Entwicklung von Generator-Plug-ins mit Eclipse und openArchitectureWare

Plug-ins modellgetrieben

>> PETER FRIESE

Speaker @ jax

Quellcode auf CD

Viele Wizards – wie etwa der Wizard zum Anlegen einer neuen Java-Klasse – dienen als Frontend für einen Codegenerator. Die meisten dieser Generatoren bestehen aus mehr oder weniger umfangreichem Java-Code. In diesem Artikel wird mit einem möglichen Weg zum Einsatz ausgereifter Generatortechnologien in Wizards gezeigt, dass es auch anders geht.

Ohne es zu merken, nutzen die meisten Entwickler heutzutage Codegenerierung. So verwendet zum Beispiel jeder Anwender des Eclipse JDT – also die Mehrzahl der Eclipse User – den Eclipse Wizard zum Anlegen von neuen Projekten. Dieser Wizard erzeugt nach den Vorgaben des Benutzers ein neues Projekt und konfiguriert es für die Verwendung mit Java. Andere Wizards gehen deutlich weiter und erzeugen Rohgerüste für Eclipse Plug-ins oder gar ganze RCP-Applikationen. Die Vorgaben des Benutzers (wie z.B. der Projektname, zu erzeugende Extensions, zu verwendende Java-Version oder Ort und Name von Quellcode- und Binärordner) kann man bei genauerer Betrachtung eigentlich als Modell auffassen.

Listing 1

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="SimpleJavaProject"
                                default="compile">
 <description>Build file for SimpleJavaProject
                                </description>

