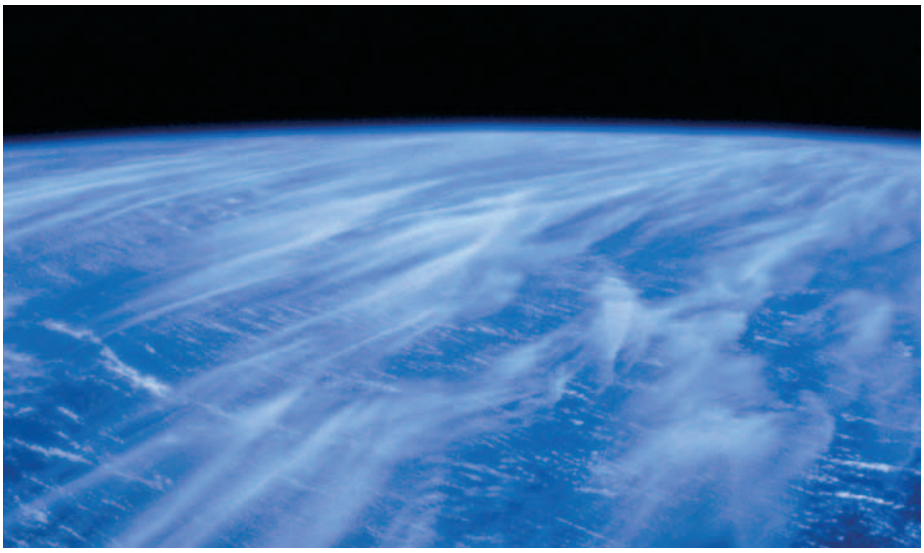
Agile MDA mit EMF: dargestellt am Beispiel einer PHP-Website

von Jens von Pilgrim

Mr. M

Kerngedanke von MDA ist die automatische Generierung von Code auf Basis von Modellen. Das Eclipse Modeling Framework (EMF) stellt die hierfür benötigen Grunddienste bereit: ein Metamodell zur Beschreibung des Modells sowie einen über Templates gesteuerten Generator zur Erzeugung des Codes und einfacher Editoren. Dass die Anwendung einfach und effektiv sein kann, zeigt der Artikel an einem Beispiel.



Dieser Artikel stellt eine kleine MDA-Anwendung mit dem EMF vor. Als Beispiel dient ein Editor für Use Cases, mit dem prototypisch PHP-Seiten generiert werden können, die die Use Cases in Form navigierbarer Webseiten abbilden. Nach einer kurzen Diskussion der benötigten Begrifflichkeiten und des grundsätzlichen Vorgehens beim Einsatz von MDA werden ein eigenes Modell erzeugt und die besonderen Eigenschaften von EMF-basierten Modellen vorgestellt. Abschließend nutzen wir die Code-Generierungsfähigkeiten von EMF, um aus dem Modell PHP-Seiten zu generieren. Für MDA-Kenner und Freunde von Akronymen definieren wir also zunächst ein PIM, um dann daraus ein PSM abzuleiten – und nebenbei erklären wir auch die Abkürzungen.

Um das Beispiel nachvollziehen zu können bzw. um eigene EMF-Anwendungen zu entwickeln, muss EMF als Eclipse-Plugin installiert sein. EMF ist ein Eclipse Tool Project, die benötigten Files sowie weitere Dokumentation finden sich auf der EMF Homepage [1]. Für diesen Artikel wurden Eclipse 3.0.1 und EMF 2.0.1 verwendet.

Modelle und Metamodelle

Zunächst zur Klärung einiger Begriffe: Das „M“ der Abkürzung EMF steht für Modeling. Frankel beschreibt im Glossar von [2] Modell als „an abstraction of a system“. Diese Erklärung ist sehr allgemein gehalten und führt zu neuen Fragen, etwa ob nicht ein „system“ selbst bereits eine Abstraktion ist. Im Zusammenhang mit UML kann der Begriff Modell pragmatischer mittels Diagrammen erklärt werden: Diagramme sind grafische Repräsentationen von Modellen. Ein Klassendiagramm zeigt etwa Klassen und deren Beziehungen untereinander, wie Vererbung oder Assoziationen. Eine andere agilere Repräsentationsform wäre die simple Abbildung des Modells als Programmcode. EMF nimmt diesen Gedanken auf, hier lassen sich Modelle in Form von Java-Interfaces beschreiben.

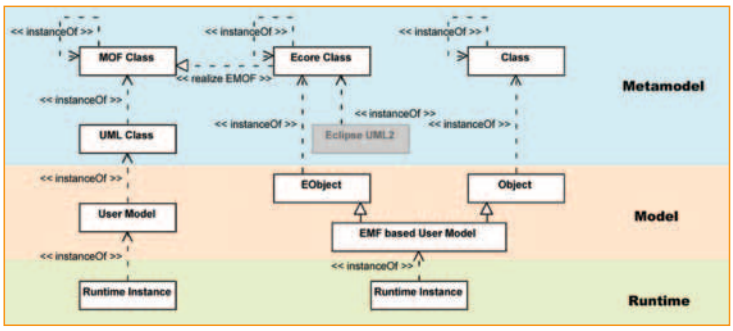
Eine andere bekannte Verwendung des Begriffs Modells kommt aus dem Bereich der Entwurfsmuster: das Modell im MVC Pattern. Die wichtigste Eigenschaft des Modells im MVC Pattern ist, dass es das Observer Pattern unterstützt. Somit werden Controller und View sehr lose an das Modell gekoppelt bzw. das Modell hat keine strukturellen Abhängigkeiten zu diesen beiden Komponenten. EMF nimmt diesen

