 16, XPath: `/plugin/extension/wizard/@class`). Diese wird von der Basisklasse `BasicNewResourceWizard` abgeleitet und implementiert die nötigen Operationen der Navigations-Elemente (`Finish`, `Cancel`, etc.). Das eigentliche Layout des Wizards wird hingegen von einer `WizardPage` bestimmt.

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
  <extension
      id="CocoWizard"
      name="Coco/R Wizard"
      point="org.eclipse.ui.newWizards">
    <category
        id="at.ssw.coco.ide.category"
        name="Coco/R"/>
      <wizard
          category="at.ssw.coco.ide.category"
          class="at.ssw.coco.ide.wizard.CocoWizard"
          icon="icons/ssw.gif"
          id="at.ssw.coco.ide.atgwizard"
          name="Coco/R Parser &amp; Scanner"/>
  </extension>
  <extension
      id="CocoActionSet"
      name="Coco Action Set"
      point="org.eclipse.ui.actionSets">
    <actionSet
        id="at.ssw.coco.ide.actions.actionSet"
        label="The Actions Set"
        visible="true">
      <action
          class="at.ssw.coco.ide.actions.OpenATGWizardAction"
          icon="icons/coco.gif"
          id="at.ssw.coco.ide.action.OpenATGWizardAction"
          label="New Coco/R Parser &amp; Scanner"
          toolbarPath="org.eclipse.jdt.ui.JavaElementCreationActionSet"
          tooltip="New Coco/R Parser &amp; Scanner"/>
      </actionSet>
   </extension>

   <!-- more extensions here -->
</plugin>
```

Listing 16. Ausschnitt aus dem Manifest von `at.ssw.coco.ide`

Als Alternative für den Aufruf dieses Wizards wurde für das Coco/R Plugin ein Button der Toolbar hinzugefügt, welcher als *ActionSet* in der Manifest Datei (Listing 16) eingetragen wurde. Mit diesem Eintrag wird bereits erreicht, dass das Eclipse Framework den Button mit den entsprechenden Attributen (Icon, Label, Tool-Tip, etc.) erzeugt und anzeigt. Die eigentliche Funktion steckt wiederum in der angegebenen Klasse, `at.ssw.coco.ide.actions.OpenATGWizardAction` (XPath: `/plugin/extension/actionSet/action/@class`)

5.4 Coco/R Integration

Die Integration des Compiler Generators Coco/R brachte einige Schwierigkeiten mit sich, da dessen Design und Konzept eine Integration in ein anderes Software-Projekt nicht vorsah. Insofern wurden z.B. bei schwerwiegenden Fehlern in der Speicherverwaltung, Verarbeitung der Parameter o.ä. der komplette Generator mit Angabe einer Fehlermeldung und entsprechendem Exit-Code beendet.
Ein weiteres Problem auf dem Weg, die Eclipse SDK als IDE für Coco/R zu verwenden, war die Tatsache, dass Fehlermeldungen bzw. Warnungen direkt auf den *Standard-Output* geschrieben wurden.
Der mit der Coco/R Version von Juni 2006 generierte Compiler (Parser + Scanner) konnte also nicht ohne zusätzlicher Modifikationen verwendet werden.
Da seitens der Projekt-Leitung des Coco/R für das kommende Release keine Änderungen diesbezüglich geplant waren, musste ein entsprechendes „Work-Around“ für diese Probleme geschaffen werden. Grund hierfür ist die Möglichkeit, Coco/R unabhängig von diesem Plugin zu warten und zu erneuern, unbedingt zu erhalten. Es galt also folgende Schwierigkeiten zu bewältigen:

- *Fehlermeldungen.* Umleiten des Standard-Outputs und Parsen der Fehlermeldungen.

- *System.exit().* Abfangen der Programmabbrüche.

Der erste der beiden Punkte stellte sich als der wesentlich einfachere Teil heraus, und konnte ziemlich „geradlinig“ gelöst werden. (Siehe Listing 17)

```
public static ArrayList<CocoError> execute(...) {
  ByteArrayOutputStream out = new ByteArrayOutputStream();
  PrintStream printer = new PrintStream(out),
  errBackup = System.err, outBackup = System.out;

  System.setErr(printer);
  System.setOut(printer);

  // execute Coco/R

  System.setErr(errBackup);
  System.setOut(outBackup);

  return analyseOutput(out)
}
```

Listing 17. Umleiten des Standard-Outputs

Das Parsen der so erhaltenen Fehlermeldungen wurde wiederum mit einem Coco/R generierten Parser/Scanner Paar gelöst. Das Problem der *System.exit()* Aufrufe könnte hierbei durch Ändern der Frame-Dateien umgangen werden. Dies ist allerdings bei der offiziellen Jar-Version des Generators nicht möglich, da diese, aus oben genannten Gründen, unangetastet bleiben sollte. Darum musste zu einer etwas unschönen Maßnahme gegriffen werden, welche einen eigens hierfür angepassten *Security Manager* erforderlich machte.
Auf diesem Weg konnten Programm-Abbrüche, welche das Beenden der gesamten Umgebung bedeuteten, hervorragend abgefangen werden. Das Problem hierbei bestand allerdings darin, dass die Standard-Einstellung eines `SecurityManagers` es verbietet, erneut einen anderen zu setzen bzw. den aktuellen zu entfernen. Diese Einschränkung kann auf saubere Art und Weise lediglich durch Angabe einer externen, der Java VM als Parameter übergebenen „Policy"-Datei aufgehoben werden. Da dies im Fall eines Eclipse SDK Plugins unter keinen Umständen möglich ist, musste ein `SecurityManager` wie in Listing 18 abgeändert werden.

```
public class CocoSecurityManager extends SecurityManager {
  private boolean fCheckExitVM;

  public CocoSecurityManager() {
    fCheckExitVM = false;
  }

  public void checkPermission(Permission perm, Object context) {}

  public static void throwAccessControlException(String msg)
  throws AccessControlException
  {
    throw new AccessControlException(msg);
  }

  public void checkExitVM(boolean b) {
    fCheckExitVM = b;
  }

  public void checkPermission(Permission perm) {
    if (fCheckExitVM && "exitVM".equals(perm.getName())) {
      throwAccessControlException("System.exit(...) called!");
    }
  }
}
```

Listing 18. Der eigens für diesen Spezialfall angepasste `SecurityManager`, mit der Aufgabe, die Programm-Abbrüche abzufangen.

Glücklicherweise reagierte die Projekt-Leitung des Coco/R doch noch auf die Anfragen, diese Punkte abzuändern, schränkte die exzessive Nutzung der Programmabbrüche ein und verwendete stattdessen eine `FatalErrorException`. Diese wird im offiziellen Produkt erst in der `main` Methode abgefangen und ermöglicht so eine problemlose Integration in andere Projekte. Weiters werden nun alle Fehler- und Warnmeldungen über eine eigene Fehler Klasse transportiert.
Der aktuelle Stand der Integration des Coco/R verwendet also die erwähnte `Errors` Klasse als Basis und überschreibt die darin enthaltenen Methoden zur

Ausgabe der Fehler und Warnungen, um diese stattdessen in einer Liste abzulegen und an das Plugin weiterzureichen.

