1.Allgemeines

Wir beschäftigten uns während der Softwarestudie eingehender mit GMF 1.02. Ziel war, die Abhängigkeiten und Mechanismen hinter der Oberfläche zu verstehen, um sie im Projekt selbst verbessern zu können. Als Prototypen haben wir dafür drei verschiedene Editoren mithilfe von GMF entworfen, die als eigenständige eclipse-Plugins vorliegen.

2. Produktübersicht

Der erste Editor bietet die Möglichkeit, einen Zustandsautomaten mit Anfangs- und Endzustand sowie zwei Zwischenzuständen zu modellieren. Dieser hat bereits festgelegte Zustandsübergänge und davon höchstens 5.
Der zweite Editor bietet die Möglichkeit, aufbauend auf der gmfgraph.ecore, ein einfaches Rechteck auf der Arbeitsfläche darzustellen.
Der dritte Editor baut auf dem zweiten auf und bietet zusätzlich die Möglichkeit, mithilfe eines Labels im Rechteck die Farbe zu verstellen. Dort kann man „green“, „yellow“ oder „red“ eingeben und je nachdem ändert sich die Farbe des Rechtecks.

3. Grundsätzliche Struktur-und Entwurfsprinzipien für das Gesamtsystem

Ausschließlich die Javaklassen werden als eclipse-Plugin compiliert und ausgeführt.
Aber wie setzen sie sich zusammen? Da wir viel automatischen Code erzeugt haben, den wir kaum verändern mussten, beschreibe ich zuerst die Entstehung dieses Codes.
Der Grundaufbau aller eCores ist gleich: Es gibt ein Diagrammelement, das Referenzen auf alle anderen eClasses enthält, die nachher im Editor enthalten sein sollen. eClasses können dann sowohl Links als auch Nodes sein, darüber entscheidet man bei der Erstellung der GMF-Tool und GMF-Graph-Modelle. Änderungen am Tool-Modell waren nicht notwenig, jedoch haben wir im GMF-Graph davon Gebrauch gemacht, den Nodes verschiedene Formen und Farben zuzuweisen. Die drei Modelle eCore, GMF-Graph, GMF-Tool werden im nächsten Arbeitsschritt zum GMF-Map zusammengefügt. Dort wird endgültig festgelegt, welche Referenz durch welche Kante und welche eClass durch welches Node dargestellt wird.
Jetzt benutzen wir das EMF, um aus dem eCore eine .genmodel-Datei zu erstellen und mithilfe dieser die Java-Klassen für den eCore (im Modellpaket unter /src/) und den Editorcode (unter .edit) zu erstellen. Diese Javaklassen werden mit dem GMF-Map-Modell verbunden und daraus wird die GMF-Gen-Datei erzeugt, das Erzeugungsmodell für unser fertiges Plugin. Dieses erstellt die Java-Pakete unter .diagram.
Es war für die ersten beiden Editoren nicht notwendig, Java-Klassen zu verändern. Das ist erst beim dritten Editor passiert und dort haben wir ausschließlich Veränderungen im .diagram – Verzeichnis vorgenommen.

Kurzvorstellung der Pakete:

ColorRechteck/src	/gmfgraph		Interfaces zu allen Elementen (für Mehrfachvererbung)
	/gmfgraph.impl		komplette Klassen
	/gmfgraph.util		Hilfsklassen für das Plugin
.diagram/src/gmfgraph.diagram		.edit.commands	für die Verbindungen zw. Klassen
		.edit.helpers	Klassen um die element requests zu erstellen
		.edit.parts	enthält domänenspezifische Regeln für die Elemente
		.edit.policies	Regeln für das Editieren von Elementen
		.part	Wizards zum Erzeugen des Editordiagramms
		.providers	Ressourcenverwaltung
		.view.factories	Verwaltung der Layout-Styles des Graph-Modells
.edit/src	/gmfgraph.provider		Verwaltung der Elementeigenschaften des Domänenmodells

4.Grundsätzliche Struktur- und Entwurfsprinzipien der einzelnen Pakete

Erster Editor

Die Kanten im ersten Editor haben wir auch mit eClasses modelliert. Eine Referenz zeigt auf das Zielobjekt und eine auf das Quellobjekt und alle Kanten erben von einer Oberklasse Kanten. Nodes sollten dann die einzelnen Zustände werden. Im Graph wurde festgelegt, dass die Nodes als Ellipsen dargestellt werden: Anfangszustand Grün, normale Zustände gelb und Endzustand rot. Die Verbindungen dazwischen sind alle grau, haben aber ein festgelegtes Label, mit dem der gelesene Buchstabe verdeutlicht wird. Dadurch, dass alle Kanten eine eigene Klasse mit eigenen source- und target-features bekommen, kann nur je eine Kante zwei genau festgelegte Zustände verbinden. Es gibt eine Maximalanzahl von 5 für die Kanten, die wir als Zustandsübergänge unseres Automaten ansehen. Im GMF-Tool gibt es nur ein einziges Creation-Tool für die Kanten, das alle Link Mappings benutzen. Dennoch ist durch die Art der Referenzen in den einzelnen Kanten-eClasses genau festgelegt, was miteinander verbunden werden darf – z.B. gibt es keine Kanten auf den Anfangszustand.

Zweiter Editor

Die Übung ist einfach: Die mit dem Paket org.eclipse.gmf.graphdef importierte gmfgraph.ecore wird mit den anfangs beschriebenen Arbeitsschritten in ein lauffähiges Plugin umgewandelt.

Dritter Editor

Der ecore wird aus der zweiten Aufgabe übernommen, diesmal wird allerdings eine neue eClass erzeugt, die von Rectangle und ConstantColor erbt. So entsteht das ColorRectangle mit einem eigens gespeicherten Farbwert namens „value“. Dieser value wird als Feature-Attribut auf ein Label gemappt, so kann es nur Beschriftungen wie „black“ oder „red“ annehmen.
Ab jetzt haben wir ein fertiges Modell und haben nur noch einige Änderungen in den Javaklassen vorgenommen. Am wichtigsten sind die .diagram.edit.parts-Pakete, denn dort finden die Reaktionen der einzelnen Objekte auf die Eingaben des Users statt. Es wurde schnell auffällig, dass das Label sein zugehöriges Rechteck kennen sollte, um bei einer Textänderung auf diesem die „setBackgroundColor“-Methode auszuführen. So mussten wir ein neues Label vom „org.eclipse.gmf.runtime.draw2d.ui.figures.WrapLabel“ ableiten. Es bekam eine set-Methode, um die Referenz auf das ColorRectangle zu setzen und eine Hilfsvariable b, mithilfe derer die Beschriftung des Labels umgewandelt wird. Dafür haben wir noch eine Hilfsklasse namens „Farben“ programmiert, die wir von der erzeugten ColorRechteck/src/gmfgraph/ColorConstants.java – Klasse ableiteten. Denn aus dieser Klasse konnte komischerweise die richtige „org.eclipse.draw2d.ColorConstants“ nicht importiert werden. Außerdem mussten wir noch eine weitere Methode hinzufügen, mit der man Farben aus dieser ColorConstants-Klasse in Farben aus der richtigen Klasse konvertieren konnte, um sie dann dem

Rechteck zu übergeben. Diese heißt getColor. Nun musste wir noch die Methode „setText()“ des ColorLabels zu überschreiben, sodass die Farbe auch im Rechteck aktualisiert wird. Am Ende musste die createContents() in der ColorRectangle2EditPart so veröndert werden, dass das ColorLabel als Inhalt hinzugefügt wird und es den Anfangstext „white“ bekommt. Fertig!