UNIKASSEL VERSITÄT

Research Group Software Engineering

Software Engineering II

Codegenerierung für den SmartIO Editor mit der Modeling Workflow Engine

Wintersemester 10/111

Fachgebiet Software Engineering

Albert Zündorf / Nina Geiger

Wiederholung

- **Bisher im Laufe des Semesters umgesetzt:**
  - Modellierung eines Meta-Modells für die Hausautomatisierung mit Fujaba
  - Generierung von EMF Quelltext aus dem Fujaba Klassendiagramm
  - Generieren eines Editor und eines Edit Eclipse Plugins für das EMF Modell (baumartiger Editor)
  - Anfügen von Annotation an des Klassendiagramm und automatische Generierung des grafischen Editors aus diesen Annotationen mit Hilfe von Eugenia
  - Implementierung von Refactoring Operationen mittels Fujaba Storydiagrammen.
    - Modellieren der Operationen
    - Generieren von Quelltext
    - Einbinden der Operationen als PopupActions in den grafischen Editor
  - Implementierung einer einfachen Interpreterlogik für den grafischen Editor
  - Custom Figures zur weiteren Verfeinerung der grafischen Darstellung des Editors

Nächstes Ziel

- **Per Rechtsklick auf den Hintergrund des Diagramms Java Quelltext aus dem Diagramm generieren**
  - Verwendung der MWE Engine zur Codegenerierung
  - Verfassen von Templates und Workflow files
  - Implementierung einer Action in Fujaba, die die Codegenerierung aus dem grafischen Editor heraus aufruft.

Installation der MWE Plugins ins Eclipse

- **Von der Helios Update Site installieren (falls noch nicht in eurem Eclipse vorhanden):**
  - MWE 2 language SDK
  - MWE 2 runtime SDK
  - MWE SDK
  - Xpand SDK

SmartIO Projekt auf die Codegenerierung vorbereiten

- **Herunterladen von CodeGenClasses.ctr aus dem Blog**
  - Enthält notwendige Klassen aus dem MWE Framework als Referenz
- **Hinzufügen der CodeGenClasses.ctr in euer Eclipse Projekt**
- **CodeGenClasses.ctr als Abhängigkeit in eurer SmartIO.ctr angeben**
  - Genau wie bei **`JavaClasses`** und **`EclipseClasses`**
- **In die Manifest.MF zu den Plugin Dependencies hinzufügen:**
  - **`org.eclipse.emf.mwe.core`**

Action für die Codegenerierung anlegen

- **Im Klassendiagramm in euerem Action package eine neue Klasse für die Codegenerierung anlegen**
  - Erbt von **`TransactionActionDelegate`**
  - **`runImpl`** implementieren
- **Action soll auf dem kompletten Diagramm aufgerufen werden, wird also als Contribution für das Board angelegt.**

Implementierungsdetails runImpl

1. Board aus der `IStructuredSelection` holen (wie gehabt)
2. StatementActivity für das Setzen notwendiger MWE Parameter:

```
//define parameters for MWE
String resourceUri = #nameDesBoardObjektes.eResource().getURI().toPlatformString(false);
String pathUri = resourceUri.subSequence(0, resourceUri.lastIndexOf('/')).toString();

String fileString = ResourcesPlugin.getWorkspace().getRoot().getLocation().toString() +
pathUri;

System.out.println("Zielpfad: " + fileString);

Map<String,String> properties = new HashMap<String,String>();
Map slotContents = new HashMap();
slotContents.put("model", #nameDesBoardObjektes);
properties.put("srcGenPath", fileString);

final String WORKFLOW_FILE = „#nameDesWorkflowFiles";
```

Implementierungsdetails runImpl

3. Neues `NullProgressMonitor` Objekt anlegen

- Klasse kommt aus `CodeGenClasses.ctr`

4. Neues `WorkflowRunner` Objekt anlegen

- Klasse kommt aus `CodeGenClasses.ctr`

5. Auf dem `WorkflowRunner` mittels CollaborationStatement

```
run(WORKFLOW_FILE, monitor, properties, slotContents)
```

aufrufen

Optional Folders refreshen

```
//// refresh folder
IResource tmpFile = ResourcesPlugin.getWorkspace().getRoot().findMember(pathUri);
if (tmpFile == null)
     return;
try {
    tmpFile.refreshLocal(IResource.DEPTH_INFINITE, null);
} catch (CoreException e) {
     e.printStackTrace();
}
```

Genaues Beispiel meiner `GenerateCodeAction` im Blog als PDF downloadbar!!!

