An der Quelle

Dokument-Editoren als Eclipse-Plug-Ins

Leif Frenzel

Quellcode-Editoren sind das Herzstück jeder integrierten Entwicklungsumgebung (IDE). Für die tägliche Arbeit am Quelltext ist es unerlässlich, ein komfortables und zuverlässiges Werkzeug zur Verfügung zu haben – das nach Möglichkeit auch an individuelle Bedürfnisse anpassbar sein sollte. Eclipse ist nicht nur eine Java-IDE, sondern eine universelle Plattform für Programmierwerkzeuge, und kann daher um Unterstützung für jede denkbare Programmiersprache oder beliebige Formate für spezielle Konfigurationsdateien erweitert werden. Dieser Artikel erläutert die von Eclipse zur Verfügung gestellten Konzepte und Schnittstellen zur Quelltextbearbeitung, mittels deren ein Editor für beliebige Dokumenttypen als Plug-In implementiert werden kann.

Normal gibt's schon – die erste grundlegende Implementierung eines Texteditors

Um eine erste funktionsfähige Basis für einen Texteditor zu bekommen, ist nicht viel mehr nötig als eine leere Klasse anzulegen, die von TextEditor abgeleitet ist – der Basisimplementierung eines Texteditors im Eclipse-Framework. Dieser Editor enthält bereits eine Reihe von Standardfunktionalitäten zur Textbearbeitung, wie Kopier-, Ausschneide- und Einfüge-Operationen oder das Abspeichern bzw. Verwerfen von Änderungen über die gewohnten Menüs und Buttons der Workbench. (Diese Implementierung eines Texteditors wird in der Eclipse-Workbench außerdem als Standardeditor für sämtliche Dateitypen verwendet, denen kein spezieller Editor zugewiesen ist.)

Der Editor muss freilich, wie die meisten Werkzeuge eines Plug-Ins, in der Manifest-Datei plugin.xml deklariert werden (s. Listing 1, [Bsp]). Hier können u. a. auch Icons für die mit dem Editor assoziierten Ressourcen und Workbench-Windows angegeben werden. Außerdem muss festgelegt werden, für welche Ressourcen (i.d.R.: welche Dateitypen) der Editor in der Workbench verwendet werden soll. Dies geschieht durch Angabe einer komma-separierten Liste von Dateierweiterungen im Attribut extensions. Im Attribut class wird der eigentliche Editor spezifiziert: Die hier benannte Klasse muss IEditorPart implementieren.

Für Ihre Unterlagen – der DocumentProvider erzeugt das Modell für den Texteditor

Der Texteditor arbeitet nicht direkt auf der Ressource, sondern auf einem *Dokument*, welches das Interface IDocument implementieren muss. Wie ein solches Dokument beschaffen ist, hängt natürlich vom jeweils bearbeiteten Dateityp ab. Daher kann die Workbench dem Editor nur dann zu einem passenden Dokument verhelfen, wenn sie über eine Implementierung eines *DocumentProviders* (Interface IDocumentProvider) verfügt. Für einen gegebenen Ressourcentyp existiert innerhalb der Workbench jeweils nur eine Instanz des zuständigen DocumentProviders.

Wie jeder Editor in der Eclipse-Workbench bekommt auch der Texteditor die von ihm bearbeitete Ressource als IEditorInput zur Verfügung gestellt. Da diese Ressource in aller Regel eine Datei ist, muss der DocumentProvider aus dem Input ein entsprechendes Dokument-Objekt erstellen. Die Beispielimplementierung (MyDocumentProvider) delegiert auch diese Aufgabe an die von der Plattform

```
<extension
  point="org.eclipse.ui.editors">
  <editor name="Beispiel-Editor"
    icon="icons/myImage.gif"
    extensions="my"
    class="de.leiffrenzel.articles.jspektrum.editor.MyEditor"
    id="de.leiffrenzel.articles.jspektrum.editor.MyEditor">
  </editor>
</extension>
```

Listing 1: Deklaration eines Editors für einen Dateityp (Ausschnitt aus der plugin.xml)

```
<extension
  point="org.eclipse.ui.documentProviders">
  <provider extensions="my"
    class=
"de.leiffrenzel.articles.jspektrum.editor.MyDocumentProvi-
der"
    id=
"de.leiffrenzel.articles.jspektrum.editor.MyDocumentProvi-
der">
  </provider>
</extension>
```

Listing 2: Deklaration eines DocumentProviders für einen Dateityp (Ausschnitt aus der plugin.xml)

angebotene Standardimplementierung (FileDocumentProvider), von der sie abgeleitet ist.

