

eclipse

MAGAZIN

www.eclipse-magazin.de

Mit
CD

Alle Infos S. 3



Tools, Open Source & more

- » Eclipse Visual Editor Project
- » Apache Tomcat 6.0
- » ATLAS Transformation Language
- » Eclipse Modeling Framework (EMF)
- » Eclipse Graphical Modeling Framework (GMF)
- » Graphical Editing Framework (GEF)
- » Standard Widget Toolkit (SWT)



BONUS:

Artikelserie aus dem Java Magazin „OSGi applied“
+ Bonusartikel aus dem Eclipse Magazin Vol. 12

Professionell Testen

Best Practices: GUI-Tests für Eclipse RCP

» Embedded Eclipse:
Model Driven Development
bei eingebetteten Systemen

Praxis

Build-Prozess von Eclipse Plug-ins

Rich Clients

Neu seit Eclipse 3.3:
Eclipse DataBinding

Model Driven

Dokumentationen generieren

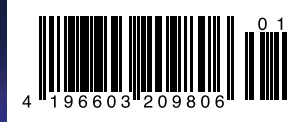
Plattformen

Eclipse und NetBeans Platform im Vergleich



Datenträger enthält nur
Lehr- oder Infoprogramme

D 68864



4 196603 209806

0 1

Customizing eines mit GMF generierten Diagrammeditors

Komplett effizient

>> ANDREAS SCHUSTER

Das Eclipse Graphical Modeling Framework (GMF) ermöglicht es, mit geringem Aufwand Grundgerüste für grafische DSL-Editoren zu generieren, allerdings fehlen Funktionen wie der Content Assist. Mithilfe des individuellen Customizing lassen sich derartige Komfortfunktionen in den eigenen DSL-Editoren jedoch gut nachrüsten. Am Beispiel eines mit GMF generierten Klassendiagramm-Editors wird hier gezeigt, wie sich dieser mit Content Assist erweitern lässt und es wird eine Einführung in die Grundlagen des GMF Customizings gegeben.

Die beiden vorangegangenen Artikel [1], [2] haben die Grundlagen von GMF [3] bereits eingehend beschrieben. Es sollen an dieser Stelle dennoch kurz die Zusammenhänge rekapituliert werden: GMF, die Verschmelzung der Eclipse-Projekte EMF [4] und GEF [5], ermöglicht es, auf Basis eines EMF-Metamodells und mithilfe weiterer Konfigurationsmodelle grafische DSL-Editoren komplett zu generieren. GMF selbst teilt sich dazu in Tooling- und Runtime-Komponenten auf. Erstere enthalten alle Werkzeuge, mit denen sich der Editor konfigurieren und generieren lässt. Dazu existieren vier Konfigurationsmodelle. Das *Gmfgraph*-Modell spezifiziert die grafischen Elemente, das *Gmftool*-Modell definiert die Werkzeugpalette, und das *Gmfmap*-Modell führt die Elemente aus *Gmfgraph*, *Gmftool* und Domänen-Modell zusammen, woraus schließlich das *Gmfgen*-Modell erzeugt wird, um den Code-Generator zu steuern. Das Resultat ist ein prototypischer, grafischer Modell-Editor, der eine ideale Ausgangsbasis für individuelle Erweiterungen bietet. Genau hier setzt dieses Tutorial an und integriert eine Content-Assist-Funktion in den Editor (Abb. 1).

Vorgehensweise

Zunächst werden wir den GMF-Editor als Grundlage für unsere Erweiterung

generieren und darauf aufbauend unsere Erweiterung planen, gefolgt von einer kurzen Analyse der zur Auswahl stehenden SWT/JFace-Komponenten [6] für Content Assist. Anschließend erörtern wir, an welchen Punkten im Code eingegriffen werden muss. Im Anschluss wird demonstriert, wie sich das GMF API für Erweiterungen nutzen lässt und wie unser Content Assist sich sinnvoll in den GMF-Editor integriert. Hierzu werden wir im Detail auf die Im-

plementierung der einzelnen Klassen und ihre Integration in den generierten Code eingehen und abschließend den GMF-Editor und die erweiterten Funktionen testen.

Domänenmodell

Wie in jedem GMF-Projekt, so bildet auch in unserem Projekt ein EMF-Domänen-Modell die Grundlage des Editors. Der Einfachheit halber verwenden wir dazu das bereits vorhandene EMF-Modell (Ecore), welches in jeder GMF-fähigen Eclipse-Umgebung als Plug-in vorhanden ist. Aufgrund der Größe des EMF-Modells wird der GMF-Editor nur einen Ausschnitt des kompletten Modells abbilden (Abb. 2). Dies dient vor allem der besseren Übersicht im Code und zur Veranschaulichung der Content-Assist-Integration.