 <target name="compile">
  <javac srcdir="src" destdir="bin"/>
 </target>

 <target name="jar" depends="compile">
  <mkdir dir="dist"/>
  <jar destfile="dist/SimpleJavaProject.jar"
                                basedir="bin">
  </jar>
 </target>
</project>
```

Wizards bieten dem Anwender oftmals eine Vielzahl von Optionen, aus denen er auswählen kann. Je mehr Optionen ein Wizard anbietet, desto komplizierter wird der Code des Wizards selbst. Ein Blick hinter die Kulissen offenbart, dass die meisten Wizards ihre Zielartefakte wie Projekte, Ordner und Java-Klassen durch Aufruf von APIs erzeugen. Dies führt dazu, dass der zu erzeugende Zielcode und die Steuerungslogik für die Erzeugung des Zielcodes eng miteinander verwoben sind. Lesbarkeit und Wartbarkeit leiden darunter. Die Situation ist ähnlich wie in den Anfangstagen der Entwicklung von Java-Webanwendungen, als zur Realisierung von Webanwendungen nur Servlets zur Verfügung standen. Auch hier wurde die Anwendungslogik mit der Präsentation – in diesem Fall HTML – vermischt. Mit dem Aufkommen von Servlets änderte sich die Situation dann, da nun Logik und Präsentation getrennt werden konnten (mit der Betonung auf „konnten“, denn auch in Servlets können Logik und Präsentation vermischt werden). Anhand eines einfachen Beispiels soll in diesem Artikel gezeigt werden, wie solche in Eclipse integrierten Codegeneratoren durch den Einsatz der Tools aus dem Eclipse Modeling Project vereinfacht werden können.

Als Beispiel soll die Erzeugung einer Ant-Datei dienen. Ant [1] ist trotz (oder gerade wegen?) seines Alters immer noch eine gute Wahl für die Automatisierung von Build-Prozessen. Umso mehr verwundert es, dass Eclipse keinen Wizard bietet, der anhand eines bestehenden Projekts eine passende *build.xml*-Datei für Ant erzeugt. Ein solcher Wizard soll in diesem Artikel entwickelt werden. Bei der Entwicklung von Generatoren ist es Best Practice, von existierendem Code auszugehen und daraus den Generator abzuleiten. Listing 1 zeigt ein Ant-Buildskript für ein einfaches Java-Projekt. Es besteht aus zwei Targets: eines für die Kompilierung des Quellcodes, das andere für das Erzeugen eines Jar-Archivs aus den kompilierten Klassen.

Das Beispiel ist bewusst einfach gehalten, um nicht von der eigentlichen Thematik abzulenken. Die im Quelltext fett hervorgehobenen Passagen stellen diejenigen Teile des Quellcodes dar, die von Projekt zu Projekt unterschiedlich sein können und somit variabel sind. Es handelt sich dabei um den Projektnamen, die Beschreibung des Projekts sowie Angaben zu den Verzeichnissen für Quellcode und kompilierte Klassen und den Namen und Pfad des Zielarchivs (Tabelle 1).

Für die Umsetzung des Beispiels in einen Generator sollen die folgenden Komponenten des Eclipse Modeling Projects verwendet werden:

Variable	Bedeutung
name	Projektname
sourceFolder	Quellcodevereichnis
binaryFolder	Binärcodeverzeichnis
fullyQualifiedJarFileName	Name der Jar-Datei

Tabelle 1: Templatevariablen

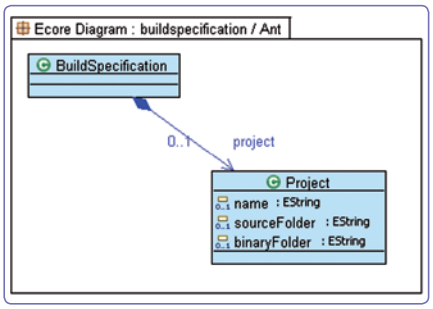

Abb. 1: Metamodell für den Buildfile-Generator

- Eclipse Modeling Framework (EMF)
- Modeling Workflow Engine (MWE)
- Model-to-Text (M2T)
- Model-to-Model (M2M)

Einige der Komponenten dürften den Lesern des *Eclipse Magazins* eher unter dem Namen openArchitectureWare geläufig sein. Die einzelnen Komponenten von openArchitcetureWare werden derzeit in das Eclipse Repository migriert und dann als Subprojekte unter dem Oberprojekt Eclipse Modeling zur Verfügung stehen. Für unsere Zwecke werden wir auf openArchitectureWare 4.2 zurückgreifen, das sowohl über die Updatesite [2] in Form von Features als auch als All-in-One- Distribution (d.h. inklusive Eclipse SDK) über die Eclipse Downloads Site [3] oder von itemis [4] bezogen werden kann.

Bilden des Metamodells

Die aus dem Originalquellcode abgeleiteten Variablen dienen nun als Grundlage für das Metamodell des Generators. Das Metamodell beschreibt Aufbau und Struktur des Modells, das vom Generator verarbeitet werden kann.

In Abbildung 1 ist das Metamodell für unseren Generator dargestellt. Wie man sehen kann, wird das Modell aus einer *BuildSpecification* bestehen, die ihrerseits maximal ein *Project* enthalten kann. Dieses *Project* enthält dann die im vorigen Schritt ermittelten Variablen als Attribute.

Das Metamodell basiert auf dem Ecore-Metametamodell und kann mit den Tools des Eclipse Modeling Frameworks erzeugt werden (die Darstellung in Abbildung 1 wurde mithilfe des Top-Cased Ecore Editors [5] erzeugt, der eine grafische Modellierung von Ecore-Modellen erlaubt):

- Zunächst wird mit File | New | Other openArchitectureWare Project ein neues, noch leeres Projekt angelegt. Dieses Projekt enthält nun die benötigten Dependencies zu den relevanten Plug-ins.
- Im nächsten Schritt erzeugen wir mit File | New | Ecore Model ein leeres Ecore-Modell und legen darin die Metamodell-Klassen *BuildSpecification* und *Project* an.

Auf der Basis dieses Metamodells können nun Modellinstanzen erzeugt werden. Dazu selektieren wir im Ecore Editor das Element *BuildSpecification*, wählen im Kontextmenü den Eintrag *Create Dynamic Instance…* und geben an, dass das Modell unter dem Namen *BuildSpecification.xmi* im Verzeichnis *src* gespeichert werden soll. Zur Bearbeitung öffnen wir das Modell mit dem *Sample Reflective Ecore Model Editor* (Abb. 2).

Transformieren

Dieses Modell soll nun mithilfe einer Modell-zu-Text (M2T, Model to Text)-

Listing 2