Aspekt des Modells ebenfalls auf; mit dem Framework generierte Modelle unterstützen das Observer Pattern. EMF geht sogar noch einen Schritt weiter. Aus dem Modell kann ein entsprechender Editor generiert werden. Da das EMF innerhalb des Eclipse-Frameworks angesiedelt ist, wird der Editor als Eclipse-Plug-in realisiert. Der Use-Case-Editor unseres Beispiels ist ein auf diese Weise generierter Editor.

Um Modelle zu verarbeiten bzw. etwa daraus Code zu generieren, müssen diese semantisch eindeutig beschrieben werden – sonst ist nicht klar, was eigentlich generiert werden soll. Um ein Modell entsprechend genau zu definieren, werden Metamodelle verwendet. Die Metamodelle sind schlicht und einfach Modelle von Modellen. Da UML zur Erstellung eigener Modelle eingesetzt wird, ist die Spezifikation von UML die Definition eines Metamodells. Das Problem der Eindeutigkeit wurde damit allerdings nur verschoben, vom Modell hin zum Metamodell. Tatsächlich ist auch das Modell von UML, also das Modell vom Metamodell, sprich Metametamodell, eindeutig definiert und hört im Fall von UML auf den Namen MOF – Meta Object Facilites. Stark vereinfacht definiert MOF eine Art Klassendiagramm (das übrigens direkt von UML importiert wird) und ergänzt dieses um weitere Fähigkeiten. Die in diesem Rahmen vielleicht wichtigste Erweiterung des einfachen Klassendiagramms ist ein Reflection-Mechanismus, der Java-Programmierern vom Java Reflection API her bekannt vorkommen dürfte. Ein Subset der MOF, das Essential MOF (EMOF), stellt alle Funktionen bereit, die etwa benötigt werden, um MOF-Modelle zu serialisieren, etwa in Form von XMI. EMF enthält mit Ecore ein eigenes Metamodell. Ecore stellt Metaklassen für Klassen und ihre Abhängigkeiten voneinander bereit, also die Elemente des Klassendiagramms. So gesehen stellt Ecore einen Ausschnitt des UML-Modells bereit. Andererseits wird genau dieser Ausschnitt von MOF aus der UML importiert. Also ist Ecore eine Implementierung des EMOF – bis auf teilweise andere Benennungen. Abbildung 1 zeigt die Metalevel von MDA, EMF und Java im Vergleich.

Eigene Modelle übernehmen wichtige Eigenschaften von ihrem Metamodell, hier

Abb. 1: Metalevel von MDA, EMF und Java im Vergleich



Ecore, wie die Serialisierbarkeit in Form von XMI und den Ecore-eigenen Reflection-Mechanismus. „Normale" Java-Klassen sind immer von *java.lang.Object* abgeleitet. Alle von *java.lang.Object* abgeleiteten Klassen haben über *getClass()* Zugriff auf ihre Klasse. Diese stellt Metainformationen über die Klasse bereit, etwa den Namen (*getName()*), die Methoden der Klasse (*getMethods()*) oder Felder (*getFields()*). Ecore-basierte Klassen sind natürlich auch Java-Klassen und damit von *java.lang.Object* abgeleitet. Zusätzlich sind hier aber alle Modellklassen auch von *org.eclipse.emf.ecore.EObject* abgeleitet. Über *eClass()* haben diese Klassen Zugriff auf die Ecore-Klasse *org.eclipse.emf.ecore.EClass*, das Gegenstück zu *java.lang.Class*, mit Zugriff auf die Metainformationen, die Ecore bzw. EMOF bereitstellen. Alle mit Ecore zusammenhängenden Klassen beginnen mit einem großen E, alle Funktionen mit einem kleinen e.

Welche der Metainformationen bietet Ecore? Wie bei Java können die Methoden und Felder einer Klasse abgefragt werden – hier mit den UML üblichen Benennungen, also Operationen und Attribute. Eine weitere, höchst entscheidende Metainformation stellen References dar. Im Gegensatz zum Metamodell von Java können in Ecore nicht nur einfache Attribute, sondern Referenzen definiert werden, also die aus den Klassendiagrammen bekannten m:n-Assoziationen zwischen Klassen bzw. Interfaces.

Diese werden in Java im Allgemeinen über Collections nachgebildet, mit dem Nebeneffekt, dass die Information über den Typ der Elemente verloren geht – erst mit J2SE 5 ist dies über den Einsatz von Templates zu kompensieren. Abbildung 2 zeigt einen Ausschnitt der Ecore-Klassen aus dem Paket *org.eclipse.emf.ecore*.