Basis-Installation

Bevor wir mit dem Tutorial beginnen können, benötigen wir Eclipse 3.3 (Europa) mit GMF 2.0 inklusive aller

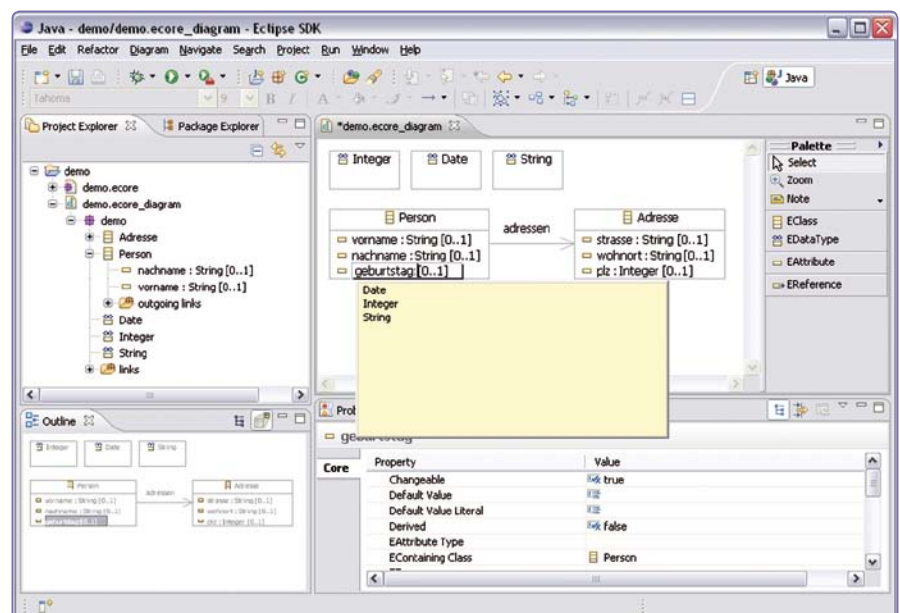


Abb. 1: Beispiel des erweiterten GMF-Editors mit aktivem Content Assist



Abhängigkeiten. Die Plug-ins werden idealerweise über die bereits vorkonfigurierte Eclipse Europa Update Site heruntergeladen. Zu beachten ist, dass wir das EMF Plug-in mit allen Sourcen benötigen, sodass wir das EMF SDK auswählen. Nachdem Eclipse gestartet ist, wird über FILE | IMPORT | PLUG-IN DEVELOPMENT | PLUG-INS AND FRAGMENTS zunächst das Plug-in *org.eclipse.emf.ecore* als Source-Projekt in den Workspace importiert. In diesem Plug-in befindet sich das Ecore-Modell, für welches wir im nächsten Schritt den GMF-Editor generieren. Dazu benötigen wir sämtliche GMF-Konfigurationsmodelle. Diese befinden sich auf der Heft-DVD im Verzeichnis /GMFMODEL des Projekts *org.eclipse.emf.ecore*. Ist das Verzeichnis in den Workspace kopiert, kann aus dem Generatormodell (*ecore.gmfgen*) der Diagramm-Code generiert werden. Nach einem erfolgreichen Generatordurchlauf sollte ein neues Projekt mit dem Namen *org.eclipse.emf.ecore.diagram* im Workspace angelegt worden sein.

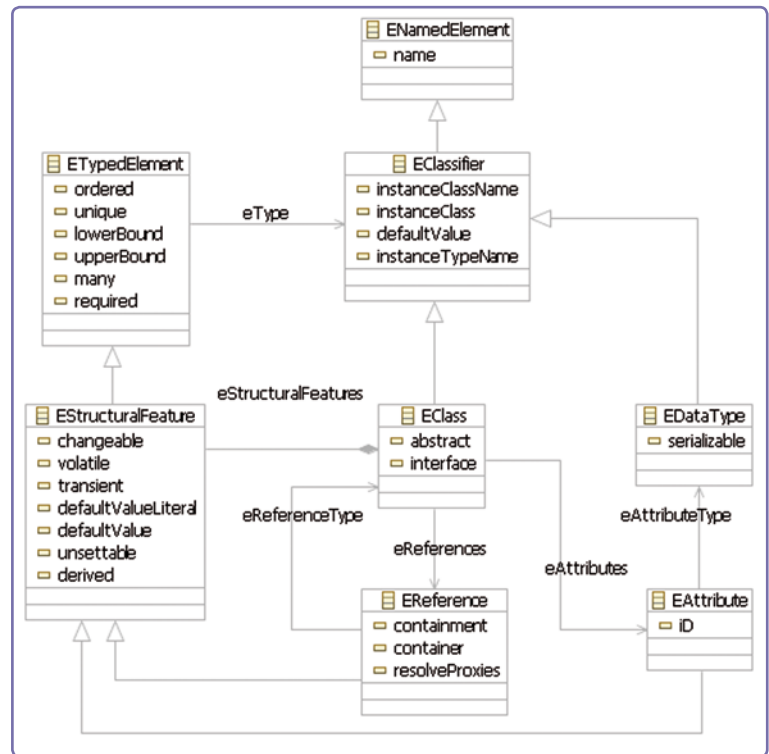
Spezifikation der Erweiterung

Mit dem generierten GMF-Editor haben wir nun eine solide Ausgangssituation geschaffen, um mit der eigentlichen Erweiterung beginnen zu können. Unsere Erweiterung soll die folgende Funktionalität in den Editor integrieren:

- Das Label zur Abbildung eines EAttribute soll zusätzlich zu *name* die Modellwerte *upperBound*, *lowerBound* und *eAttributeType* beinhalten.
- Die Darstellung soll der Form `<name>:<eAttributeType>[<lowerBound>..<upperBound>]` entsprechen.
- Alle Werte sollen auf dem Label direkt editierbar sein.
- Für die *eAttributeType*-Werte soll eine Content-Assist-Funktion zur Verfügung stehen. Als *eAttributeType* dürfen alle im ResourceSet des Diagramms enthaltenen *EDataTypes* verwendet werden.