Auch der DocumentProvider wird am besten in der plugin.xml deklariert (s. Listing 2), sodass sich die Workbench selbständig darum kümmern kann, eine Instanz zu verwalten und diese zum Erzeugen von Dokumenten zu verwenden, wenn der Benutzer einen Editor öffnet.

Gerecht aufgeteilt – das Partitionieren des Dokuments

Viele der Editorfunktionen arbeiten nicht unterschiedslos auf dem gesamten Dokument, sondern nur auf Quelltexten eines bestimmten Typs. Beispielsweise wird ein Java-Editor unterschiedliche Strategien bei der automatischen Code-Vervollständigung verfolgen, je nachdem, um was für einen Bereich des Dokuments es sich handelt: Befindet sich der Textcursor über einem Java-Codeabschnitt, so stehen andere Schlüsselworte zur Verfügung als etwa über einem JavaDoc-Kommentar – auf einem anderen Kommentar gibt es überhaupt kein Code Assist usf.

Eine grundlegende Eigenschaft eines IDocuments ist daher, dass es *partitioniert* ist, also in typisierte, disjunkte Abschnitte unterteilt ist. Das Dokument verwaltet diese Partitionierung und aktualisiert sie, wenn Änderungen vorgenommen werden.

Die Unterteilung des Dokuments übernimmt ein *Partitionierer* (der vom Typ IDocumentPartitioner sein muss). Er wird vom Dokument verwendet, um beim Erzeugen oder bei Änderungen die Partitionierung zu bestimmen. In der Beispielimplementierung wird der von der Plattform bereitgestellte DefaultPartitioner verwendet (s. Listing 3), der mit einem regelbasierten Scanner zusammenarbeitet.

In diesem Fall ist der Scanner (MyPartitionScanner) sehr einfach: Er kennt nur eine Regel für einzeilige Kommentare. Für realistische Scanner-Implementierungen bietet das Eclipse SDK eine Reihe weiterer Hilfsmittel im Paket org.eclipse.jface.text.rules. Beispiele für regelbasiertes Scannen finden sich in [JEd] und [XEd].

Wir gehen nun daran, für die beiden Partitionstypen jeweils das Verhalten des Editors zu programmieren, und begeben uns damit von der Seite des Modells auf die Seite des Viewers.

Die Vielfalt der Ansichten – Konfigurationen für den SourceViewer

Der Editor enthält als Texteditor einen SourceViewer, der für das Darstellen des Textes verantwortlich ist, und ein Objekt vom Typ SourceViewerConfiguration definiert die dazu verwendete Funktionalität. Bei der Initialisierung des Editors wird dieses Objekt erzeugt und dem Editor übergeben (s. Listing 4).

Funktionalitäten werden hier bei Bedarf nach dem „Strategy"-Muster [GoF96] hinzugefügt. Die Methoden der SourceViewerConfiguration werden vom SourceViewer benutzt, um Dokument-spezifisches Verhalten aufrufen zu können*. Dazu muss die Klasse SourceViewerConfiguration abgeleitet und um die jeweils gewünschten Strategien ergänzt werden. Im Folgenden wird dies anhand zweier häufig benötigter Beispiele beschrieben: *Syntax Coloring* und *Code Assist*.