6 Ausblick

Naturgemäß gelangt Software selten an einen Punkt an dem es nichts mehr daran zu arbeiten gibt, seien es Verbesserungen oder Erweiterungen der momentanen Funktionalität. Ebenso ist dies in Zusammenhang mit dem Coco/R Plugin der Fall und somit lassen sich folgende Punkte, welche in Zukunft realisiert werden bzw. werden könnten, anführen. Zum Teil stellen diese Punkte konkrete Pläne für die nähere Zukunft dar, beschreiben aber auch wage idealisierte Ideen, welche womöglich sogar durch Dritte umgesetzt werden könnten.

– *Syntax-Highlighting.* Das Syntax-Higlighting ist speziell in Bezug auf den Bereich der semantischen Aktionen deutlich ausbaufähig. Hier wäre es vor allem wünschenswert, die Scanner und Methoden der JDT zu verwenden, um den selben Umfang an Unterstützung auch in diesen eingeschränkten Bereichen zu erfahren.

– *Mappen der Fehler der JDT.* Wesentlich zum Komfort der Entwicklung mit Coco/R würde das „Mappen“ der Fehler, welche im erzeugten Parser und Scanner angezeigt werden, auf die dazugehörige ATG Datei und dessen semantischen Aktionen, beitragen.

– *Sprachenerweiterung.* Da Coco/R mittlerweile eine große Palette an unterschiedlichen Sprachen unterstützt, also für die ein Compiler generiert werden kann, wäre es vor allem für neuere und verbreitete Sprachen sinnvoll, das Plugin dahingehend zu erweitern. Hierbei würde sich neben C#, wofür es zum gegebenen Zeitpunkt scheinbar an aktiven Projekten einer IDE auf Basis des Eclipse Frameworks mangelt, C bzw. C++ anbieten. Mit den *C/C++ Development Tools* (CDT) der Eclipse Foundation würde auch eine entsprechende entwickelte Entwicklungsumgebung als Basis bereitstehen.

– *I18N – L10N.* Internationalisierung und Lokalisierung, also die Anpassung an andere Sprachen und Kulturen, ohne in weiterer Folge den Quellcode zu verändern.

– *Annotations.* Die aus den Editoren des JDT bekannten Hinweise können natürlich auch im Fall der Intergration des Coco/R weiter ausgebaut werden.

– *Frame Dateien.* Support für das Arbeiten mit den Frame Dateien, ähnlich dem der attributierten Grammatik. Wobei dies vermutlich, wegen der Ähnlichkeit zu herkömmlichen Java Quellcode, auf eine Erweiterung der Scanner des JDT hinausläuft.

– *Refactoring Tools.* Im Speziellen fällt u.a. das projekt-weite Ändern von Namen einzelner Methoden, Dateien oder Feldern in diese Kategorie.

– *Indentation support.* Um den Schreibfluss zu erhalten bzw. zu erhöhen ist es nützlich die strukturellen Eigenschaften (Einrücken, Klammer, etc.) automatisch vorzunehmen.

– *Properties.* Hierbei bieten sich einige Möglichkeiten an, um die Integration weiter zu erhöhen. Vorallem die Angabe eines Namens für die erzeugten Parser und Scanner wäre hier vorstellbar.

7 Zusammenfassung

Das Eclipse SDK ist ein Framework, welches es ermöglicht unterschiedlichste Applikationen und Software-Projekte auf dessen Basis zu realisieren bzw. zu integrieren. Das zentrale Konzept von *Extensions* und *Extension-Points* zieht sich durch das gesamte Framework und ermöglicht so einen entsprechend hohen Grad an Modularität und loser Bindung der einzelnen Komponenten.

Die Art und Weise der Plugin-Entwicklung für die Eclipse SDK hat sich jedoch vor allem mit der Implementierung des OSGi Frameworks wesentlich geändert. Mit dem dritten Major Release der Eclipse SDK wurde dieser Prozess begonnen und immer weiter umgesetzt, sodass die aktuelle Version nun das Aushängeschild der OSGi für Desktop Anwendungen darstellt.

Am Institut für System Software der Johannes Kepler Universität gab es bereits ein Projekt, um den Compiler Generator Coco/R in Form eines Plugins in das Eclipse Framework und die *Java Development Tools* zu integrieren. Die oben erwähnten Änderungen, sowie die fortschreitende Entwicklung des Coco/R machten es jedoch erforderlich, dieses aufbauend auf den neuen Konzepten und Möglichkeiten neu entstehen zu lassen.

Für den Erfolg dieser Arbeit waren grundlegende Änderungen in der Struktur des Compiler Generators Coco/R notwendig, welche nach längerer Diskussion der Dringlichkeit und Ausloten unterschiedlicher Lösungsansätze, in Zusammenarbeit mit dessen Projektleitung durchgeführt wurden.

Vorallem für die grafische Benutzeroberfläche wurden die erwähnten *Extension-Points* genutzt und erweitert, um die aus den „Eclipse Tools“ bekannten Stil- und Bedienelemente auch in diesem Plugin zur Verfügung zu stellen. Dies umfasst unter anderem folgende Komponenten: *Syntax-Highlighting, Outline View, Keyword Completion, Hyperlink Navigation, Folding*

Literatur

[Arthorne, 2004] Arthorne, J. (2004). Project builders and natures. In *Eclipse Corner Articles*. Eclipse.org.

[Bolour, 2003] Bolour, A. (2003). Notes on the eclipse plug-in architecture. In *Eclipse Corner Articles*. Eclipse.org.

[Clayberg and Rubel, 2004] Clayberg, E. and Rubel, D. (2004). *Eclipse. Building Commercial-Quality Plug-Ins*. Addison-Wesley Professional.

[Daum, 2006] Daum, B. (2006). *Java-Entwicklung mit Eclipse 3.1*. dpunkt Verlag.

[Dobler and Pirklbauer, 1990] Dobler, H. and Pirklbauer, K. (1990). Coco-2 – a new compiler compiler. *ACM SIGPLAN Notices*, 25(5):82–90.

[Eclipse Foundation, 2003] Eclipse Foundation, I. (2003). Eclipse platform technical overview. URL: http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

[Eclipse Foundation, 2004] Eclipse Foundation, I. (2004). Eclipse public license -v 1.0. URL: http://opensource.org/licenses/eclipse-1.0.php.

[Eclipse Foundation, 2006a] Eclipse Foundation, I. (2006a). About us. URL: http://www.eclipse.org/org.

[Eclipse Foundation, 2006b] Eclipse Foundation, I. (2006b). Eclipse platform technical overview. URL: http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf.

[Eclipse Foundation, 2006c] Eclipse Foundation, I. (2006c). Help - eclipse sdk. URL: http://help.eclipse.org/help32/topic/org.eclipse.platform.doc.isv/reference/misc/plugin_dtd.html.

[Freier, 2005] Freier, M. (2005). Erweiterung eines kompilier-rahmenwerks zur generierung von konzeptorientierten inhaltverwaltungsanwendungen. Master's thesis, Technische Universität Hamburg-Harburg.

[Friese, 2004] Friese, P. (2004). Incremental project builder in eclipse. *iX – Magazin für professionelle Informationstechnik*, 8:121 – 123.

[Gamma and Beck, 2004] Gamma, E. and Beck, K. (2004). *Contributing to Eclipse: Principles, Patterns, and Plug-Ins*. Addison-Wesley Professional.

[IBM, 2002] IBM (2002). Common public license version 1.0. URL: http://opensource.org/licenses/cpl1.0.php.

[Mössenböck, 1990a] Mössenböck, H. (1990a). Coco/r - a generator for fast compiler front-ends. Technical report, ETH Zürich.