```
«DEFINE project FOR Project»
«FILE "build.xml"-»
<?xml version="1.0" encoding="UTF-8"?>
<project name="SimpleJavaProject"
                                  default="compile">
 <description>Build file for SimpleJavaProject
                                  </description>

 «EXPAND compileTarget FOR this»
 «EXPAND jarTarget FOR this»

</project>
«ENDFILE»
«ENDDEFINE»

«DEFINE compileTarget FOR Project-»
 <target name="compile">
  <javac srcdir="src" destdir="bin"/>
 </target>
«ENDDEFINE»

«DEFINE jarTarget FOR Project-»
 <target name="jar" depends="compile">
  <mkdir dir="dist"/>
  <jar destfile="dist/SimpleJavaProject.jar"
                                  basedir="bin">
  </jar>
 </target>
«ENDDEFINE»
```

Listing 3

```
«IMPORT buildspecification»

«EXTENSION template::GeneratorExtensions»

«DEFINE project FOR Project»
«FILE "build.xml"-»
<?xml version="1.0" encoding="UTF-8"?>
<project name="«name»" default="compile">
 <description>«description»</description>

 «EXPAND compileTarget FOR this»
 «EXPAND jarTarget FOR this»

</project>
«ENDFILE»
«ENDDEFINE»

«DEFINE compileTarget FOR Project-»
 <target name="compile">
  <javac srcdir="«this.sourceFolder»"
                  destdir="«this.binaryFolder»"/>
 </target>
«ENDDEFINE»

«DEFINE jarTarget FOR Project-»
 <target name="jar" depends="compile">
  <mkdir dir="«this.distributionDir()»"/>
  <jar destfile="«this.fullyQualifiedJarFileName()»"
                  basedir="«this.binaryFolder»">
  </jar>
 </target>
«ENDDEFINE»
```

Listing 4

```
import buildspecification;

String distributionDir(Project this):
 "dist";