Farbe bekennen – Syntax Coloring implementieren

Eine der grundlegenden Fähigkeiten eines Editors für einen bestimmten Dokumenttyp ist das farbliche Hervorheben bestimmter Textelemente: Schlüsselworte oder Kommentare in Quelltext, Attributnamen oder Zeichenketten in Konfigurationsdateien usf. Dies geschieht jedoch nicht überall im Dokument in gleicher Weise. Beispielsweise ist es in Kommentaren üblicherweise nicht erwünscht, dass Schlüsselworte oder Zeichenketten eine Sonderbehandlung erfahren. Syntax Coloring setzt also voraus, dass das Dokument, wie oben beschrieben, in typisierte Partitionen aufgeteilt wurde.

```
protected IDocument createDocument( final Object elem )
throws CoreException {
  // Wir lassen die Oberklasse ein Model (IDocument)
  // für den Editor aus der Datei erzeugen.
  IDocument document = super.createDocument( elem );
  // Ein Partitionierer muss mit dem Model verknüpft
  // werden. Hier wird der von der Plattform angebotene
  // Default-Partitionierer verwendet, der mit einem
  // regelbasierten Scanner zusammenarbeitet.
  if ( document != null ) {
    String[] tokenTypes = new String[] {
      MyPartitionScanner.COMMENT,
      IDocument.DEFAULT_CONTENT_TYPE };
    MyPartitionScanner scanner =
      new MyPartitionScanner();
    IDocumentPartitioner partitioner =
      new DefaultPartitioner( scanner, tokenTypes );
    partitioner.connect( document );
    document.setDocumentPartitioner( partitioner );
  }
  return document;
}
```

Listing 3: Erzeugen des Dokuments und des Partitionierers (MyDocumentProvider)

```
protected void initializeEditor() {
  super.initializeEditor();
  // hier wird dem SourceViewer des Editors mit Hilfe eines
  // Konfigurationsobjekts die spezielle Funktionalität
  // 'eingepflanzt'
  setSourceViewerConfiguration(
    new MySourceViewerConfiguration() );
}
```

Listing 4: Setzen der SourceViewerConfiguration im Editor

* Die Eclipse-Dokumentation spricht in dieser Hinsicht gelegentlich von „source viewer *plugins*" statt von Strategien; gemeint ist hier jedoch lediglich die Implementierung einer speziellen Editor-Funktionalität, oft nur in einer einzelnen Klasse – es geht nicht um eigenständige Plug-Ins für die Workbench.

Die Konfiguration muss deshalb angeben, für welche Partitionstypen sie Funktionalität bereitstellt. Dies geschieht durch Überschreiben der Methode getConfiguredContentTypes(ISourceViewer), die standardmäßig nur IDocument.DEFAULT_CONTENT_TYPE zurückgibt. Da im Beispiel außerdem Kommentare farblich hervorgehoben werden sollen, ergänzen wir die zurückgegebene Liste hier um den von uns definierten Partitionstyp MyPartitionScanner.COMMENT.

SourceViewerConfiguration definiert eine Reihe von Zugriffsmethoden für die unterschiedlichen Strategien, die in Hilfsobjekten entsprechenden Typs implementiert sind. In SourceViewerConfiguration selbst erzeugen diese Methoden lediglich Default-Implementierungen; um ein spezielles Verhalten zu erreichen, müssen sie überschrieben und eigene Implementierungen zurückgegeben werden.