[Mössenböck, 1990b] Mössenböck, H. (1990b). A generator for production quality compilers. *3rd intl. workshop on compiler compilers (CC'90), Lecture Notes in Computer Science*, 477:42–55.

[Mössenböck, 2005] Mössenböck, H. (2005). The compiler generator coco/r – user manual. URL: http://ssw.jku.at/Research/Projects/Coco/Doc/UserManual.pdf.

[Mössenböck et al., 2006] Mössenböck, H., Löberbauer, M., and Wöß, A. (2006). The compiler generator coco/r. URL: http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/.

[Perscheid, 2005] Perscheid, M. (2005). Eclipse plug-ins. URL: wendtstud1.hpi.uni-potsdam.de/sysmod-seminar/SS2005/elaborations/06-Eclipse-Plugins.pdf.

[Rechenberg and Mössenböck, 1985] Rechenberg, P. and Mössenböck, H. (1985). *Ein Compiler-Generator für Mikrocomputer*. Hanser-Verlag.

[Wirth, 1977] Wirth, N. (1977). What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):822 – 823.

[Wressnegger, 2006] Wressnegger, C. (2006). Coco/r development tools. URL: http://padre.madebywR.org.

A Benutzerdokumentation

A.1 Systembeschreibung

Zweck. Mit Hilfe des Coco/R Plugins für Eclipse wurde ein unterstützendes Werkzeug zur Entwicklung von Compilern mit Hilfe des Compiler Generators Coco/R geschaffen. Hierzu wurde die IDE für die attributierte Grammatik, welche den Compiler beschreibt (ATG Dateien), in das vorhandene *Java Development Toolkit* der Eclipse Foundation integriert.

Somit ist es möglich, mit minimalem Aufwand Coco/R nahtlos in ein vorhandenes Java Projekt einzugliedern. Dies umfasst selbstverständlich auch die automatische Erzeugung des beschriebenen Compilers und dessen Platzierung in der Projekt-Hierarchie bzw. Package Struktur. Weiters wurde besonderen Wert darauf gelegt, das Arbeiten an den ATG Dateien entsprechend zu erleichtern und so effizient wie möglich zu unterstützen.

Voraussetzungen. Das Plugin wurde unter den Gesichtspunkten des neu implementierten Konzeptes des OSGi Frameworks entwickelt und darauf abgestimmt. Insofern ist es also auf die aktuelle Version des **Eclipse SDK 3.2** ausgelegt und kann eine Rückwärtskompatiblität nicht garantieren. Das Eclipse SDK in der aktuellsten Version ist unter `http://www.eclipse.org/downloads/` erhältlich, das Coco/R Plugin ist wiederum von dessen Projektseite zu beziehen: `http://padre.madebywR.org/`.

Unterstützung zur Installation des Plugins finden sie im Kapitel A.2, sollten sie Hilfe beim Einrichten des Eclipse SDK benötigen, finden sie diese unter `http://www.eclipse.org/documentation/`

Wie auch die komplette Eclipse SDK und die Java Development Tools (JDT), ist auch das Coco/R Plugin gänzlich plattform-unabhängig, kann also auf allen gängigen Betriebssystemen verwendet werden.

Bedienung. Bei der Bedienung wurde darauf geachtet die von Eclipse geläufigen und bewehrten Komponenten wieder zu verwenden. Dies betrifft sowohl die Steuer- und Feedback-Elemente, als auch die verwendeten grafischen Stilmittel.

Somit finden sich die von der Java IDE und den Java Development Tools bekannten Grundbausteine, wie zB ein Editor mit *Syntax-Highlighting, Outline View, Keyword Completion, Hyperlink Navigation, Folding*, etc. auch in dem Plugin wieder.
Auch die Erweiterung eines Java Projekts zu einem Projekt, welches Coco/R und somit das Coco/R Plugin verwendet, geschieht anhand eines einfach zu bedienenden Wizards.

Datenfluss. Als Eingabe erwartet das Plugin, wie auch bei der herkömmlichen Verwendung des Coco/R, eine attributierte Grammatik (die ATG Datei). Um die im Normalfall benötigten Programm-Parameter muss sich allerdings nicht mehr gekümmert werden, da diese vom Plugin gesetzt werden.

Da der Compiler Generator jedesmal beim Abspeichern einer Änderung, automatisch aufgerufen wird, erstellt dieser auch bei jeder Modifikation der ATG den dazugehörigen korrekt in das Projekt eingegliederten Scanner und Parser. Dies wurde mit der Abbildung 11 versucht zu verdeutlichen.
Eine Ausnahme stellen hier natürlich Fehler in der Compiler Beschreibung (ATG) dar. In diesem Fall werden diese entsprechend aufbereitet und, wenn vorhanden, mit Zeilennummer und Spalte angezeigt.

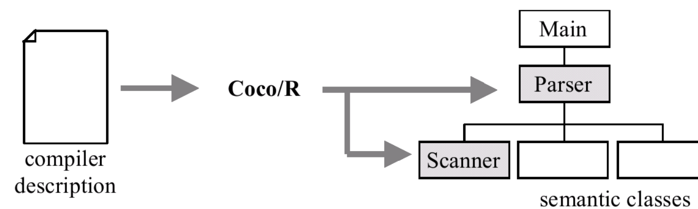


Abbildung 11. Ein- und Ausgabe von Coco/R

Einschränkungen. Dem Plugin ist es leider noch nicht möglich, die Sprachenvielfalt von Coco/R widerzuspiegeln und ist somit auf Java beschränkt. Weiters sind die von der Java IDE gewohnten Features wie Keyword Completion, Folding, Live-Error Detection, etc. nicht auf den Bereich der semantischen Aktionen übertragbar.

Ein weiteres Defizit, welches allerdings bereits von Coco/R herrührt, ist die Tatsache, dass es zum gegebenen Zeitpunkt nicht möglich ist, die Namen der Klassen der erzeugten Scanner und Parser zu veränderen. Die Möglichkeit hierzu könnte aber bereits mit dem nächsten Release von Coco/R gegeben sein.

A.2 Installationsanleitung

Wie auch sämtliche von der Eclipse Foundation veröffentlichten und zur Verfügung gestellten Plugins, ist es ebenfalls möglich, das Coco/R Plugin online zu beziehen und mittels des im Eclipse SDK integrierten *Update Manager*s zu installieren. Der Update Manager wird über das Hauptmenü `Help > Software Updates > Find and Install...` aufgerufen und präsentiert sich wie in Abbildung 12 mit der Frage nach der Art des Updates.

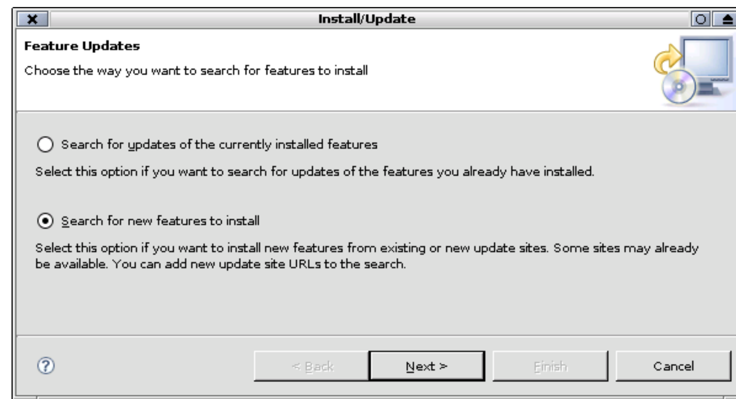


Abbildung 12. Der Updatemanager, mit der Frage nach der Art des zu installierenden Features. (Installation eines neuen Features bzw. Update eines Features)

Hier ist „`Search for new features to install`“ auszuwählen, womit man mit dem Betätigen des `Next`-Buttons zur Auswahl der Update Seite gelangt.