String fullyQualifiedJarFileName(Project this):
 this.distributionDir() + "/" + this.name + ".jar";
```

Xpand-Schlüsselwörter

- **IMPORT:** Importiert ein Package aus einem Metamodell
- **EXTENSION:** Importiert eine Datei mit Xtend Extensions
- **DEFINE:** Definiert ein Xpand Template
- **FILE:** Leitet den Inhalt des Statementkörpers in eine Datei um
- **EXPAND:** Ruft ein Subtemplate auf
- **FOREACH:** Iteriert über eine Collection und ruft den Körper des Statements mit jedem einzelnen Element der Collection auf
- **IF/ELSEIF/ELSE/ENDIF:** Führt den Inhalt der Entscheidungszweige je nach erfüllter Bedingung aus
- **PROTECT:** Umgibt den enthaltenen Code mit einer *Protected Region*
- **LET:** Definiert eine lokale Variable
- **ERROR:** Bricht die Auswertung des Templates ab
- **REM:** Leitet einen Kommentar ein

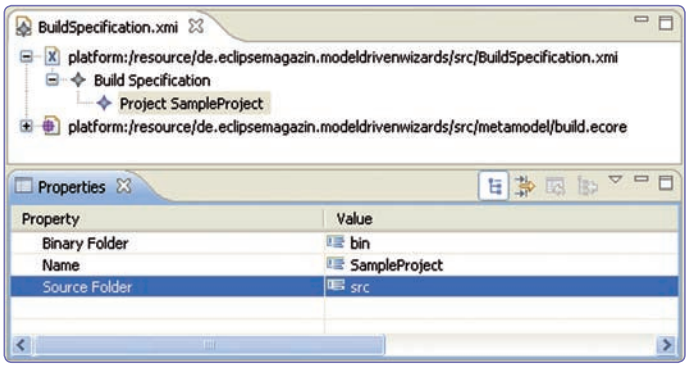

Abb. 2: Beispielmodell im Reflective Ecore Model Editor

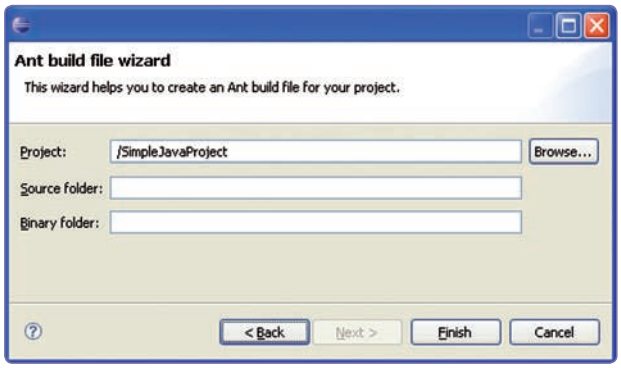

Abb. 3: Wizard zum Anlegen der Ant-Datei

Transformation in die bereits in Listing 1 dargestellte Build-Datei umgewandelt werden. openArchitectureWare bietet für M2T-Transformationen die Template-Sprache Xpand an. Xpand zeichnet sich durch einen recht kompakten Sprachumfang (Template-Blöcke, Schleifen, Entscheidungen), Polymorphie sowie die Verwendung von französischen Anführungszeichen aus. Eine kurze Beschreibung der Syntax finden Sie im Kasten „Xpand-Schlüsselwörter".

Xpand Templates werden in *.xpt*-Dateien gespeichert, also legen wir eine solche Datei über *File | New | Other | openArchitectureWare | xPand Template* neu an. Xpand erlaubt die Definition von beliebig vielen Templates pro Xpand-Datei. Soll der von einem Template erzeugte Text in eine Datei ausgegeben werden, so ist der entsprechende Block mit einem *File*-Statement zu umgeben. Templates können weitere Templates aufrufen, was einerseits eine bessere Strukturierung der Template-Dateien ermöglicht und andererseits später die non-invasive Anpassung von Templates mit aspektorientierten Mechanismen ermöglicht. Der Aufruf von Subtemplates erfolgt mit dem Schlüsselwort *Expand*. Die Build-Datei aus Listing 1 kann somit in drei Templates zerlegt werden: das erste Template (*project*) kapselt die Erzeugung der Build-Datei an sich und erzeugt den Rumpf des Build-Skripts. Die Templates *compileTarget* und *jarTarget* kapseln den Code für das jeweilige Ant-Target. Die Struktur der Templates ist Listing 2 zu entnehmen.

Nachdem wir die Struktur der Templates definiert haben, sollen nun die variablen Teile der Templates durch Variablen ersetzt werden. Die möglichen Variablen werden durch das Metamodell vorgegeben, das nun importiert werden muss, damit es sowohl bei der Template Engine als auch beim Editor registriert wird. Dies geschieht mit dem Schlüsselwort *Import* (Listing 3).

Xpand Templates haben einen Kontext, innerhalb dessen das Template Zugriff auf Elemente des Modells hat. Durch die Anweisung *Define project for Project* erhält das Template Zugriff auf eine Instanz des Metamodellelements *Project* und kann auf alle Features (d.h. Attribute und Methoden) dieses Modellelements zugreifen. Innerhalb des *File*-Blocks wird z.B. der Name des Projekts aus dem Attribut *name* ausgelesen. Auch die weiteren Attribute (*sourceFolder*, *binaryFolder*) werden analog ausgelesen. Besondere Erwähnung verdienen die Aufrufe *this.distributionDir()* und *this.fullyQualifiedJarFileName()*. Hierbei handelt es sich um Aufrufe so genannter Extensions. Extensions sind Operationen, die in der oAW-Sprache Xtend definiert werden und mit denen das Metamodell dynamisch erweitert werden kann. Dies ist z.B. dann sinnvoll, wenn bestimmte Codefragmente immer wieder verwendet werden (wie z.B. zusammengesetzte Texte). Die Ausführung von Modelltransformationen wird in openArchitectureWare über eine Workflow Engine gesteuert. Die einzel-

Listing 5