Neben der Content-Assist-Funktion gibt es noch eine weitere Besonderheit, die der GMF-Editor ab Werk nicht zur Verfügung stellt, nämlich die Abbildung von Referenzen auf Label. Ein *EAttribute* referenziert *EDataTypes*. Ein Problem wird es also sein, diese Referenz

Abb. 2: Ausschnitt des EMF-Ecore-Modells



für die Abbildung in einen Namen aufzulösen, oder umgekehrt daraus wieder eine Referenz zu einem *EDataType* herzustellen.

Komponentenauswahl

Eclipse bietet innerhalb SWT/JFace [6] zwei verschiedene Komponenten für Content Assist an. Auf der einen Seite das Package *org.eclipse.jface.text.contentassist*, welches hauptsächlich im Bereich der textuellen Editoren (z.B. beim Java-Editor) zum Einsatz kommt. Auf der anderen Seite gibt es das Package *org.eclipse.jface.fieldassist*, welches für den punktuellen Einsatz von Content Assist geeignet ist, wie z.B. bei Textfeldern in Formularen oder bei Labels von GMF-Diagrammen. Im Vergleich gestalten sich die *contentassist*-Komponenten komplizierter in der Realisierung als *fieldassist*. Letzteres ist dagegen aufgrund seiner schlanken und einfachen Implementierung ideal für die Integration in einen GMF-Editor.

Interessanterweise bietet das GMF API momentan nur Unterstützung für das *contentassist* API an. Darüber hinaus beschränkt sich die „Unterstützung“ auf die Generierung leerer Methoden. Die eigentliche Implementierung des Completion-Prozessors müsste auch hier per Hand nachgereicht werden. Ohne große Umstände lässt

sich aber dennoch das *fieldassist* API in den Editor integrieren.

Prinzipiell sind aus dem Package *org.eclipse.jface.fieldassist* drei Komponenten von Bedeutung. Dazu gehört der *ContentProposalAdapter*, der das SWT Widget für Content Assist instanziiert. Die erweiterte Version *ContentAssistCommandAdapter* ermöglicht eine noch komfortablere Handhabung von Content Assist. Hinzu kommen die Interfaces *IContentProposalProvider* und *IContentProposal*. Dieser Provider versorgt den Proposal-Adapter mit einer entsprechenden Liste von Content Proposals, die der Benutzer letztlich bei aktivem Content Assist als Pop-up-Liste angezeigt bekommt.

GMF-Architektur

Bevor wir mit der Erweiterung unseres GMF-Editors beginnen können, werfen wir einen Blick auf die GMF-Architektur und identifizieren die Punkte, an denen wir für unsere Erweiterung eingreifen müssen. Zentrales Element jedes generierten GMF-Editors ist das *GEF EditPart*, das von der GMF Runtime über das Interface *IGraphicalEditPart* erweitert wird. *EditParts* übernehmen die Rolle des Controllers aus dem bekannten MVC (Model View Controller) Pattern (Abb. 3). Sie haben die Aufgabe, das Modell zu lesen, Listener