Um nun das Coco/R Plugin installieren zu können, muss zunächst die dazugehörige Update Site hinzugefügt werden. Den geringsten Aufwand erzielt man unter Verwendung einer Online-Ressource, welche man durch `New Remote Site...` hinzufügt. (siehe Abbildung 13)

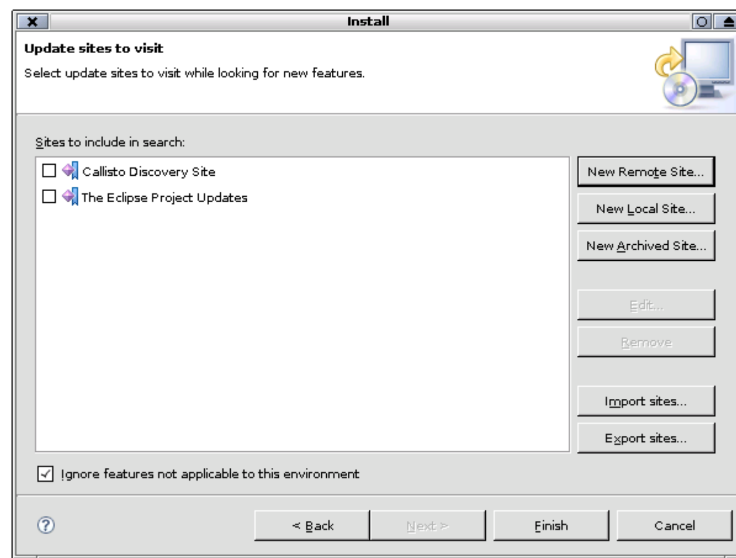


Abbildung 13. Der Updatemanager für die Auswahl der Quelle des zu installierenden Features.

Darauhin erscheint ein Dialog, welcher die Eingabe der *Update Site* ermöglicht. Dieser ist wie in Abbildung 14 gezeigt, mit dem Namen der Update Site und der entsprechenden URL („Uniform Resource Locator“) zu füllen.
> `http://padre.madebywR.org/update/`

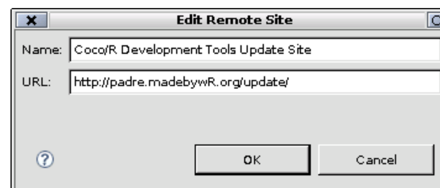


Abbildung 14. Eingabe-Maske für die *Remote Update Site*.

Sollte ein direkter Online-Bezug der Quellen nicht möglich sein, sei es durch Restriktionen der Firewall, dem Proxy des Online-Anbieters (der Firma), o.ä., so gibt es auch eine Offline-Lösung das Coco/R Plugin zu installieren. Nähere Details hierzu sind am Ende der Anleitung für die Online Installation (siehe Kapitel A.2.Offline) zu finden.

Mit dem `OK`-Button kann die Update Site nun der Liste hinzugefügt werden. Um den Vorgang zu starten, muss der neu erstellten Eintrag, analog zur Abbildung 15, selektiert und anschließend mit `Finish` bestätigt werden.

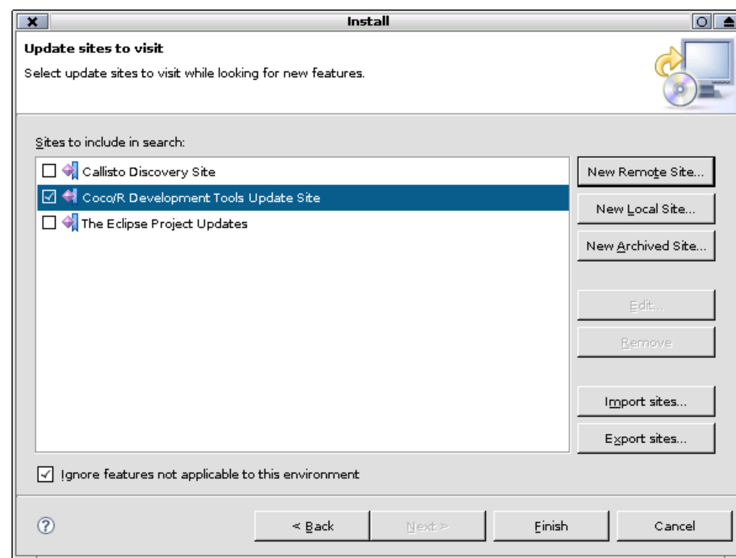


Abbildung 15. Der Updatemanager mit der ausgewählten **Coco/R Update Site**.

Nach kurzer Verarbeitungsphase (das Plugin wird heruntergeladen) erscheint der Inhalt, der auf der Update Site zur Verfügung steht. Nach entsprechender Auswahl des Coco/R Plugins (wie in Abbildung 16 zu sehen), kann mit `Next` fortgefahren werden.

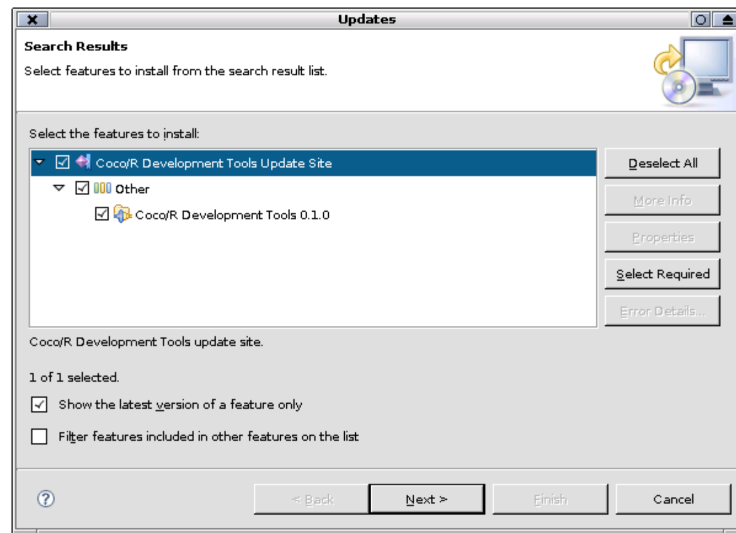


Abbildung 16. Selektion des Coco/R Plugins.

Bevor das Plugin nun endgültig installiert wird, müssen noch einige Einstellungen vorgenommen werden. Unter anderem müssen noch die Lizenz Bedingungen gelesen und bestätigt werden. Sofern man mit den so anführten Vereinbarungen einverstanden ist, kann man dies, im Fall dieses Dialogs mit „`I accept the terms in the licence agreement`“ und der Betätigung des `Next` Buttons, tun. (Siehe Abbildung 17)

Will man den vorgegebenen Installations-Pfad des Coco/R Plugins nicht ändern, wogegen aus Sicht des Autors nichts spricht, kann der folgende Dialog mit ruhigem Gewissen mit `Finish` abgeschloßen werden. Daraufhin wird der letzte Dialog in dieser Reihe aufgerufen, mit dessen Hilfe nun entschieden werden kann, ob das Plugin installiert (`Install` bzw. `Install All`) oder der Vorgang vielleicht doch noch abgebrochen werden soll. (`Cancel`)

Nach erfolgreich abgeschlossener Installation wird angeboten, das Eclipse SDK neu zu starten. Dies sollten angenommen werden, da ohne einem Neustart das

Coco/R Plugin, zumindest in der momentan aktuellen Version des SDKs, nicht zur Verfügung steht.