Definition des Modells

Zurück zum Beispiel: Am Anfang steht das Modell des Editors, hier ein Modell für Use Cases. Um das Beispiel einfach zu halten, werden Use Cases in reduzierter Form definiert. Akteure und Use Cases können zueinander bidirektional in Beziehung gesetzt werden. Die bekannten *include*- und *extend*-Beziehungen zwischen Use Cases werden als einfache Assoziationen abgebildet und Akteure wie Use Cases werden im Rahmen eines Systems angelegt. Letzteres wird, um Namenskonflikte mit *java.lang.System* zu vermeiden, mit *PortalSystem* bezeichnet. Ähnlich wie die „echten" Use Cases der UML-Spezifikation kann unser Modell hier in einem Klassendiagramm beschrieben werden, wobei die abgebildeten Klassen keine UML-Klassen, sondern Ecore- bzw. MOF-Klassen sind.

Wie bereits oben erwähnt, ist ein Diagramm nur eine mögliche Darstellungsform des Modells. Eine andere Möglichkeit ist die Erstellung von Java-Interfaces. Um mittels EMF später Code zu generieren, muss ein Modell als Ecore-Modell vorliegen, d.h., die definierten Klassen müssen als Instanzen von *org.eclipse.emf.ecore.EClass* verfügbar sein. Das EMF-Plug-in liefert einen eigenen baumorientierten Editor mit, um direkt Ecore-Modelle zu erstellen. Dieser Editor ist über File | New | Other | Example EMF Creation Wizards | Ecore Model erreichbar. Alternativ kann EMF das Modell aus existierenden Dateien importieren, etwa aus Rational-Rose-Modellen (*.mdl*-Dateien).

Wir definieren das Modell über Java-Interfaces. Dabei müssen allerdings die Einschränkungen von Java hinsichtlich der Assoziationen berücksichtigt bzw. kompensiert werden. Dazu werden die Interfaces mit entsprechenden Kommentaren im

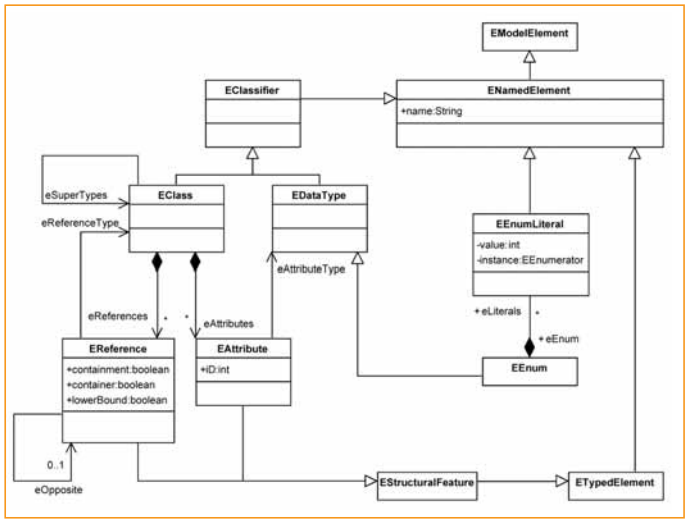


Abb. 2: Ecore-Ausschnitt

Javadoc-Stil erweitert. EMF erkennt innerhalb der Kommentare das Tag *@model* und ergänzt die Java-Deklaration um die als Attribute von *@model* definierten Metainformationen. Alle Interfaces, Attribute und Methoden, die ins Modell übertragen werden sollen, müssen entsprechend gekennzeichnet werden. Die Tabelle listet die wichtigsten Attribute auf. Die Generierung des EMF-Modells auf Basis der Interfaces, die EMF übrigens einfach im Source-Verzeichnis des entsprechenden Projekts findet, geschieht über den mitgelieferten Wizard. Listing 1 zeigt die hier verwendeten Interfaces. Um Schreibarbeit zu sparen, werden die getter definiert. EMF generiert die setter automatisch, sofern das Attribute *unsettable* nicht gesetzt ist.

Für unser Beispiel legen wir ein Projekt *portal.model* an und speichern die Schnittstellen im Source-Verzeichnis *src* im Paket *portal*. Das Modell legen wir in einem eigenen Ordner *model* ab. Dazu wählen wir im Wizard-Menü den entsprechenden Wizard aus (File | New Eclipse Modeling Framework | EMF Models) und speichern das Modell unter *model/portal.genmodel*. Der Wizard liest die Schnittstellen ein und generiert dann für das ausgewählte Paket das Modell.

Genauer gesagt werden sogar zwei Modelle erzeugt: ein Ecore- und ein Genmodel-Modell. Ersteres ist das eigentliche Modell mit den definierten Klassen, Letzteres bezieht sich auf Ersteres und erweitert es um spezifische Informationen zur Codegenerierung, wie etwa die Spezifizierung von Datei- und Pfadangaben. Das Modell kann nun in Eclipse mit dem bereits erwähnten baumorientierten Editor geöffnet werden.