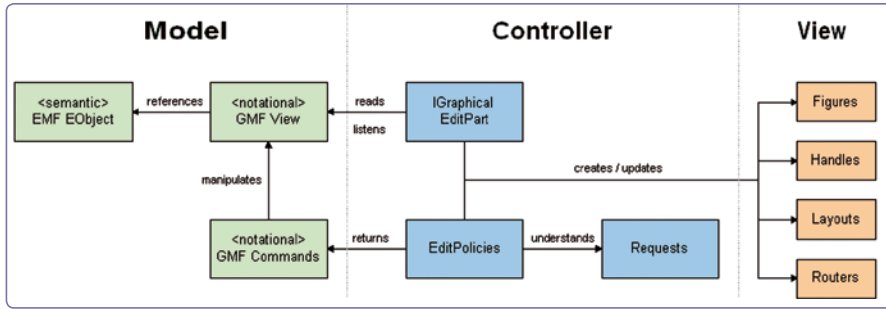


Abb. 3: GMF Model View Controller

am Modell zu registrieren oder auf Benutzereingaben zu reagieren. Die vom Benutzer initiierten Anfragen, auch Requests genannt, werden allerdings nicht vollständig vom *EditPart* übernommen, sondern überwiegend an die so genannten *EditPolicies* delegiert. *EditPolicies* sind spezifische Helper-Klassen, die im wörtlichen Sinne Richtlinien für das Editierverhalten von *EditParts* definieren. Beispielsweise existiert eine *EditPolicy* (*LabelDirectEditPolicy*) für das Editierverhalten von Label. Erhält die *EditPolicy* eine entsprechende Anfrage, werden über das reflektive GMF API entsprechende Commands erzeugt, die das Modell manipulieren oder die Sicht verändern. Diese Commands wiederum werden nicht immer direkt innerhalb der *EditPolicy* erstellt. Oft existieren zusätzliche Komponenten, welche die spezifische Editierfunktionalität implementieren und damit auch die entsprechenden Commands für die *EditPolicy* erzeugen. Im Falle eines Labels sind diese Komponenten ein JFace *CellEditor*, der ein direktes Editieren im Diagramm ermöglicht, und ein String Parser, welcher außer der Übersetzung zwischen lesbarem Label-Text und Modellelementen die entsprechenden Commands erzeugt. Sie sind, neben dem eigentlichen *EditPart*, die zwei Komponenten, an de-

nen wir eingreifen müssen, um unsere Erweiterung in den GMF-Editor einzufügen.

Customizing

Nachdem wir jetzt wissen, an welchen GMF-Komponenten wir eingreifen müssen, können wir die zusätzlichen Komponenten einplanen. Das GMF API lässt sich unter anderem dank der Verwendung von Interfaces sehr gut erweitern. Diese werden entweder durch Referenzimplementierungen der GMF Runtime, durch den GMF-Code-Generator oder per Hand implementiert. Das Klassendiagramm in Abbildung 3 zeigt einen Überblick der zu implementierenden (Custom) bzw. der zu modifizierenden (GMF-generierten) Klassen.

Ausgangspunkt der Erweiterung ist die Klasse *EAttributeEditPart*. Sie implementiert den Controller für das Label, welches mit Content Assist erweitert werden soll. *EditParts*, welche Text auf Label abbilden, implementieren das Interface *ITextAwareEditPart*. Damit wird sichergestellt, dass jedes Label auch einen *IParser* besitzt. Dieser Parser hat die Aufgabe, assoziierte Modellelemente in lesbare Strings umzuwandeln und daraus ggf. wieder Commands für Modelländerungen zu

erzeugen. Über die Factory *EcoreParserProvider* holt sich *EditPart* eine Instanz des passenden Parsers. Damit ein Label editiert werden kann, muss *EditPart* zunächst einen *TextDirectEditManager* instanziiieren. Dieser Manager wiederum instanziiiert den eigentlichen JFace *CellEditor*, welcher das Editieren im Diagramm ermöglicht. In der erweiterten Version *LabelDirectEditManager* wird das Content Assist Widget in Form des *ContentAssistCommandAdapter* schließlich an diesen *CellEditor* angehängt.

Für die Trennung von generiertem und handgeschriebenem Code erzeugen wir im Projekt *org.eclipse.emf.ecore* *.diagram* einen neuen Source Folder */SRC-CUSTOM*. Darin erstellen wir die Packages *org.eclipse.emf.ecore.diagram.parsers*, *org.eclipse.emf.ecore.diagram.part* und *org.eclipse.emf.ecore.diagram.providers*, in denen die nachfolgenden Custom-Klassen untergebracht werden. Generell gilt für alle generierten Klassen, in denen Methoden per Hand geändert werden, dass das `@generated` Tag auf `@generated NOT` geändert werden muss, um ein Überschreiben durch den Generator zu verhindern.

Datentypen cachen

Unsere Erweiterung mit Content Assist beinhaltet, dass wir die verfügbaren *EDataTypes* in einer Art Cache sammeln, um von beliebiger Stelle darauf zugreifen zu können. Der *EcoreDataTypeProvider* bietet diese Möglichkeit durch statischen Zugriff über die Member Variable *INSTANCE*. Zuvor sollte jedoch eine Initialisierung des Caches erfolgt sein, was in unserem Fall bei jedem Aufruf der Content-Assist-Funk-