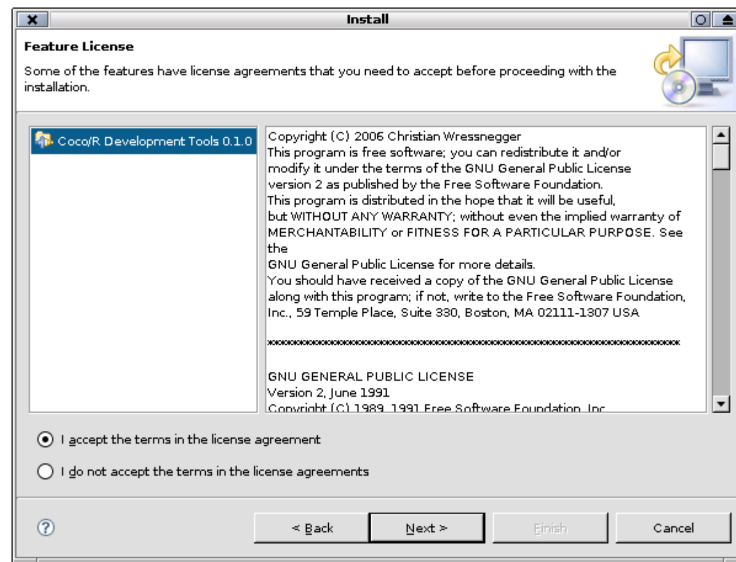


Abbildung 17. Lizenz-Bedingungen des Coco/R Plugins.

Nach dem Neustart sollte, sofern alles korrekt verlaufen ist, im *About Eclipse SDK* Dialog (zu erreichen über das Hauptmenü `Help > About Eclipse SDK`) das Logo des Coco/R Plugins neben den Repräsentationen der anderen Plugins zu sehen sein. Die Anzahl der Icons variiert natürlich je nach Anzahl der installierten Plugins, der Dialog wird aber ähnlich der Abbildung 18 aussehen.

Offline. Sollte es nicht möglich sein per *Eclipse SDK* auf die *Update Site* des Coco/R Plugins zu gelangen, so ist es notwendig das, ansonsten automatisch besorgte Quell-Packet manuell zu beschaffen.
Hierzu besucht man mit dem Web-Browser der Wahl die Update Site des Coco/R Plugins (`http://padre.madebywR.org/update/`), wo ein Archiv mit dem Namen `at.ssw.coco_x.y.z`, wobei `x.y.z` für die aktuelle Version des Plugins steht, zu finden ist.

Das weitere Vorgehen gestaltet sich analog zur Online-Variante (ab Abbildung 13), anstatt `New Remote Site` muss allerdings `New Archived Site` gewählt werden. Daraufhin erscheint ein Dialog zur Auswahl des Archivs, welches sie zuvor von der Update Site bezogen haben. Sobald die Selektion per `OK` bestätigt wurde,

Abbildung 18. *About Eclipse SDK* Dialog mit erfolgreich installiertem Coco/R Plugin.

erscheint die Eingabe-Maske, welche bereits aus Abbildung 14 bekannt ist. Von diesem Punkt an verläuft die Installation genau wie im ersten Abschnitt dieser Anleitung.

Der Vollständigkeit halber werden auf der Projekt Homepage außerdem die einzelnen Komponenten dieses Plugins angeboten, was eine weitere Alternative der Installation zulässt.
Da mit einem solchen Vorgehen allerdings der anschließende Komfort der Plugin-Verwaltung, welche die Eclipse SDK implementiert, und die Überprüfung der Plugins verloren geht, wird an dieser Stelle die erwähnte alternative Installation nicht mehr näher thematisiert.

Sollte dafür dennoch Interesse bestehen, kann das offiziellen Eclipse Wiki empfohlen werden: `http://wiki.eclipse.org/`, welches selbstverständlich auch eine Menge anderer nützlicher Informationen bietet.

A.3 Bedienungsanleitung

Getting Started. Die Einführung in die Bedienelemente (Wizards, Views, etc.) wird anhand drei typischer Beispiel-Szenarien, welche im Laufe der Arbeit mit dem Coco/R Plugin immer wieder auftreten, erfolgen:

- Erstellen eines **neuen** Java Projekts mit Coco/R Unterstützung.
- Importieren eines solchen Java Projekts.
- Navigation in Coco/R unterstützten Projekten.

In den folgenden Ausführungen werden die Schritte, welche das allgmeine Java Projekt betreffen bzw. sich mit dieser Vorgehensweise überschneiden, bewusst kurz gehalten, aber dennoch die wichtigsten Punkte angeführen.

Sollte es in diesen Abschnitten Probleme geben oder für den Fall das näher Informationen erwünscht sind, wird an dieser Stelle an einen der zahlreichen, entsprechenden Artikel und Anleitung zum Thema „Java Development Tools" (JDT) auf `http://www.eclipse.org` verwiesen.

Erstellen eines neuen Projekts. Um ein neues von Coco/R unterstütztes Projekt zu erstellen, muss zunächst ein neues Java Projekt erzeugt werden. Hierzu wählt man aus dem Hauptmenü: `File > New > Other...` aus oder betätigt den entsprechenden Button in der Toolbar. (siehe Abbildung 19 – 8. Button von links)

Abbildung 19. Toolbar der Eclipse SDK

Für den Fall, dass das Hauptmenü benutzt wurde, erscheint zunächst der *New Wizard* (Abbildung 20), der es Ihnen erlaubt einen Wizard für ein beliebiges Projekt zu wählen. In diesem Beispiel ist, wie erwähnt, ein Java Projekt nötig, welches in der Liste auszuwählen und per `Next`-Button zu bestätigen ist.

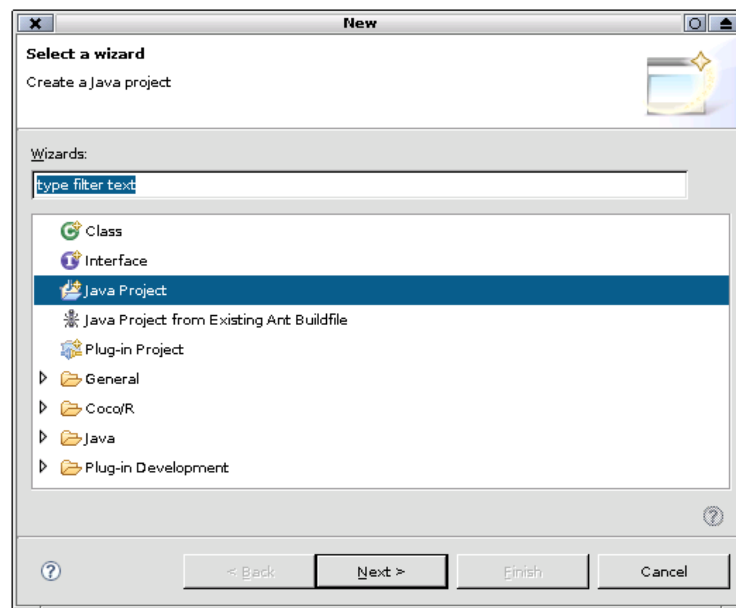


Abbildung 20. Der *‚New' Wizard*, zur Auswahl eines beliebigen Projekt-Typs.