@model-Attribut	Wert	Beschreibung
type	Klassenname	Typ der Elemente bei Referenzen, wird bei einfachen Attributen aus dem Rückgabewert ermittelt
containment	true | false (default)	Art der Assoziation, bestimmt u.a., wie die Kinder-Elemente serialisiert werden
lower, upper	{-1,0, ..}	Untere bzw. obere Grenze der Multiplizität einer Assoziation, -1 bedeutet unbeschränkt
opposite	Referenzname	Entgegengesetzte Referenz bei bidirektionalen Beziehungen
changeable	true (default) | false	Attribut bzw. Referenz darf geändert werden, bestimmt u.a., ob ein Setter generiert wird
id	true | false (default)	Attribut kann als ID verwendet werden, setzt ID-Attribut des Attributes
transient, volantile	true | false (default)	Deklariert Attribut oder Referenz entsprechend
abstract, interface	true | false (default)	Deklariert Klasse als abstrakt bzw. Interface

Tabelle 1: Annotationen für Java-Interfaces (Auswahl)

Nun können wir unser Modell bzw. eine Java-Implementierung des Modells generieren. Der Editor des Genmodel-Modells fügt im Eclipse-Menü den passenden Eintrag Generator hinzu, alternativ sind die entsprechenden Befehle über die Kontextmenüs der Einträge im Editor zu erreichen. Zuvor ändern wir ein paar Voreinstellungen im Genmodel. EMF bietet mehrere Möglichkeiten der Persistierung. EMF selbst unterstützt die Serialisierung in XMI- oder XML-Format. CDO (www.sympedia.org/cdo/), ein optionales EMF-Plug-in, bietet die Möglichkeit, EMF-Modelle in relationalen Datenbanken zu speichern. Für das Beispiel verwenden wir XML-Serialisierung, entsprechend ist dies im Property-Fenster unter Resource Type einzustellen. Um die originalen Schnittstellen der Modelldefinition nicht zu verlieren, soll das generierte Modell in einem anderen Projekt gespeichert werden. Daher ändern wir in den Eigenschaften alle Projektnamen, indem die *.model*-Endung entfernt wird.

Über Generator | Generate All starten wie die Generierung. Dabei werden gleich drei neue Projekte generiert: Das eigentliche Modell (unter *portal*), ein Edit- (*portal.edit*) und ein Editor-Plug-in (*portal.editor*). Das Edit-Plug-in definiert die in der Eclipse-Umgebung wichtigen Provider (*PropertySourceProvider* u.a.), das Editor-Plug-in verwendet diese und stellt einen baumbasierten Editor bereit. Eine hier nicht weiter verfolgte Möglichkeit ist die Generierung eines passenden XML-Schemas, die über das gleiche Menü ausgewählt werden kann.

Das Modell-API

Das von EMF generierte API des Modells der Use Cases wurde unter *portal* im aktuellen Workspace abgelegt. Neben den bereits erwähnten Interfaces wurden ein Paket mit Implementierungen sowie ein Utility-Paket erstellt.

Die eigentlichen Modellelemente sind als Interfaces definiert worden – sie unterscheiden sich bis auf die generierten setter und zusätzliche Kommentare wenig von den ursprünglichen Interfaces. Da es sich bei dem generierten Modell ja um ein Ecore-Modell handelt, sind alle Klassen bzw. Interfaces nun von *EObject* abgeleitet, so kann u.a. die Ecore Reflection verwendet

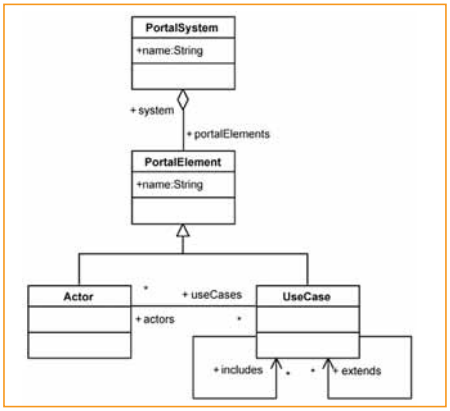


Abb. 3: Klassendiagramm des Use-Case-Modells

werden. Im Paket *portal.impl* sind die passenden Implementierungen hinterlegt. Zum Erzeugen eigener Instanzen der Modellklassen verwendet das generierte API das „abstract factory" Pattern, die dafür nötige Factory (*portal.PoralFactory*) wurde mitgeneriert. Auch sie ist als Interface deklariert, die Implementierung ist über *PortalFactory.eINSTANCE* verfügbar. Im Paket *portal.util* liegt außerdem die Implementierung der Ressource für das Modell. Die Ressource ist Teil des EMF Persistence API und dient als Persistenz-Container für die Modellobjekte. Da vor der Generierung der Ressource-Typ XML ausgewählt wurde, ist die Ressourceimplementierung von *XMLResourceImpl* abgeleitet. Weitere Details zur Ressourceimplementierung sind in [3] nachzulesen.