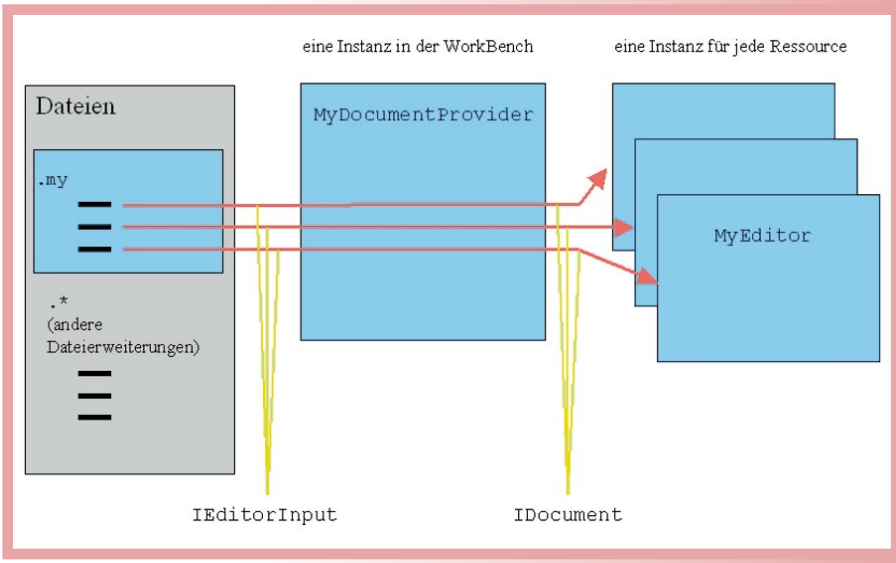

Abb 1: Der DocumentProvider erzeugt aus den Inhalten der Dateien Dokumente für den Editor

```
public IPresentationReconciler getPresentationReconciler(
final ISourceViewer sv ) {
  PresentationReconciler reconciler =
    new PresentationReconciler();

  DefaultDamagerRepairer defaultDr =
    new DefaultDamagerRepairer( getKeywordScanner() );
  reconciler.setDamager( defaultDr,
    IDocument.DEFAULT_CONTENT_TYPE );
  reconciler.setRepairer( defaultDr,
    IDocument.DEFAULT_CONTENT_TYPE );

  DefaultDamagerRepairer commentDr =
    new DefaultDamagerRepairer( getCommentScanner() );
  reconciler.setDamager( commentDr,
    MyPartitionScanner.COMMENT );
  reconciler.setRepairer( commentDr,
    MyPartitionScanner.COMMENT );

  return reconciler;
}
```

Listing 5: Konfiguration für das Syntax Coloring (Presentation Reconciling) in MySourceViewerConfiguration

```
public IContentAssistant getContentAssistant(
final ISourceViewer sv ) {
  ContentAssistant result = new ContentAssistant();
  result.setContentAssistProcessor(
    new KeywordCompletionProcessor(),
    IDocument.DEFAULT_CONTENT_TYPE );
  return result;
}
```

Listing 6: Konfiguration des Content Assistant in MySourceViewerConfiguration

Für die Syntaxfarben zuständig ist die Methode getPresentationReconciler(), deren Rückgabewert vom Typ IPresentationReconciler ist. Für das spezielle Syntax Coloring in unserem Editor müssen wir also diese Methode überschreiben und einen geeigneten Reconciler zurückgeben. Dazu wird ein Objekt vom Typ PresentationReconciler erzeugt. Damit verständlich wird, was genau nun mit diesem Reconciler geschieht, sind jedoch noch einige Hintergrundinformationen zum grundlegenden Konzept des Syntax Coloring in Eclipse angebracht.

Presentation Reconciling

Während der Benutzer den im Editor dargestellten Text bearbeitet, ändert sich eine bestimmte Region des Codes hinsichtlich der Einfärbung. Wird z. B. ein Schlüsselwort geschrieben, dann bleibt das eingegebene Wort zunächst in der Standardfarbe, wechselt aber dann zu einer anderen (der Farbe für Schlüsselworte), sobald es vollständig und damit als Schlüsselwort erkennbar ist. Diese Situation ergibt sich oft sogar hinsichtlich größerer Passagen als nur einzelner Worte. Fügt der Benutzer inmitten des Textes ein öffnendes Kommentarzeichen ein, dann wird der gesamte nun folgende Quellcode (oft eine beträchtliche Textportion) bis zum nächsten schließenden Kommentarzeichen als Kommentar betrachtet und entsprechend behandelt („auskommentiert“) – schreibt der Benutzer wiederum einige Worte und schließt den Kommentar dann, so wechselt auch die Präsentation des Textes zurück zu einer Darstellung als Quellcode.

Der Editor benötigt also einen Mechanismus, mit dessen Hilfe er bestimmen kann, welche Textregionen von Änderungen betroffen sind und daher neu gezeichnet werden müssen. Diese Aufgabe erledigt ein so genannter *Damager*. Anschließend muss geklärt werden, wie die so bestimmten Regionen nun neu eingefärbt werden sollen. Dafür ist ein *Repairer* zuständig. Der gesamte Vorgang wird als *reconciling* („versöhnen“) bezeichnet.