Daraufhin erscheint der Wizard für das Erstellen eines Java Projekts, welches mittels des Buttons in der Toolbar ohne dem zuvor angeführten Zwischenschritt aufgerufen worden wäre.

Alles was für dieses einfache Beispiel nötig ist, ist in dem dafür vorgesehenen Eingabefeld, den Namen, im Fall der Abbildung 21 „Example#1“, einzutragen. Nun muss nur noch der `Finish`-Button betätigt werden, um das Java Projekt zu erstellen.

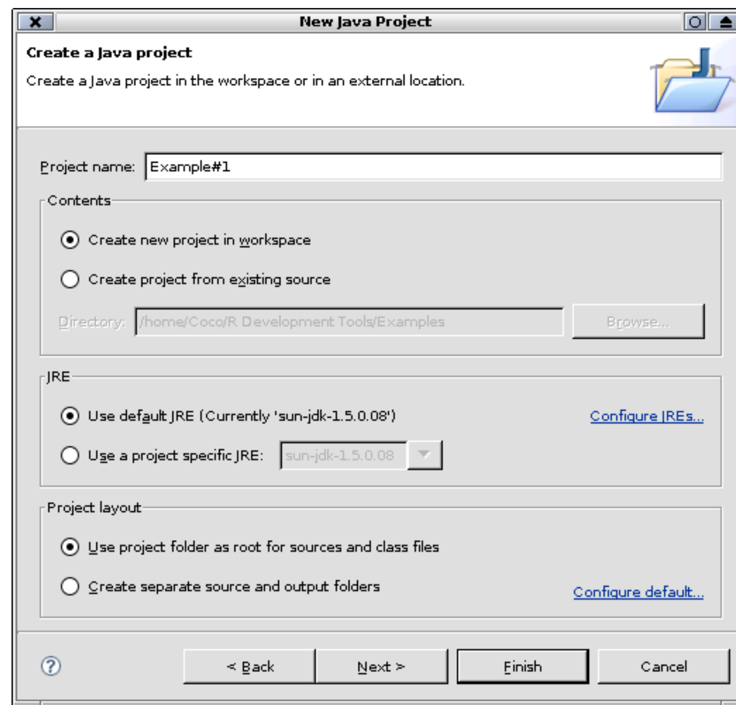


Abbildung 21. Der *New Java Project Wizard*.

Mit dem Schließen des Wizards erscheint bereits der erste Eintrag mit dem Namen „Example#1“ bzw. mit dem Namen, der zuvor für das Java Projekt vergeben wurde, im *Package Explorer*.

Um nun dieses Projekt dahingehend zu erweitern, Coco/R zu integrieren, muss erneut der *New Wizard* (Abbildung 20) über das Hauptmenü aufgerufen werden. (`File > New > Other...`) In der Liste befindet sich ein zusätzlicher Eintrag namens „Coco/R“ mit einem dazugehörigen Unterpunkt („Coco/R Parser & Scanner“).

Wählt man diesen aus, erscheint ein neuer Wizard, welcher die Aufgabe, Coco/R zu integrieren, übernehmen wird. (Abbildung 22)

Alternativ ist es natürlich auch wieder möglich, das gleiche Ergebnis mit der Eclipse SDK Toolbar zu erreichen. (siehe Abbildung 19 – 4. Button von links)

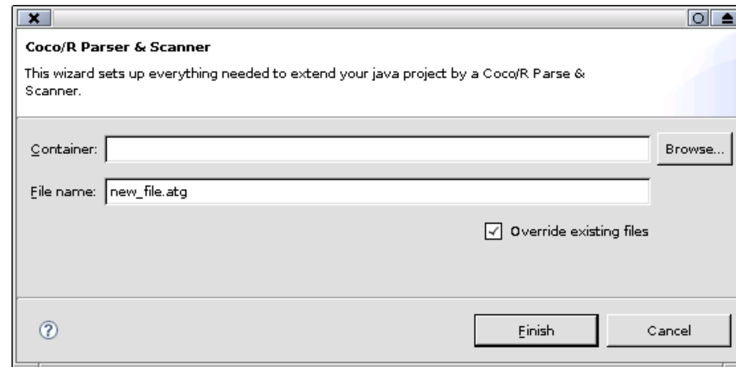


Abbildung 22. Der *Coco/R Extension Wizard*.

Ein Dateiname für die attributierte Grammatik ist bereits von vorne herein eingetragen, welcher natürlich geändert werden kann. Für unser einfaches Beispiel ist dies allerdings nebensächlich, viel wichtiger und unbedingt erforderlich ist es, ein Java Projekt auszuwählen, welches mit den zusätzlichen Funktionen ausgestattet werden soll. Betätigen sie hierzu den `Browse...` Button, woraufhin das in Abbildung 23 gezeigte Dialog Fenster erscheint.

In unserem Fall steht ohnehin nur ein einziger Eintrag zur Auswahl, eine einfache Bestätigung per `OK` Button reicht als aus.
Alles was jetzt noch fehlt, ist, den *Coco/R Extension Wizard* (Abbildung 22) mit `Finish` zu beenden.

Wenn man nun den *Package Explorer* betrachtet, wird man merken, dass sich dieser im Vergleich zu einem normalen Java Projekt bereits erheblich gefüllt hat. (vgl. Abbildung 24) Wie man sehen kann, wurden im *default Package* bereits der Scanner und Parser angelegt und außerhalb (im Dateisystem befinden sich diese natürlich auf der selben Ebene) befinden sich die beschreibenden Dateien. Die attributierte Grammatik (ATG) ist mittlerweile bekannt, die beiden anderen (`*.frame`) stellen die Grundgerüste für den Compiler Generator dar und werden jedem Coco/R Projekt separat beigelegt.
Es kann nun bereits versucht werden, erste Änderungen an der `new_file.atg` vorzunehmen und man wird sehen, dass sich bei jeder abgespeicherten Änderung die beiden Java Dateien entsprechend ändern. Sollte seitens Coco/R ein Fehler

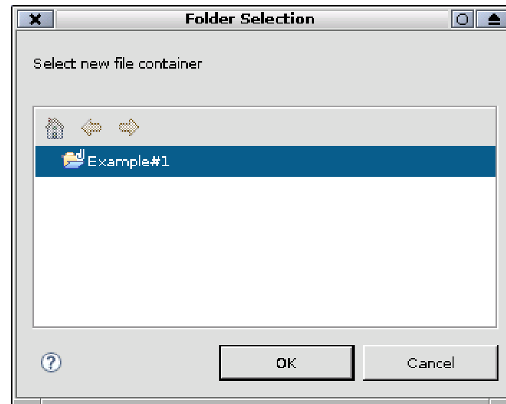


Abbildung 23. Dialog zur Auswahl des Ziel-Projektes bzw. -Packages der Coco/R Extension.

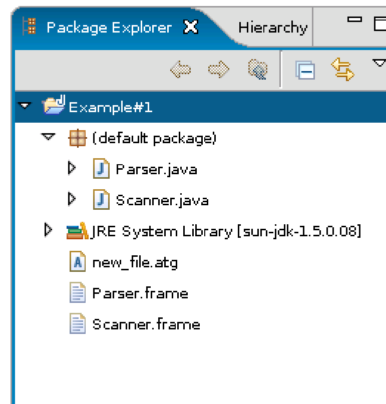


Abbildung 24. Der Package Explorer, welcher das „befüllte“ neue Java Projekt mit Coco/R Extension zeigt.