Abbildung 5 zeigt die Zusammenhänge des Modell-API und Ecore sowie die verwendeten Patterns. Das bereits erwähnte Observer Pattern wird über die Schnittstellen Notifier und Adapter realisiert. Bei Änderungen schicken die Notifier (Subject/Subjekt) Notification-Objekte an angemeldete Adapter (Observer/Beobachter). Die generierten Modellklassen sind indirekt über *EObject* von Notifier abgeleitet. Als Besonderheit kennen die Adapter hier das von ihnen beobachtete Objekt (Target). D.h., in Erweiterung des Observer Pattern ist hier die Assoziation zwischen Subjekt und Beobachter bidirektional. Daher kann der Observer über seine passive Rolle als Beobachter hinaus aktiv das beobachtete Subjekt steuern. Dies lässt sich zur Umsetzung von Adaptern einsetzen, d.h., das Verhalten bzw. die Schnittstelle des Subjekts werden so, ohne den Einsatz von Spezialisierung, an spezielle Bedürfnisse angepasst – daher der Name Adpater der Beobachterschnittstelle. Eine ausführliche Beschreibung dieses Musters, seiner Anwendung und Besonderheiten, etwa der Einsatz von so genannten AdapterFactories, ist ebenfalls in [3] nachzulesen. Die Klasse *PortalPackage* enthält IDs für alle Elemente, also Klassen, Methoden und Attribute, des Modells, die u.a. in der Auswertung der Notifications in eigenen Klassen (Controllern o.Ä.) verwendet werden können. Sie ist, wie die Factory, als Singleton implementiert, die Instanz kann mittels *Portal-Package.eINSTANCE* erreicht werden. Die wichtigsten Eigenschaften des generierten API sind also

Abb. 4: EMF Model Wizard

- Serialisierung mittels Ressource durch die EMF-Persistenz-API in XML oder XMI
- Unterstützung des Observer Pattern (etwa für MVC) durch Implementierung der Schnittstelle Notifier
- Sicherstellung der Modellkonsistenz, etwa über die automatische Berücksichtigung bidirektionaler Assoziationen

Der generierte Editor

Die ebenfalls im Workspace generierten Projekte *portal.edit* und *portal.editor* bil-

Listing 1

```
...
/**
 * @model abstract="true"
 */
public interface PortalElement {
  /**
   * @model
   */
  String getName();
  /**
   * @model opposite="portalElements" lower="1"
                                                upper="1"
   */
  PortalSystem getSystem();
}

...
/**
 * @model
 */
public interface Actor extends PortalElement {
  /**
   * @model type="UseCase" opposite="actors" lower="0"
                                                upper="-1"
   */
  List getUseCases();
}

...
/**
 * @model
 */
public interface UseCase extends PortalElement {
  /**
   * @model type="Actor" opposite="useCases" lower="0"
                                                upper="-1"
   */
  List getActors();
  /**
   * @model type="UseCase" lower="0" upper="-1"
   */
  List getExtends();
  /**
   * @model type="UseCase" lower="0" upper="-1"
   */
  List getIncludes();
}

...
/**
 * @model
 */
public interface PortalSystem extends EObject{
  /**
   * @model
   */
  String getName();
  /**
   * @model containment="true" opposite="system" type=
                                                "PortalElement"
   */
  List getPortalElements();
},
```

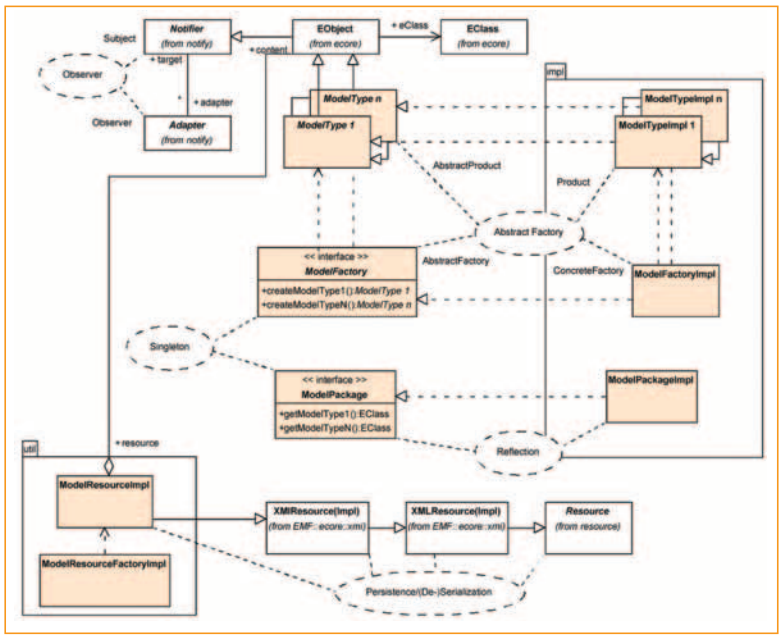


Abb. 5 Klassendiagramm des Modell-API

den zusammen den Editor, der die gleichen Eigenschaften besitzt wie der bereits bekannte Editor für die EMF-Modelle – tatsächlich ist der EMF-Modell-Editor der entsprechend generierte Editor des Ecore-Modells. Dieser generierte Editor ist ein Eclipse-Plug-in und verwendet die vom Eclipse UI her bekannten Klassen und Interfaces, etwa *TreeViewer* und *PropertySheet*. Um das Modell an die speziellen Anforderungen der Eclipse-Klassen anzupassen, werden im Projekt *portal.edit* entsprechende Adapter (nach obigem Muster) der Modellklassen bereitgestellt, die die jeweiligen Modellelemente an die benötigen Schnittstellen des Eclipse UI anpassen. Das eigentliche Editor-Plug-in ist in *portal.editor* implementiert.