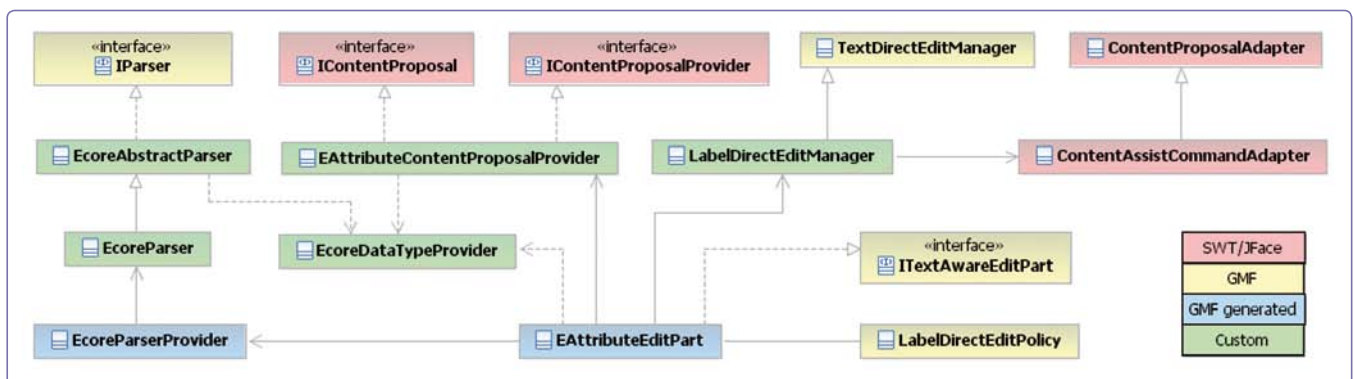


Abb. 4: Architektur der Erweiterung



tion vom *EditPart* aus erfolgt. Listing 1 zeigt die *init()*-Methode. Ihr wird die *EditingDomain* des *EditPart* übergeben, um damit an das *ResourceSet* des Diagramms und somit an die verfügbaren *EDataTypes* zu gelangen. Der Cache besteht aus einer *HashMap*, welche aus Key Value Pairs von *EDataType*-Namen sowie *EDataType* besteht. Dass einfache *EDataType*-Namen (im Gegensatz zu voll qualifizierten Namen) nicht eindeutig sind und ggf. überschrieben werden, soll für das Tutorial keine weitere Rolle spielen.

Label String Parser

Der *EcoreParser* hat zum einen die Funktion, aus den Modellelementen, die auf das Label abgebildet werden, einen lesbaren String zu bilden. Zum anderen sollen nach dem Editieren des Labels diese Änderungen auch wieder in das Modell überführt werden. Bezüglich der Abbildung von Modellelementen bringen die GMF-Tooling-Komponenten hier eine Einschränkung mit sich. Auf Label können nur Properties vom Metatyp *EAttribute* abgebildet werden. Diese Aussage verwirrt zunächst etwas, hängt jedoch mit der Beziehung von Modell und Metamodell zusammen. Betrachten wir das EMF-Metamodell, so wird deutlich, was damit gemeint ist. Unser EMF-Modell wird durch *Ecore* selbst definiert und ist als solches sein eigenes Metamodell. Wir arbeiten also mit einer Instanz des *Ecore*-Modells.

Unsere *EAttribute* Properties *name*, *lowerBound* und *upperBound* sind im Metamodell selbst als *EAttributes* definiert. Das Property *eAttributeType* hingegen entspricht im Metamodell einer *EReference*. Es hängt vermutlich mit der Komplexität einer generischen Implementierung für den GMF-Generator zusammen, dass GMF die Abbildung von *EReferences* (noch) nicht unterstützt.

Unser *EcoreParser* wird daher speziell für das *Ecore*-Modell angepasst werden. Die Implementierung des *EcoreAbstractParser* wurde in weiten Teilen aus der GMF-Referenzimplementierung *AbstractParser* übernommen. Die wichtigen Änderungen sind in den Methoden *getValues()* und *getValidNewValue()* zu finden. In der ersten wird vom *EDataType* zum *EDataType*-Namen übersetzt, was für die Ausgabe auf dem Label verwendet wird (Listing 2).

In der zweiten Methode wird der zum Namen passende *EDataType* zurückgegeben, um damit Modelländerungen durchzuführen. Dazu wird unser *EDataType* Cache nach entsprechenden Einträgen durchsucht (Listing 3).

Label Parser Factory

Die Instanziierung des *EcoreParser* erfolgt über die generierte Factory *EcoreParserProvider*. GMF vergibt beim Generieren des Codes für jedes *EditPart* Identifier, mit deren Hilfe die Factory entsprechende Objektinstanzen erzeugt

und zurückgibt. Außerdem werden diese Identifier auch für die Bildung von Klassen- und Methodennamen verwendet. Der GMF-Generator hat unserem *EAttributeEditPart* die ID 2001 zugewiesen. Die zu ändernde Methode, welche den Parser für das *EditPart* erzeugt, hat daher den kryptischen Namen *createEAttribute_2001Parser()*.

Der *EcoreParser* wird nun im Kontext des Modellelements *EAttribute* instanziiert und arbeitet unter Zuhilfenahme von Java *MessageFormat* Pattern und einer Liste von *EStructuralFeatures*, welche bei der Instanziierung als Parameter übergeben werden. Mithilfe dieser Liste lassen sich mittels des reflektiven EMF API jederzeit die Werte (Features) des Modellelements abfragen. Dieser generische Ansatz macht es möglich, den *EcoreParser* auch für andere Modellelemente zu verwenden, also auch für die Abbildung auf andere Label, indem jeweils eine passende Liste mit *EStructuralFeatures* übergeben wird.