gemeldet werden, werden die alten Dateien gelöscht und die Fehlermeldungen in der *Problems View* angezeigt. Mehr dazu im Abschnitt A.3.Navigation.

Importieren eines bestehenden Projekts. In diesem Beispiel nehmen wir an, dass ein bereits vorhandenes Java Projekt mit Coco/R Extension vorhanden ist, allerdings nicht in dem aktuellen Workspace zur Verfügung steht. Hierbei gibt es wiederum zwei unterschiedliche Ansätze:

– Bei dem Projekt handelt es sich um ein komplettes von der Eclipse SDK erstelltes Coco/R unterstütztes Projekt.

– Lediglich die Dateien eines solchen Projekts (exkl. den Konfigurationsdateien der Eclipse SDK) sind im Dateisystem vorhanden.

Die erste Variante erfolgt wie bei jedem anderen Projekt auch, und wird aus diesem Grund hier nicht beschrieben. Da die Konfigurationsdateien der Eclipse SDK vorhanden sind, sind es auch die am Projekt vorgenommenen Modifikationen. Anders sieht es im zweiten Fall aus, darum wird dieser im Folgendem noch genauer beschrieben.

Sollten lediglich die Coco/R spezifischen und Quellcode Dateien vorhanden sein, muss zunächst ein leeres Java Projekt erstellt werden. Dies geschieht analog zu den Erläuterungen des ersten Anwendungsbeispiels („Erstellen eines neuen Projekts“). In den weiteren Erklärungen wird für den Namen dieses Projekts „Example#2“ verwendet werden.

Wie bei allen zu importierenden Projekten wird auch dieses über den Import Wizard (Abbildung 25) in den aktuellen Workspace eingepflegt. Zu diesem gelangt man unter anderem über das Hauptmenü: `File > Import...`

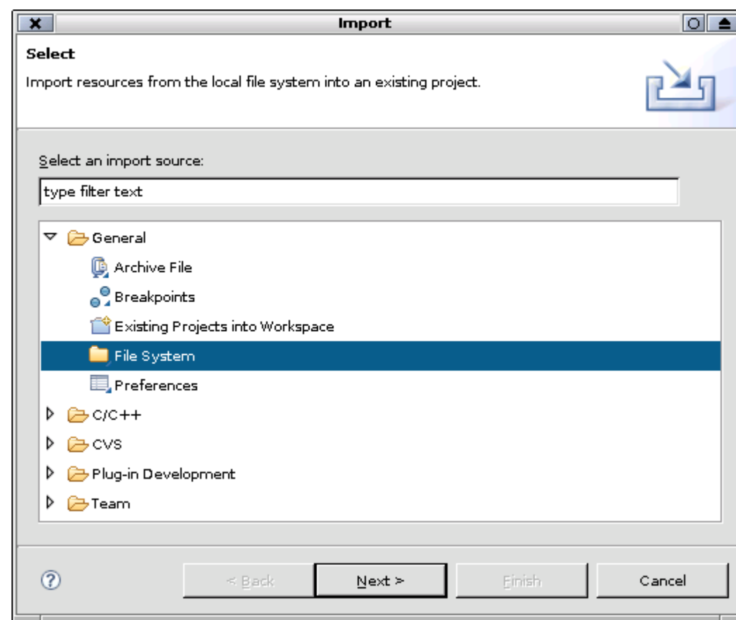


Abbildung 25. Der *Import Wizard* stellt unterschiedlichste Möglichkeiten zum Importieren von Projekten bereit.

In diesem Dialog wählt man nun den Unterpunkt `File System` in der Kategorie `General`. Dies führt einen zu, der in Abbildung 26 gezeigten, jedoch noch ohne den nötigen Angaben versehenen, Fortsetzung des Import Wizards.

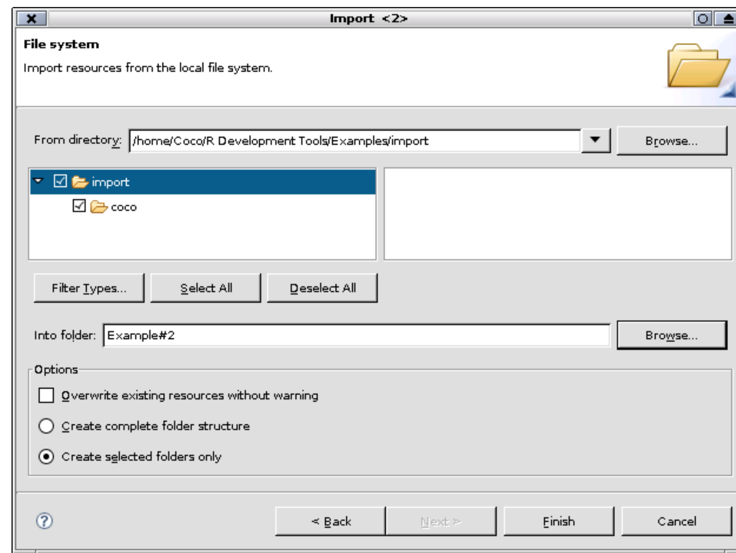


Abbildung 26. *Import Wizard*; um eine beliebigen Ordner-Struktur zu importieren.

Das aktuelle Verzeichnis des Projektes, als auch dessen Ziel werden jeweils durch die nebenstehenden `Browse`-Buttons angegeben. Der obere ruft einen „Öffnen“ Dialog auf, in dem nun die Ordner des zu importierenden Projekts ausgewählt werden. Unmittelbar nach erfolgter Auswahl sollte eine Liste, welche den Inhalt des ausgewählten Projekts repräsentiert, erscheinen.
Selektieren man nun den ersten Eintrag dieser Liste, wird der komplette Inhalt des Projekts als zu importieren markiert. (vgl. Abbildung 26)

Die Auswahl des Ziel-Ordners erfolgt über das Dialog Fenster, welches bereits aus Abbildung 23 bekannt ist. Allerdings stehen nun bereits zwei Projekte zur Auswahl. Hier ist also das eigens für dieses Beispiel erstellte Projekt, „Example#2“, zu wählen.

Um den Import abzuschließen, muss nur noch der `Finish`-Button gedrückt werden. Sollten irgendwelche Dateien im alten Projekt den gleichen Name tragen, wie jene die gerade importiert werden, so wird man gefragt, ob diese Überschrieben werden sollen. In diesem kleinen Beispiel sollte dies allerdings nicht der Fall sein.

Der letzte Schritt, um dem importierten Projekt dessen ursprüngliche Funktionalität (in Hinsicht auf den Compiler Generator Coco/R) zurückzugeben, erfolgt über dessen Kontextmenü. Selektiert man also das Projekt mit dem Name „Example#2“ im *Package Explorer* und wählt den Eintrag „*Extend to Coco/R Projekt*“ aus, wird dieser Prozess eingeleitet.

Zum Abschluss wird noch eine Bestätigung gezeigt und der ürspünglich Zustand ist wieder hergestellt.