Pro Partitionstyp muss dem Reconciler ein Damager und ein Repairer angegeben werden. Im Beispiel (s. Listing 5) werden beide mit Hilfe der Plattform-Klasse DefaultDamagerRepairer zusammengefasst. Dieser Repairer arbeitet wieder mit einem Scanner zusammen, der auf dem Partitionstyp arbeitet. Für Kommentare gibt es einen CommentScanner, für einfachen Text einen KeywordScanner, der Schlüsselworte erkennt.

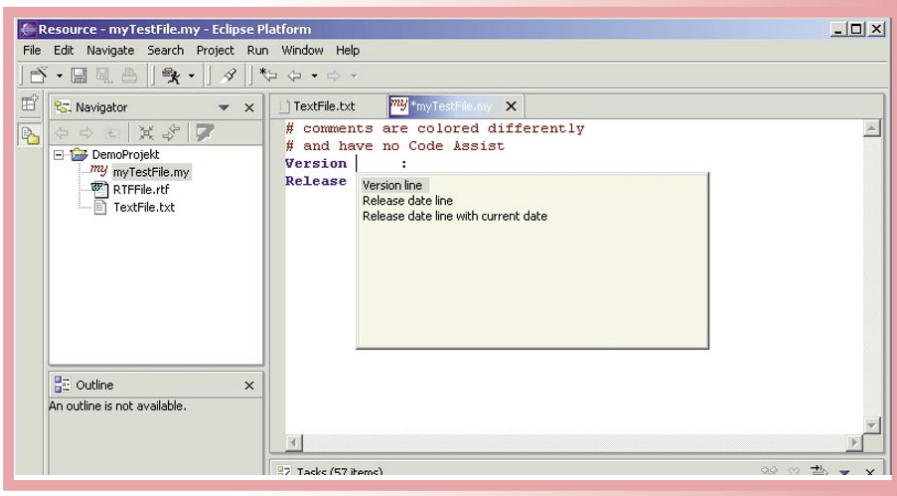

Abb 2: Das Beispiel-Plug-In in Aktion

Rufen Sie Ihren Assistenten – Content Assist aktivieren

Auch die automatische Code-Vervollständigung setzt ein partitioniertes Dokument voraus. In der SourceViewerConfiguration wird ein *Content Assistant* konfiguriert, der die entsprechenden Funktionalitäten implementiert. Da dies stark von dem jeweils bearbeiteten Dokument abhängt, gibt es dafür im Standard-Texteditor kein generisches Verhalten. Die zuständige Methode ist hier getContentAssistant() (s. Listing 6).

Wie auch schon beim Syntax Coloring wird im Beispiel die Standardimplementierung von IContentAssistant benutzt: ContentAssistant. Für jeden der Partitionstypen, innerhalb dessen der ContentAssistant tätig werden soll, muss nun wiederum eine Strategie spezifiziert werden, nach welcher der ContentAssistant verfahren soll.

In unserem Fall ist dies nur bei einfachem Text (dessen Typ durch IDocument.DEFAULT_CONTENT_TYPE angegeben wird) erwünscht: Der einfache KeywordCompletionProcessor soll auf Kommentaren nicht zur Verfügung stehen. Er implementiert IContentAssistProcessor und stellt auf Anfrage eine Liste von Vorschlägen zum Einfügen in den Text bereit (s. computeCompletionProposals()). Diese Liste soll erscheinen, wenn der Benutzer die Tastenkombination CTRL-SPACE ausführt (vorausgesetzt, er befindet sich im Text an der passenden Position, in diesem Fall: nicht über einem Kommentar). Wählt der Benutzer aus der Liste, wird das zugehörige Element in den Text eingefügt. Der Prozessor ist also für die Inhalte der Ergänzungsvorschläge verantwortlich. Weitere Funktionen von Content Assist, beispielsweise kontextsensitive Ergänzungsvorschläge bei Eingabe eines speziellen Zeichens wie etwa „.“, finden sich in [JEd].