```
<?xml version="1.0"?>
<workflow>
 <property name="basedir" value="." />
 <property name="model"
 value="de.eclipsemagazin.modeldrivenwizards/src/
                                BuildSpecification.xmi" />

 <!-- set up EMF for standalone execution -->
 <bean class="org.eclipse.mwe.emf.StandaloneSetup" >
  <platformUri value=".."/>
 </bean>

 <!-- load model and store it in slot 'model' -->
 <component class="org.eclipse.mwe.emf.Reader">
  <uri value="platform:/resource/${model}" />
  <modelSlot value="model" />
 </component>

 <!-- generate code -->
 <component class="org.openarchitectureware.xpand2.
                                Generator">
  <metaModel id="mm"
   class="org.eclipse.m2t.type.emf.
                                EmfRegistryMetaModel"/>
  <expand
   value="template::BuildTemplate::project FOR model.
                                project" />
  <outlet path="${basedir}/"/>
 </component>
</workflow>
```

Name	Art	Attributname	Typ	Lowerbound	Upperbound	Containment
BuildSpecification	Referenz	project	Project	0	1	ja
Project	Attribut	name	EString			
	Attribut	sourceFolder	EString			
	Attribut	binaryFolder	EString			

Tabelle 2: Metamodell

nen Verarbeitungsschritte werden von Workflowkomponenten durchgeführt. Der Workflow und die einzelnen Komponenten werden über eine XML-Datei konfiguriert. Listing 5 zeigt den Workflow für unser Beispiel. Nachdem einige Eigenschaften gesetzt wurden (z.B. der Pfad zu unserem Modell) und EMF konfiguriert wurde, wird das Modell eingelesen und die Xpand Template Engine gestartet. Nun können wir den Workflow starten, um das Template zu testen. Das Starten des Workflows erfolgt, indem man aus dem Kontextmenü der Workflowdatei den Eintrag Run as | oAW Workflow auswählt. Nach Ausführung des Workflows befindet sich die Zieldatei *build.xml* im Wurzelverzeichnis des Projekts.

Einbindung in Eclipse

Nachdem der Workflow nun also standalone einsetzbar ist, muss er in Eclipse integriert werden, was in Form eines Wizards geschehen soll. Der Wizard muss Eingabefelder für das Projekt, den Quell- und den Zielordner enthalten (Abb. 3). Der Quellcode für das GUI des Wizards sowie die Einbindung in der Datei *plugin.xml* finden Sie auf der Heft-CD.

Modell programmgesteuert erstellen

Damit die vom Benutzer in das Wizard GUI eingegebenen Werte im Generator ausgelesen werden können, müssen sie in ein Modell geschrieben werden. Bisher haben wir diesen Schritt manuell vorgenommen, indem wir mithilfe des Sample Ecore Model Editors eine dynamische Instanz des Metamodells angelegt haben. Diesen Schritt müssen wir nun automatisieren, d.h. das Modell muss programmatisch erzeugt werden.

Hier hilft EMF weiter: es bietet eine Möglichkeit, aus dem Metamodell Klassen zu erzeugen, mit denen das Modell manipuliert werden kann – so genannte Modell-Klassen. Zu diesem Zweck müssen wir zunächst ein EMF-Generatormodell erzeugen. Dazu markieren wir die Datei *build.ecore*, die unser Metamodell enthält, und wählen dann aus dem Hauptmenü *File | New | Other... | Eclipse Modeling Framework - EMF Model*. Nach einem Klick auf *Next* schlägt der Wizard als Namen für das Generatormodell *build.genmodel* vor. Diesen Vorschlag übernehmen wir durch einen Klick auf *Next*, wählen auf der nachfolgenden Seite *Ecore model* und klicken erneut auf *Next*. Auf der nun folgenden Seite müssen wir unser Ecore-Metamodell auswählen: *platform:/resource/de.eclipsemagazin.modeldrivenwizards/src/metamodel/build.ecore*. Auf der letzten Seite bestätigen wir das Wurzel-Package (*buildspecification*) und schließen den Wizard über einen Klick auf *Finish* ab. Bevor wir aus diesem Generatormodell die Klassen für den Zugriff auf das Modell erzeugen können, müssen wir noch das Base Package festlegen. Hierzu markieren wir das Element (*Buildspecification*) und setzen im *Properties View* die Eigenschaft *Base Package* auf den Wert *de.eclipsemagazin.modeldrivenwizards*. Nun rufen wir aus dem Kontextmenü des Elements *Buildspecification* den Eintrag *Generate Model Code* auf. EMF erzeugt daraufhin im Projekt im Package *de.eclipsemagazin.modeldrivenwizards.buildspecification* die Klassen für die Manipulation des Modells. Diese Klassen können nun genutzt werden, um das Modell zu erzeugen.

Listing 6