Edit Imports

- **Klassen aus den Statement Activities müssen per Hand in die `GenerateCodeAction` importiert werden**
  - Klasse im Klassendiagramm selektieren
  - Rechtsklick -> Edit Imports
    - `java.util.Map`
    - `java.util.HashMap`
    - `org.eclipse.core.resources.ResourcesPlugin`
    - `org.eclipse.core.resources.IResource`
    - `org.eclipse.core.runtime.CoreException`
  - Falls auf linker Seite nicht bereits vorhanden über new -> Auswahl Class hinzufügen und auf die rechte Seite des Dialogs übernehmen.
  - Projekt speichern.

Workflow file anlegen

- **In den src Ordner eures Projektes eine Workflow Datei anlegen:**
  - Rechtsklick -> New -> Other -> Modeling Workflow Engine -> Workflow File
  - Gleichen Namen verwenden wie in der Statement Activity eurer `GenerateCodeAction`, sonst wird die Datei nicht gefunden.

Inhalt der Workflow Datei

```
<?xml version="1.0"?>
<workflow>
    <component id="generator" class="org.eclipse.xpand2.Generator" skipOnErrors="true">
        <fileEncoding value="ISO-8859-1"/>
        <metaModel id="mm" class="org.eclipse.xtend.typesystem.emf.EmfMetaModel">
            <metaModelPackage value="#vollqualifizierter Name eurer Package Klasse"/>
        </metaModel>

        <outlet path="${srcGenPath}">
            <postprocessor class="org.eclipse.xpand2.output.JavaBeautifier"/>
        </outlet>

        <!--protected regions configuration -->
        <prSrcPaths value="${srcGenPath}"/>
        <prDefaultExcludes value="false"/>

        <expand value="SmartIOMain::main FOR model"/>
    </component>
</workflow>
```

Wert wird in Statement Activity gesetzt

Name der Template Datei ist SmartIOMain, Name des Templates darin ist main

Template Datei anlegen

- **Im src Ordner eures Projektes: New -> Other -> Xpand -> Xpand Template**
  - Name muss dem im Workflow file angegebenen entsprechen
- **Importieren eures Meta Modells ins Template**
- `«IMPORT de::unikassel::se::smartio::model»`
- **Zugriff auf die Elemente des Meta Modells über Code Completion möglich**
  - `«DEFINE main FOR model::Board»`

- **Genaue Syntax und Verwendung der Xpand language unter:**

  http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/core_reference.html#xpand_reference_introduction

Aus Template zu generierender Code

- **Eine Java Klasse pro Board**
- **`main` Methode in Java Klasse**
- **`init()` Methode die aus `main` aufgerufen wird und die Struktur des Boardes, wie in Diagramm spezifiziert anlegt**
  - Alle Objekte anlegen (EMF Objekte, nicht mit new, sondern über die Factory anlegen)
  - Alle Links ziehen
  - Alle Attribute setzen
- **Eine `public void toggleSwitch(Switch switch)`**
  - Kann über eDobs aufgerufen werden. Toggelt einen Switch on/off und liefert die entsprechenden Nachrichten aus. (Nachrichten müssen natürlich erst erstellt werden)
- **Eine `public void deliverMessage(Channel channel)`**
  - Kann über eDobs aufgerufen werden. Liefert die im Channel befindlichen Nachrichten an alle angrenzenden Komponenten aus.

Fertigstellung

- Action Code wie gewohnt aus Fujaba generieren
- Action in der Manifest.MF eintragen
- Plugin starten und testen

- Code wird im Runtime Eclipse in das gleiche Projekt generiert, in welchem auch die Diagrammdatei liegt.

TIPP: Beim Codegenerieren ist es immer sinnvoll, sich den Quelltext, der herauskommen soll einmal per Hand zu schreiben und anschließend nur die dynamischen Teile im Template auszutauschen