Definition des PSM

Die mitgelieferten Codegeneratoren liefern schnellen und robusten Code, über die Adapter sind die Modelle an eigene Bedürfnisse ohne Änderung der Schnittstellen oder Implementierungen anpassbar und der erstellte Editor dürfte für die Erstellung z.B. von Daten für Testfälle ausreichen. Um einen Blick hinter die Kulissen der Codegenerierung von EMF zu bekommen und um vielleicht eigenen Code zu generieren, soll das Beispiel nun um genau diese Fähigkeit erweitert werden. Wie wäre es, wenn – etwa für einen Prototypen – direkt aus den erstellten Use Cases PHP-Skripte erstellt würden, die einen greifbareren Eindruck des Modells, etwa in Form navigierbarer Webseiten, vermitteln würden?

In den Begrifflichkeiten der MDA bedeutet dieser Schritt, aus einem PIM (Platform-Independant Model) – also in unserem Fall die Use Cases – ein PSM (Platform-Specific Model) zu generieren. Das Modell des PSM besteht in unserem Beispiel aus PHP-Klassen, die direkt im PHP-Code repräsentiert werden. Bevor wir uns die technischen Fragen dieses Übergangs zuwenden, müssen wir die Abbildungsvorschriften festlegen, die das eine Modell in das andere überführen.

Für unser Beispiel wählen wir ein sehr einfaches Schema: Alle Elemente des Use-Case-Modells werden in jeweils eine PHP-Klasse überführt. Diese PHP-Klassen enthalten Methoden zur Initialisierung und Darstellung. PHP-typisch wird dann dabei HTML ausgegeben, hier in Form von geschachtelten Tabellen. Die Hierarchie der Elemente wird über die Beziehungen im Use-Case-Modell definiert: Kind der Systemklasse ist die PHP-Klasse des jeweils gewählten Akteurs, die Kinder und Kindeskinder des Akteurs sind die gerade aktiven Use-Case-Klassen. Die *include*- und *extend*-Beziehungen der Use Cases werden über Navigation nachgebildet.

Die Struktur der Klassen und der Workflow orientieren sich, stark vereinfacht, an der Robustness Analysis [4]. Dort werden die Elemente des Use-Case-Diagramms in Boundary-, Controller- und Entity-Klassen transformiert, diese Umsetzung hätte allerdings den Rahmen des Beispiels gesprengt. Abbildung 6 zeigt die Ausgabe der erzeugten PHP-Skripte, die unterschiedlichen Bereiche sind dabei markiert: der System-Bereich grau, der Akteur blau und die Use Cases grün.

JET

Prinzipiell gibt es mehrere Frameworks und Möglichkeiten, um aus dem vorhandenen PIM ein PSM zu generieren. Da als Persistenz-Typ XML gewählt wurde, könnte man an den Einsatz eines XSLT-Prozessors und entsprechende XSL-Templates denken. In vorangegangenen Artikeln des *Java Magazins* wurde das open-Generator Framework [5] vorgestellt.

EMF bringt sein eigenes Generator-Framework mit: JET – Java Emitter Templates. JET ist ein auf Templates basierter Generator. Er befindet sich zusammen mit JMerge, einem Tool zum Zusammenführen generierter und manuell erstellter Codefragmente, im Paket *org.eclipse.emf.codegen*. Die Templates von JET verwenden eine ähnliche Syntax wie JSP. Der Code von JET ist tatsächlich ein entsprechend reduzierter JSP Parser. Für Webentwickler hat das den Vorteil, dass keine neue Syntax erlernt werden muss.

Die Ähnlichkeiten zu JSP bzw. Tomcat treten noch deutlicher zu Tage, wenn der Workflow von JET betrachtet wird: Aus einem Template wird zunächst eine Java-Klasse, die Template-Klasse, erstellt, ähnlich wie JSPs zunächst in Java-Servlet-Klassen übersetzt werden. Die erstellten Template-Klassen haben immer eine Funktion *generate(Object argument)* und liefern den generierten Code in einem String zurück. Auf das *argument* kann innerhalb des Templates direkt zugegriffen werden, ebenso auf den intern verwendeten StringBuffer (die Variable heißt *stringBuffer)*. Die Verwendung erfolgt analog der vordefinierten Variabeln *request, response, session* oder *out* bei JSPs. Das Argument ist ein beliebiges Java-Objekt und stellt meistens das Modell dar. In unserem Beispiel übergeben wir entweder das Systemobjekt oder das Objekt des jeweiligen Use Case bzw. Actor. Zwei ausführliche Tutorials zu

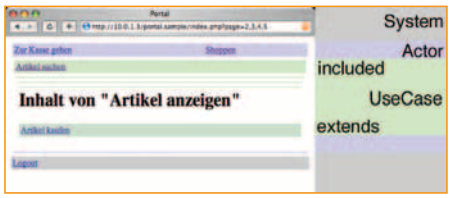


Abb. 6: Geschachtelte Elemente

JET sind in der Hilfe zu EMF enthalten, weswegen hier nicht weiter auf die JET-Syntax und andere Möglichkeiten, wie die Verwendung eigener Code-Skeletons, eingegangen werden soll. Listing 2 zeigt, wie dann in dem Beispiel aus einem Actor-Element das entsprechende PHP-Skript erzeugt wird, die anderen Templates finden sich auf der beiliegenden DVD. JET beziehungsweise alle *org.eclipse.emf.codegen*-Pakete sind prinzipiell unabhängig von EMF verwendbar.

