EMF-GMF-Tutorial: Hintergrund

Schritte 1-12: Im ersten Teil haben wir die Sprache Petrinet definiert, indem wir ein Metamodell erstellt haben. Damit ist die grundsätzliche Struktur valider Petrinet-Modelle festgelegt. Dazu haben wir die Metametamodellierungssprache Ecore benutzt, eine Implementierung des MOF-Standards der OMG – sie ist Teil des Eclipse Modeling Framework (EMF).

Schritte 13-16: Aus Ecore-Modellen wollen wir nun Code generieren. Um Anpassungen am generierten Code vornehmen zu können, wird dazu zunächst ein Generatormodell (.genmodel) generiert und angepasst. Anpassungsmöglichkeiten schließen die Generierung von Code für unterschiedliche Plattformen ein (RCP, RAP, GWT). Nun können verschiedene Plug-ins generiert werden:

- Das Modell-Plugin enthält den eigentlichen Modellcode. Dieser unterstützt u.a. effiziente Reflections (PetrinetPackage), Notification und Serialisierung. Er ist in Interfaces und deren Implementierungen (Paket impl) aufgeteilt, um u.a. Mehrfachvererbung zu ermöglichen.
- Das Edit-Plugin enthält Klassen, die es erlauben, Modellelemente auf Basis eines Command-Fameworks zu editieren (Undo-Unterstützung) sowie innerhalb von Eclipse-GUI-Komponenten darzustellen.
- Das Editor-Plugin enthält einen Baumeditor, mit dem Petrinet-Modelle editiert werden können, und basiert auf dem Edit-Plugin.

Schritte 17-19: Um den generierten Code ausführen zu können, müssen wir eine Runtime-Workbench starten. Dies ist im Wesentlichen eine Kopie der Development-Workbench, zusätzlich laufen in ihr aber noch die Plugins aus unserem Development-Workspace (also auch die generierten Plugins).

Schritte 20-24: Das Editor-Plugin fügt einen Wizard zur Erstellung von Petrinet-Modellen zur Runtime-Workbench hinzu. Diesen nutzen wir, um unser Modell zu initialisieren, und editieren dieses dann.

Schritte 25-40: Im nächsten Schritt benutzen wir das Graphical Modeling Framework (GMF), um einen graphischen Editor zu generieren. Dabei hilft uns das GMF-Dashboard, indem es uns durch die Erstellung der für die Beschreibung des graphischen Editors nötigen Modelle führt. Wir benötigen folgende Modelle:

- .gmfgraph: Ein Modell, das das Aussehen von Modellelementen festlegt. Im Wesentlichen unterscheidet GMF zwischen Node-Elementen und Connection-Elementen, die Node-Elemente verbinden. Man kann das Aussehen der Elemente festlegen (Form der Node-Elemente, Art/Decorations der Connection-Elemente).
- .gmftool: Ein Modell, das die Werkzeuge festlegt, mit denen unsere Modellelemente erzeugt bzw. editiert werden können, inkl. Labels und Icons.
- .gmfmap: Ein Mapping-Modell. Dieses führt die anderen Modelle zusammen: Es ordnet Elementen aus dem Metamodell ein graphisches Element und ein Werkzeug zu. GMF versucht, ein sinnvolles Mapping-Modell zu generieren. Dieses muss aber meist per Hand nachbearbeitet werden.

Wiederum erfolgt die Codegenerierung über den Umweg eines Generatormodells (.gmfgen).

Schritte 41-45: Zur Ausführung starten wir erneut die Runtime-Workbench. Dort haben wir nun einen graphischen Editor zur Verfügung, mit dem wir Petrinet-Modelle erzeugen können. Dieser enthält rechts die Werkzeuge aus unserem Modell Petrinet.gmftool. Der Zeichenhintergrund repräsentiert das Petrinet. Die graphischen Elemente des Editors entsprechen dem Modell Petrinet.gmfgraph: Die Veränderungen, die wir vorgenommen haben, bewirken, dass a) Places als Ellipsen und Tokens als ausgefüllte Ellipsen dargestellt werden (Schritt 32f), b) die Pfeilspitze der sourcePlaces-Referenz entgegen der Navigationsrichtung gezeichnet wird (Schritt 31) und c) Tokens zu Places hinzugefügt werden können (Schritte 29f, 36, 37).

GMF-Editoren können auf verschiedene Art und Weise angepasst werden:

- Zunächst sollte man ausnutzen, was die Modelle unterstützen (z.B. wie in Abb. 10 des Tutorials). Vorteil: Man kann einfach neu generieren und muss sich keine Gedanken darüber machen, ob manuell angepasster Code überschrieben wird. Außerdem muss man dafür die GMF-API nicht kennen.
- GMF verwendet EditParts. Diese (und vieles andere) können angepasst werden, in dem man in einem separaten Plugin (z.B. de.upb.agengels.se.petrinet.editor.customization) eigenen Code schreibt und diesen dann zur Laufzeit dem generierten Editor mittels Extension Points „unterschiebt“. Vorteil: strikte Trennung von generiertem und programmiertem Code.
- Die Codegenerierung von GMF basiert auf Templates. Diese können auf Klassenebene angepasst werden.
- Schließlich kann man auch den generierten Code direkt anpassen. Z.B. sind generierte Methoden mit „@generated“ annotiert – wenn man daraus ein „@generated not“ macht, wird die entsprechende Methode beim Neugenerieren nicht überschrieben.

Man sollte versuchen, auf Codeanpassungen und möglichst auch Anpassungen der Templates zu verzichten, da das die Wartung erleichtert (Umstieg auf neue GMF-Versionen).

Die konsequenteste Art der modellgetriebenen Softwareentwicklung ist, Generiertes nicht ins Repository einzustellen – das ist aber nicht immer praktikabel. Um Anpassungen an das .gmfgen-Modell zu automatisieren, kann man z.B. auf die GMFTools zurückgreifen (siehe unten).

Literatur

D. Steinberg, F. Budinsky, M. Paternostro, E. Merks: **EMF: Eclipse Modeling Framework (2nd Edition).** Addison-Wesley Longman, Amsterdam; ISBN-10: 0321331885

R. C. Gronback: **Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit.** Addison-Wesley Longman, Amsterdam; ISBN-10: 0321534077

http://www.eclipse.org/modeling/gmp/

http://www.eclipse.org/modeling/emf/

http://code.google.com/p/gmftools/