Listing 4 zeigt die geänderte Methode. Zunächst wird entsprechend den *EAttribute* Properties *name*, *upperBound*, *lowerBound* und *eAttributeType* eine Liste mit *EStructuralFeatures* definiert. Wir merken uns dabei die Indizes der einzelnen Listenelemente, beginnend bei Null. Die Form des La-

Listing 1 ✕

Initialisieren des EDataType Cache

```
public void init(EditingDomain editingDomain) {
    if(editingDomain != null) {
        ResourceSet rs = editingDomain.getResourceSet();
        dataTypeMap = new HashMap<String, EDataType>();
        for(Resource r : rs.getResources()) {
            for(Iterator<EObject> it = r.getAllContents();
                it.hasNext();) {
                EObject eObject = it.next();
                if(eObject instanceof EDataType) {
                    EDataType dataType = (EDataType) eObject;
                    if(dataType.getName() != null) {
                        dataTypeMap.put(dataType.getName(),
                                        dataType);
                    }
                }
            }
        }
    }
}
```

Listing 2 ✕

Den name-Wert eines EDataTypes ausgeben

```
public Object[] getValues(EObject element) {
    Object[] values = new Object[features.toArray()
        .length];
    for (int i = 0; i < features.toArray().length; i++) {
        Object object = element.eGet(features.get(i));
        if (object == null) {
            values[i] = "";
        } else {
            if (features.get(i) instanceof EReference &&
                object instanceof EDataType) {
                values[i] = ((EDataType) object).getName();
            } else {
                values[i] = object;
            }
        }
    }
    return values;
}
```

Listing 3 ✕

Den EDataType zu einem name-Wert finden

```
protected Object getValidNewValue(
    EStructuralFeature feature, Object value) {
    EClassifier metaType = feature.getEType();
    [...]
    } else if (metaType instanceof EClass) {
        Class iClass = metaType.getInstanceClass();
        if (EClassifier.class.equals(iClass)) {
            if (value instanceof String) {
                value = EcoreDataTypeProvider.INSTANCE
                    .getDataTypeMap().get(value);
            } else {
                value = new
                    InvalidValue(NLS.bind(Messages.AbstractParser_
                        UnexpectedValueTypeMessage,
                            iClass.getName()));
            }
        }
    }
    return value;
}
```

bel-Strings wird über *MessageFormat* Pattern definiert. Es werden zwei Patterns definiert. Eines gibt die gültige Form beim Editieren vor und das andere die Form der normalen Diagrammansicht. An dieser Stelle kommen wir auf die Indizes der Features-Liste zurück. Die Value Indizes (in den geschweiften Klammern) des *MessageFormat* Pattern sind kongruent mit den Indizes der Features-Liste, was wiederum für die korrekte Formatierung des Labels von Bedeutung ist.

Content Assist vorbereiten

Eine wichtige Rolle der Content-Assist-Funktion übernimmt der *EAttributeContentProposalProvider*. Dieser hat die Aufgabe, abhängig vom Kontext des Labels eine Liste von *IContentProposals* zu erstellen. Ein *IContentProposal* hat folgende Merkmale: Es enthält den String, welcher in der Popup-Liste angezeigt wird, sowie eine mögliche Beschreibung dieses Strings. Hinzu kommen die Position des Cursors und der String, der nach erfolgreicher Selektion zurückgegeben wird.

Content Assist bedeutet gleichzeitig auch Code Completion, d.h. die Auswahl der *ContentProposals* wird durch die bereits getippten Zeichen gefiltert. Gibt es hier Übereinstimmungen, wird die Liste der Vorschläge entsprechend eingeschränkt bzw. durch die Selektion eines Vorschlags der Bezeichner vervollständigt. Besonders bei komplexeren Labels, wie in unserem Fall, wird hier die Notwendigkeit eines lokalen String Par-

sers deutlich, um Funktionen die Code Completion zu ermöglichen.

Content Assist instanziiieren

Für die Instanziierung des *ContentAssistCommandAdapter* müssen wir den *TextDirectEditManager* durch den *LabelDirectEditManager* erweitern. Dort überschreiben wir zunächst den Konstruktor und fügen ihm einen weiteren Parameter hinzu, damit bei der Instanziierung im *EditPart* der passende *EAttributeContentProposalProvider* übergeben werden kann. So behält man die Flexibilität, für verschiedene Label unterschiedliche Implementierungen von *IContentProposalProvider* zu verwenden. Für die Instanziierung des *ContentAssistCommandAdapter* überschreiben wir die Methode *initCellEditor()*, wie in Listing 5 beschrieben. Die Methode ist weitestgehend selbsterklärend. Das Popup Widget des *ContentAssistCommandAdapter* benötigt lediglich einen Ankerpunkt innerhalb des SWT-Komponenten-Baumes. Hierzu holen wir uns einfach die *Text*-Komponente des *CellEditor* und übergeben diese als Parameter bei der Instanziierung des *ContentAssistCommandAdapter*.