Die Templates wurden direkt im Editor-Projekt im Verzeichnis *templates* angelegt. In der ersten Zeile des Templates wird der Name der zu erstellenden Template-Klasse definiert. Über File | New | Other | Java Emitter Templates | Convert Projects to JET Project konvertieren wir das Projekt *portal.editor* in ein JET Template. Danach müssen wir noch in den Projekteinstellungen unter JET Settings die Verzeichnisse anpassen (Source-Container *src*). JET übersetzt nun automatisch alle Templates im Template-Verzeichnis und legt den Java-Code im Source-Verzeichnis des Projekts ab. Änderungen an den Templates werden, wie beim inkrementellen Übersetzen der Java-Klassen im JDT, automatisch durchgeführt.

Erweiterung des Editors

Um in unsrem Editor aus den Use-Case-Modellen mittels der Templates entsprechende PHP-Seiten zu generieren, müssen die Template-Klassen vom Editor passend aufgerufen werden. Wir ergänzen dazu das Kontextmenü der Baumknoten des Editors um einen neuen Eintrag Generate PHP Scripts. Dieser Eintrag wird aktiv, sobald der System-Knoten ausgewählt wurde. Im Code müssen wir dazu an geeigneter Stelle der Klasse *PortalActionBar-Contributor* einen neuen Menüeintrag erzeugen.

Da der Editor automatisch generiert wurde (und damit auch der *PortalActionBarContributor*) besteht die Gefahr, bei einer erneuten Generierung, etwa aufgrund von Änderungen im Modell, die manuellen Eintragungen im Code zu verlieren. JMerge, zu finden im Paket *org.eclipse.emf.codegen.jmerge*, kann solche Fälle mittels Auszeichnungen im Code erkennen und entsprechend behandeln. Alle mittels JET erstellten Klassen, Methoden und Attribute bekommen im jeweiligen Javadoc-Teil ein Tag *@generated* zugewiesen. Bei einer erneuten Generierung werden die Elemente überschrieben, die diese Auszeichnung tragen. Um automatisch generierten Code und manuelle Fragmente zu kombinieren, können generierte Methoden mit dem Postfix *Gen* versehen werden. Diese Methoden können dann von der manuell erstellten Methode ohne das entsprechende Postfix aufgerufen werden. Beim Generieren werden dann anstelle der eigtl. Methoden die *Gen*-Methoden neu erstellt. Listing 3 zeigt den entsprechenden Ausschnitt aus dem *PortalActionBarContributor*.

Die eigentliche Erzeugung der PHP-Dateien implementieren wir der Klasse *PHPGenerateAction*, die wir als innere Klasse des *PortalActionBarContributor* definieren. Um alles einfach zu halten, werden die Skripte in einem Projektverzeichnis *php* angelegt und jedes Mal eventuell vor-

Listing 2

```
<%@ jet
  package="templates" class="ActorTemplate"
  imports="java.util.* * presentation.*"
%><?php
<% // Erzeugen der include_once Befehle
  Actor actor = (Actor) argument;
  PortalElement element;
  List listUseCases = actor.getUseCases();
  String str;

  for (Iterator iter = listUseCases.iterator(); iter.hasNext();) {
    element = (PortalElement) iter.next();
    str = PHPUtil.toClassName(element.getName());
     stringBuffer.append("include_once('")
                          .append(str).append(".php');\n");
  }
%>

class <%= PHPUtil.toClassName(actor.getName()) %> {

  public $content;
  public $pathprefix="";

  public function createContent($i_iID) {
    switch ($i_iID) {
     <%  // Erzeugen der Befehle zum Initialisieren der
         //geschachtelten UseCases
              for (Iterator iter = list-UseCases.iterator();
                                          iter.hasNext();) {
       element = (PortalElement) iter.next();
       str = PHPUtil.toClassName(element.getName());
       stringBuffer.append("    case ")
         .append(str.toUpperCase()).append(": $this->
      content = new ").append(str).append("(); break;\n");
       }
     %>
    }
    return $this->content;
  }

  public function process($i_aiPath, $i_iLevel) {
    if ($i_iLevel < count($i_aiPath)) {
      $this->createContent($i_aiPath[$i_iLevel]);
      if(isset($this->content)) {
        $this->content->process($i_aiPath, $i_iLevel+1);
      }
    }
    $this->pathprefix=computePathPrefix($i_aiPath,
                                          $i_iLevel);
  }

  public function draw() {
    ?>
<!-- System output -->
<table border="0" width="100%">
  <tr><td><!-- included usecases -->
    <table bgcolor="#ccccff" width="100%"><tr>
      <%
        for (Iterator iter = listUseCases.iterator();
                                          iter.hasNext();) {
        element = (PortalElement) iter.next();
        str = PHPUtil.toClassName(element.getName());
        stringBuffer.append("    <td><a href=\"index.
          php?page=<?php print($this->pathprefix); print('')
          .append(str.toUpperCase()).append("); ?>\">")
    .append(element.getName()).append("</a></td>\n");
        }
      %>
    </tr></table>
  </td></tr>
  <tr><td><!-- current selected usecase or index of this e
                    on --><?php $this->drawContent(); ?>
  </td></tr>
</table>
<!-- end System output -->
    <?php
  }

  protected function drawContent() {
    ...
  }
}
?>
```