```
private void invokeGenerator(IProject project, String
                        srcDirName, String binDirName,
                              IProgressMonitor monitor)
 throws CoreException {
Map<String, String> properties = new
                              HashMap<String, String>();
Map<String, Object> slotMap = new HashMap<String,
                                          Object>();

// configure properties passed to the workflow engine
properties.put("basedir", project.getLocation().
                                              toFile()
 .getAbsolutePath());

BuildSpecification buildSpecification =
                                    createModel(project,
  srcDirName, binDirName);

slotMap.put("model", buildSpecification);

WorkflowRunner runner = new WorkflowRunner();
boolean success = runner.run("workflow/build.oaw",
 new NullProgressMonitor(), properties, slotMap);

project.refreshLocal(IResource.DEPTH_INFINITE,
                                              monitor);
}
```

Wizard ruft Workflow

Nun kann der Workflow aufgerufen werden. Zu diesem Zweck stellt openArchitectureWare die Klasse *WorkflowRunner* bereit. Beim Aufruf der Methode *run()* wird der relative Pfad sowie das Modell (im Slot *model* – so wie wir es im Workflow angegeben haben) zur Workflow-Datei übergeben. Weiterhin übergeben wir das Wurzelverzeichnis des gewählten Projekts, sodass die erzeugte Build-Datei auch im richtigen Verzeichnis landet (Listing 6). Nach dem Generatorlauf wird der Zustand des Projekts durch den Aufruf der *refreshLocal()*-Methode aktualisiert, sodass die neu erzeugte Datei im Resource Explorer erscheint. Das Ergebnis unserer Arbeit können wir bewundern, indem wir das Plug-in exportieren oder es in einer Runtime-Instanz von Eclipse starten: Dazu markieren wir das Projekt *de.eclipsemagazin.modeldrivenwizards* und wählen aus dem Kontextmenü *Run As | Eclipse Application*. In der Runtime-Instanz ist der Wizard dann über *File | New | Model Driven Wizards | Create Ant Build File* erreichbar.

Fazit und Ausblick

Die Ausführungen in diesem Artikel zeigen, wie die Tools und Technologien aus dem Eclipse Modeling Project genutzt werden können, um die Entwicklung von textbasierten Generatoren zu vereinfachen und in Eclipse einzubinden. Der Code des Generators selbst wird überschaubarer und damit einfacher zu warten. Die Verwendung von Xpand als ausgereifter Template-Sprache erleichtert die Entwicklung der benötigten Templates um ein Vielfaches. In der nächsten Ausgabe des Eclipse Magazins zeigen wir, wie die hier gezeigte Vorgehensweise genutzt werden kann, um Codegenerierung on-the-fly durchzuführen. Als Beispiel wird dann eine integrierte Entwicklungsumgebung für LEGO Mindstorms Roboter dienen.

Peter Friese ist Senior-Softwarearchitekt bei itemis. Er ist spezialisiert auf modellgetriebene Softwareentwicklung, Spring sowie Eclipse-Technologien. Peter ist Committer für FindBugs und openArchitectureWare. Kontakt: peter.friese@itemis.de.

>> Links & Literatur

[1] ant.apache.org

[2] www.openarchitectureware.org/updatesite/milestone/site.xml

[3] www.eclipse.org/downloads/distros.php

[4] http://oaw.itemis.de

[5] topcased-mm.gforge.enseeiht.fr/release/update-site3.3