Da GMF nur Content Assist für das Package *org.eclipse.jface.text*.

contentassist unterstützt, ist ein kleiner Workaround in der Methode *commit()* erforderlich. Genau genommen handelt es sich sogar um einen Workaround für den Workaround. Er soll in erster Linie verhindern, dass der Fokus des *CellEditors*, respektive der Popup-Liste, verloren geht, wenn man die Maus zur Selektion eines Content Proposals verwendet. Der Workaround sucht nach dem momentan aktiven SWT Control und entscheidet dadurch, ob der Fokus gehalten wird oder nicht. Handelt es sich dabei um eine *Table*, so wird der Fokus gehalten. Die SWT Controls der Popup Windows von *jface.feldassist* und *jface.text.contentassist* unterscheiden sich lediglich durch eine tiefere Verschachtelung in ein weiteres *Composite*, wie Listing 6 zeigt.

Das EditPart

Nachdem nun vorangegangene Klassen erstellt sind, kann die generierte Klasse *EAttributeEditPart* entsprechend modifiziert werden. Dies geschieht, wie Listing 7 zeigt, in der Methode *getManager()*. Zunächst wird der *EcoreClassifierProvider* initialisiert. Anschließend wird der *EAttributeContentPropo-*

Listing 4 ✕

Instanziierung des Label Parsers

```
protected IParser createEAttribute_2001Parser() {
    List<EStructuralFeature> features = new ArrayList
        <EStructuralFeature>();
    features.add(EcorePackage.eINSTANCE
        .getENamedElement_Name());
    features.add(EcorePackage.eINSTANCE
        .getETypedElement_LowerBound());
    features.add(EcorePackage.eINSTANCE
        .getETypedElement_UpperBound());
    features.add(EcorePackage.eINSTANCE
        .getETypedElement_EType());
    EcoreParser parser = new EcoreParser(features);
    parser.setEditPattern("{0};{3}{1}..{2}");
    parser.setViewPattern("{0} : {3} {1}..{2}");
    return parser;
}
```

Listing 5 ✕

Instanziierung des Content Assist Widgets

```
@Override
protected void initCellEditor() {
    super.initCellEditor();

    Text text = (Text) getCellEditor().getControl();

    if(getProposalProvider() != null) {
        char[] autoActivationCharacters = new char[] { ':' };

        ContentAssistCommandAdapter contentAdapter =
            new ContentAssistCommandAdapter(
                text,
                new TextContentAdapter(),
                getProposalProvider(),
                ContentAssistCommandAdapter.CONTENT_
                    PROPOSAL_COMMAND,
                autoActivationCharacters);

        contentAdapter.setProposalAcceptanceStyle(
            ContentProposalAdapter.PROPOSAL_REPLACE);
        contentAdapter.setPropagateKeys(true);
        contentAdapter.setAutoActivationDelay(500);
    }
}
```

Listing 6 ✕

Workaround für Content Assist Popup-Fenster

```
@Override
protected void commit() {
    Shell activeShell = Display.getCurrent()
        .getActiveShell();
    if (activeShell != null && getCellEditor()
        .getControl().getShell().equals(
            activeShell.getParent())) {
        Control[] children = activeShell.getChildren();

        // Workaround begin
        Composite composite = (Composite) children[0];
        Control[] compositeChildren = composite
            .getChildren();

        // Workaround end

        if (children.length ==
            1 && compositeChildren[0] instanceof Table) {
            getCellEditor().getControl().setVisible(true);
            ((TextCellEditorEx) getCellEditor())
                .setDeactivationLock(true);

            return;
        }
    }
    super.commit();
}
```



salProvider instanziiert und die *ContentProposal*-Liste mithilfe des *EcoreClassifierProvider* gefüllt. Statt des *TextDirectEditManager* wird der erweiterte *LabelDirectEditManager* instanziiert und im Konstruktor der Proposal Provider mitgegeben.

Testen der Erweiterung

Nachdem nun alle Implementierungen abgeschlossen sind, können wir die Funktionalität des Editors testen. Dazu erstellen wir im Eclipse RUN-DIALOG unter der Kategorie ECLIPSE-APPLICATION einen neuen Eintrag und nennen diesen *GMF-Demo*. Der GMF-seitige Ecore-Editor *org.eclipse.gmf.ecore.editor*, der über die Eclipse Plug-ins mit installiert wurde, sollte im Register PLUG-INS unter TARGET-PLATFORM deaktiviert werden, damit wir die Ecore-Diagramme ausschließlich in unserem Editor bearbeiten können.

Sobald die Eclipse Target Platform gestartet ist, wird ein leeres Eclipse-Projekt erstellt. Anschließend rufen wir über das Menü FILE|NEW|EXAMPLE den Ecore Diagram Wizard auf, um damit ein neues Diagramm zu erstellen. Eclipse sollte nun ein leeres Diagramm zeigen. Für die weiteren Ausführungen orientieren wir uns am Beispieldiagramm aus Abbildung 1. Wir zeichnen die darauf enthaltenen *EDataTypes* und *EClasses*, dann fügen wir der *EClass* Person ein *EAttri-*