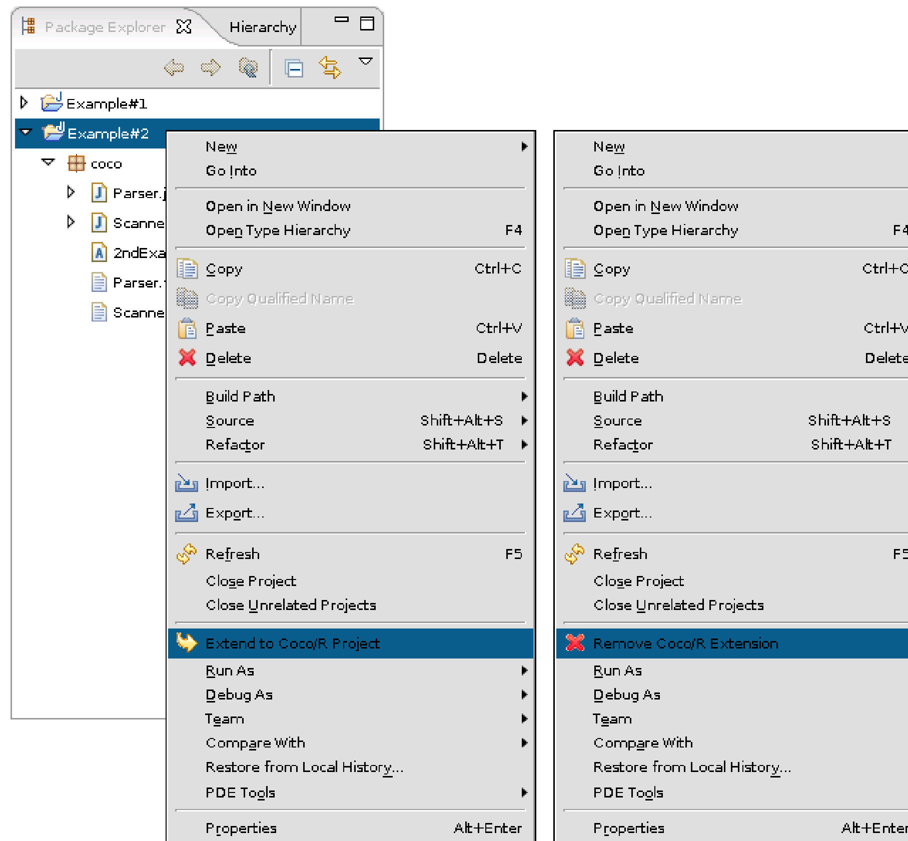


Abbildung 27. Kontext-Menü: „*Extend to Coco/R Project*“ und „*Remove Coco/R Extension*“

Analog dazu funktioniert auch das Entfernen der *Coco/R Extension*, wie es ebenfalls in Abbildung 27 angedeutet ist.

Navigation. In diesem kurzen Zusatz-Szenario werden die wichtigsten Steuerelemente, welche in dem Coco/R Plugin verarbeitet wurden, vorgestellt. Der markanteste Teil, weil auch relativ platzeinnehmend, ist die *Outline View*, welche eigentlich von allen in der Eclipse SDK angebotenen Editoren mit zur Verfügung gestellt wird. Abbildung 28 zeigt das strukturelle Gerüst der attributierten Grammatik des Templates, welches beim Erstellen eines neuen Coco/R Projekts in den „Source-Tree“ eingefügt wird.

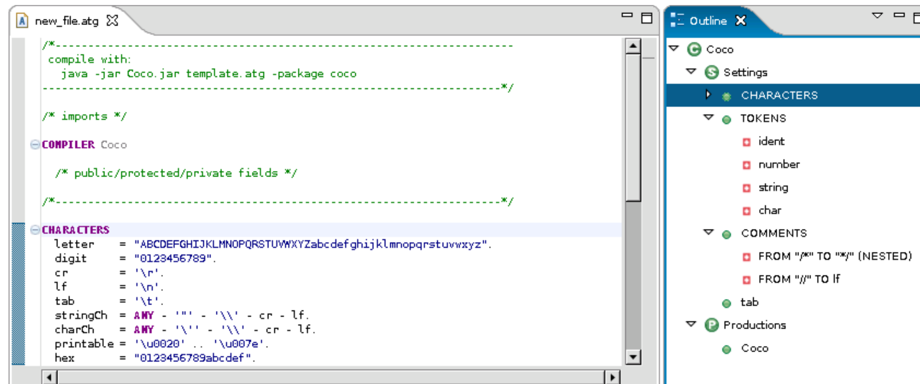


Abbildung 28. Der ATG-Editor inklusive der dazugehörigen Outline Struktur

Selektiert man nun ein Element aus der Outline View, so springt der Cursor des Editors zu der Stelle im Text wo sich dieses strukturelle Element befindet. Im Fall der Abbildung 28 ist dies der Abschnitt für die unterschiedlichen Zeichen-Definitionen („`CHARACTERS`“), welche gleichzeitig den Anfangspunkt der Scanner Einstellungen darstellt.

Sehr nützlich ist außerdem die im Editor integrierte „*Link Navigation*“, welche auch von dem Editor der Java Development Tools bekannt ist. Diese wird durch drücken der `CTRL`-Taste und gleichzeitige Mausbewegung über ein Signalwort, aktiviert. Das Signalwort erscheint daraufhin als Link, welcher auf die Stelle der Deklaration bzw. Definition dieses Ausdrucks verweist.

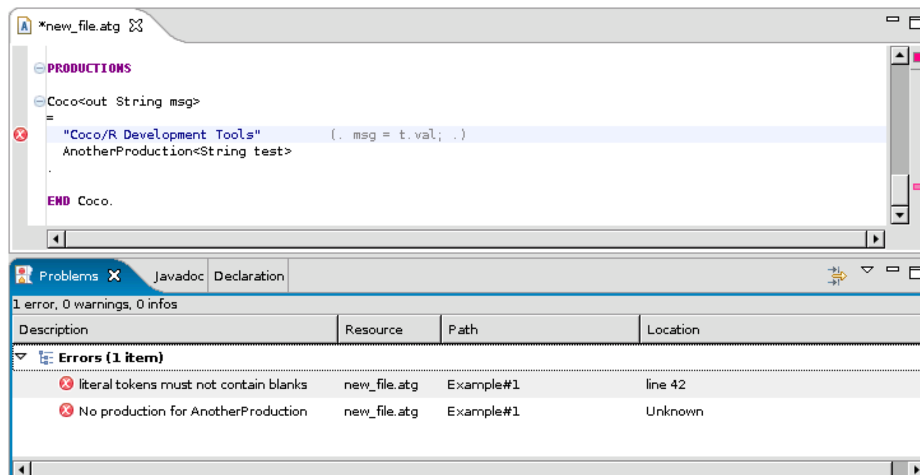


Abbildung 29. Die *Problems View* des Coco/R Plugins

Für die Navigation zu und das Anzeigen von Fehlern ist die *Problems View* zuständig. Abbildung 29 zeigt einen möglichen Zuständ dieser View.

Selektiert man nun in der *Problems View* einen Fehler, welcher mit entsprechendem Ressource-Verweis und Zeileangabe ausgestattet ist, springt der Cursor des Editors an die Stelle, an der der Fehler aufgetreten ist. Ist eine solche Zeilenangabe nicht vorhanden, wird lediglich die Datei als fehlerhaft markiert.

Zusätzlich zu den beiden Fehlerarten gibt es noch *Warnings*, welche allerdings zumeist ebenfalls ohne einer genauen Angabe der Position in der attributierten Grammatik auskommen.