Zuweisen der Tastenkombination

Abgesehen von der Implementierung des Prozessors selbst muss nun die zum Code Assist gehörige *Action* beim Editor registriert werden, damit die Liste der Vorschläge auch tatsächlich erscheint. Dies geschieht durch Überschreiben von createActions() in MyEditor (s. Listing 7). Diese Methode wird bei der Initialisierung des Editors aufgerufen und erzeugt dann eine TextOperationAction (s. Listing 8), welche durch die Tastenkombination CTRL-SPACE aufgerufen wird.

```
protected void createActions() {
  super.createActions();
  // wir konfigurieren eine Action für Content Assist, die
  // ausgeführt wird, wenn der Benutzer CTRL-SPACE eingibt
  IAction action = createAction();
  String actionId =
  ITextEditorActionDefinitionIds.
    CONTENT_ASSIST_PROPOSALS;
  action.setActionDefinitionId( actionId );
  setAction( "ContentAssistProposal", action );
}

private IAction createAction() {
  ResourceBundle bundle =
    JSpektrumPlugin.getDefault().getResourceBundle();
  // TextOperationAction gehört zu den Actions in der
  // Eclipse-Workbench, die sich selbst konfigurieren;
  // die benötigte Information wird aus dem übergebenen
  // ResourceBundle ausgelesen, verwendet werden hier
  // diejenigen Properties, deren Schlüssel mit
  // "ContentAssistProposal." beginnt
  return new TextOperationAction( bundle,
    "ContentAssistProposal.",
    this, ISourceViewer.CONTENTASSIST_PROPOSALS );
}
```

Listing 7: Registrieren einer Action für das Content Assist beim Editor

```
# Actions
ContentAssistProposal.label=Content Assist@Ctrl+SPACE
ContentAssistProposal.tooltip=Content Assist
ContentAssistProposal.image=
ContentAssistProposal.description=Content Assist
```

Listing 8: Die Konfiguration der TextOperationAction im ResourceBundle der Plug-In-Klasse

Ausblick

Viele weitere Ergänzungen der Funktionalität eines Editors sind möglich, die hier beschriebenen Verfahren stellen nur einen kleinen Ausschnitt der in der Plattform bereits angebotenen Unterstützung vor. Weitere Funktionen des Editors können über die SourceViewerConfiguration gesteuert werden, beispielsweise das Verhalten beim Doppelklick des Benutzers auf Textelemente, automatisches Einrücken (indentation) und andere.

Action Contributions des Editors können der Menüleiste und den Toolbars der Workbench hinzugefügt werden, oder die Struktur des angezeigten Dokuments kann in dem Outline Viewer als Liste oder Baum dargestellt werden. Indem eigene Preference Pages für den Editor implementiert werden, kann dem Benutzer die Möglichkeit gegeben werden, diesen für seine bevorzugte Arbeitsweise anzupassen.

Literatur und Links

[Bsp] Die begleitende Beispielimplementierung zu diesem Artikel steht unter
http://sigs-datacom.de/sd/publications/js/2003/06/index.htm
zum Download bereit.

[GoF96] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1996

[JEd] Vom Eclipse-Projekt gibt es ein Beispiel-Plug-In für einen Java-Editor, das u. a. demonstriert, wie unterschiedliche Partitionen (Java-Code, JavaDoc-Kommentare usf.) mit unterschiedlichen Strategien behandelt werden können. Das Plug-In kann unter http://eclipse.org/downloads/ heruntergeladen werden.

[JS03] D. Bäumer, D. Megert, A. Weinand, Eclipse Plug-Ins: Von der Idee zum Download, in: JavaSPEKTRUM, CEBIT 2003, Einführender Artikel zur Plug-In-Programmierung mit PDE (Plugin Development Environment)

[XEd] Der Eclipse-Wizard zum Erstellen neuer PDE-Projekte generiert auf Wunsch eine Vorimplementierung eines Editors für XML-Dateien, u.a. mit Beispielen für Syntax Coloring mittels regelbasierten Scannens und einer Doppelklick-Strategie.

Leif Frenzel ist seit mehr als zwei Jahren Mitglied des Kernentwicklungsteams für Entwicklungswerkzeuge bei der INNOOPRACT Informationssysteme in Karlsruhe. Er ist außerdem Autor von Plug-Ins zur Unterstützung funktionaler Programmiersprachen (wie Haskell, Scheme und Objective Caml). E-Mail: lfrenzel@innoopract.de.

Weiterführende Informationsquellen

http://eclipse.org/articles/
http://sigs-datacom.de/sd/publications/js/2003/06/index.htm