bute hinzu und klicken auf das Label, um es direkt zu editieren. Der *CellEditor* sollte bei einem neuen *EAttribute* den Inhalt:[0..1] markiert anzeigen, da weder *name* noch *eAttributeType* definiert wurden. Die Kardinalitäten *lowerBound* und *upperBound* besitzen hingegen Default-Werte, die bei der Initialisierung gesetzt wurden. Dieser Inhalt ist vergleichbar mit einer Mindestanforderung, die der *EcoreParser* auf Basis des *MessageFormat*-Patterns als gültig erachtet. Alle nicht gültigen Formen führen dazu, dass der Parser die Änderungen des Labels verwirft. Beim Editieren des Labels spielt es keine Rolle, ob wir den kompletten Inhalt löschen oder den Cursor entsprechend positionieren. Content Assist bzw. Code Completion erledigen für uns den Rest. Nachdem der Doppelpunkt eingefügt wurde, öffnet sich das Pop-up-Fenster mit den *EDataType* Proposals. Alternativ kann zum Öffnen des Pop-up-Fensters natürlich auch CTRL+SPACE gedrückt werden. Das Content Assist Pop-up listet alle bis dahin eingefügten *EDataTypes* auf. Tippen wir hingegen die ersten Buchstaben des gewünschten *EDataType*, wird die Liste gefiltert und entsprechend einschränkt. Wenn wir nun einen *EDataType* auswählen, wird dieser in das Label eingefügt. Sollten wir noch keine Kardinalitäten gesetzt haben, werden die Default-Werte automatisch angehängt. Wir fügen noch die weiteren *EAttributes* ein, um ein wenig mit den Möglichkeiten der Content-Assist-Funktion vertraut zu werden, und schließen damit den Test ab.

Fazit

Welcher Softwareentwickler könnte sich Eclipse noch ohne Content-Assist-Funktion vorstellen? Das Tutorial hat gezeigt, wie sich Content Assist auch in GMF-Editoren sinnvoll integrieren lässt, sodass man diese Funktionalität anschließend nicht mehr missen möchte.

GMF ermöglicht es, mithilfe seiner Konfigurationsmodelle und des integrierten Code-Generators in relativ kurzer Zeit prototypische Diagrammeditoren zu erstellen. Soll jedoch aus diesen Prototypen ein voll funktionsfähiges und komfortables Modellierungswerkzeug entstehen, ist es notwendig, für die einzelnen Komponenten gezieltes

Customizing zu betreiben. Dafür ist zunächst ein Verständnis der Funktionsweise des GMF API notwendig, das auf den GEF- und EMF-Komponenten aufbaut, was für den unerfahrenen Anwender ein nicht zu unterschätzender Zeitaufwand ist. Ist diese Hürde erst einmal genommen, ist es möglich, durch minimalinvasive Eingriffe an den generierten Artefakten seinen eigenen Code einzufügen. Oft reichen dafür nur wenige Zeilen Code an den richtigen Stellen des API, um die gewünschten Änderungen zu erreichen.

Mehrere Hersteller von Modellierungswerkzeugen setzen für ihre Produkte inzwischen auf die Möglichkeiten von GMF. Das unterstreicht, welches Potenzial sich in diesem Framework verbirgt. Viel wichtiger ist jedoch, dass GMF endlich die Lücke zu den Standardwerkzeugen schließt. Früher konnte ausschließlich mit den am Markt vorhandenen Modellierungswerkzeugen gearbeitet werden. Heute ist es möglich, sich mit GMF, angepasst an die individuellen Bedürfnisse, eigene Modellierungswerkzeuge zu bauen. Dass dies mit einem vertretbaren Aufwand auch für kleinere Entwicklungsteams möglich ist, macht GMF umso mächtiger.

Listing 7

Ausgangspunkt und Endpunkt der Erweiterung, das *EAttributeEditPart* in der Methode *getManager()*

```
protected DirectEditManager getManager() {
    // initialize the EDataType list
    EcoreDataTypeProvider.INSTANCE.init(
        getEditingDomain());
    if (manager == null) {
        // create a new ContentProposalProvider for the
        // ContentProposalAdapter
        EAttributeContentProposalProvider provider =
            new EAttributeContentProposalProvider();
        // instantiate the modified constructor in
        // LabelDirectEditManager
        setManager(new LabelDirectEditManager(this,
            TextDirectEditManager
                .getTextCellEditorClass(this), EcoreEditPartFactory
                .getTextCellEditorLocator(this), provider));
    }
    return manager;
}
```



Andreas Schuster arbeitet als Softwareentwickler bei der eXcellent solutions GmbH in Ulm und beschäftigt sich dort intensiv mit dem Einsatz und der Anwendung neuer Tools und Technologien in der modellgetriebenen Softwareentwicklung. Sein Schwerpunkt liegt dabei in der Realisierung und Integration eigener, grafischer DSL-Editoren. Kontakt: a.schuster@excellent.de.

>> Links & Literatur

- [1] Alexander Schwarz: Ein Bild sagt mehr als tausend Worte, in *Eclipse Magazin* Vol 8
- [2] Alexander Schwarz: Geordnete Verhältnisse, in *Eclipse Magazin* Vol 9
- [3] GMF: www.eclipse.org/gmf
- [4] EMF: www.eclipse.org/emf
- [5] GEF: www.eclipse.org/gef
- [6] SWT/JFace: www.eclipse.org/swt