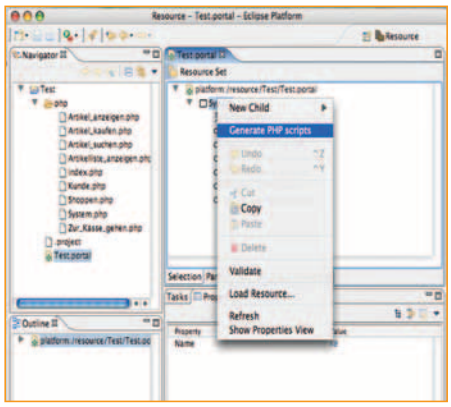


Abb. 7: Der fertige Editor

handene Dateien überschrieben. Listing 4 zeigt die für die Generierung verantwortlichen Teile, Abbildung 7 zeigt den fertigen Editor mit dem neuen Eintrag im Kontextmenü sowie die im Projekt erstellten PHP-Dateien.

Ausblick

EMF ist ein äußerst mächtiges Framework, mit dem sich sehr schnell erstaunliche Ergebnisse erzielen lassen. Mit ein wenig zusätzlichem Aufwand könnte so das besprochene Beispiel tatsächlich im Rahmen der Anforderungsanalyse und im Dialog mit Kunden oder Grafikern eingesetzt werden.

Die im EMF-Plug-in enthaltenen Module lassen sich in das Metamodell Ecore und den Codegenerator JET/JMerge unterteilen. Für die Praxis wird v.a. die Fähigkeit interessant sein, mittels EMF eigene stabile, im XMI-Format serialisierbare Modelle zu generieren. Ein weiteres Eclipse Tool Project, UML2, stellt übrigens eine auf EMF basierende Implementierung des UML 2.0-Metamodells bereit. Allerdings stellt UML2 ausschließlich das Modell bereit, ein grafischer Editor für UML-Diagramme ist im Rahmen dieses Projekts weder enthalten noch geplant. EclipseUML von Omondo, dessen freie Version Ende Oktober 2004 erschienen ist, basiert auf UML2 und kann entsprechend verwendet werden. Auch die neuen Versionen der Rational-Entwicklertools werden EMF und UML2 für die Modelle einsetzen. Damit ist eine einfache Integration unterschiedlicher Tools möglich, da sich die Modelle über die Verwendung des gemeinsamen Metamodells EMF/UML2 leicht austauschen lassen.

Wem die baumorientierten Editoren nicht ausreichen, der kann mit dem Graphical Editing Framework, kurz GEF (siehe S. 85), ebenfalls ein Eclipse Tool Project, eigene grafische Editoren erstellen und dabei das Modell mit EMF generieren lassen. GEF verwendet das MVC Pattern und kann dabei direkt den Notification-Mechanismus der EMF-Modelle verwenden. Ein IBM Redbook hilft beim Einstieg sowohl in EMF als auch GEF und zeigt, wie beide Frameworks gemeinsam verwendet werden können.

Jens von Pilgrim arbeitet z.Zt. an einer Diplomarbeit zum Thema grafische Editoren. Er kann außerdem auf über acht Jahre berufliche Erfahrung im Bereich Java und Webapplikationen zurückblicken.

Links & Literatur

[1] www.eclipse.org/emf, www.eclipse.org/uml2, www.eclipse.org/gef

[2] David S. Frankel: Model Driven Architecture, Wiley, 2003

[3] Frank Budinsky u.a.: Eclipse Modeling Framework, Addison-Wesley, 2004

[4] Dough Rosenberg, Kendall Scott: Use Case Driven Object Modeling with UML, Addison-Wesley, 1999

[5] Marc Thomas, Kasten Thoms: Die MDA-Bank. In *Java Magazin* 11.2004; Sven Efftinge: MDA praktisch – mit dem open ArchitectureWare Generator, in *Java Magazin* 1.2005

[6] Bill Moore u.a.: Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework, IBM Redbook, 2004, URL: www.redbooks.ibm.com/abstracts/sg246302.html

Listing 3

```
/**
 * This populates the pop-up menu before it appears.
 * <!-- begin-user-doc --
 * <!-- end-user-doc -->
 * @generated
 */
public void menuAboutToShowGen
                        (IMenuManager menuManager) {
  ...
}

protected IAction generatePHPAction =
        new PHPGenerateAction("Generate PHP scripts");

public void menuAboutToShow(IMenuManager
                                        menuManager) {
  menuAboutToShowGen(menuManager);
  menuManager.insertBefore("additions",
                    new Separator("generate-actions"));
    menuManager.insertAfter("generate-actions",
                                    generatePHPAction);
}
```

Listing 4

```
PortalSystem system = getModelFromSelection();
if (system==null) return;

...

String str;
try {
  // generate folder (assuming file editor input)
  IFileEditorInput editorinput = (IFileEditorInput)
    getActiveEditor().getEditorInput();
  String setCharset = editorinput.getFile().getCharset();
  IFolder folder =
    editorinput.getFile().getProject().getFolder("php");
  if (!folder.exists())
    folder.create(true, false, progressMonitor);

  // generate index.php
  IndexTemplate indextemplate = new IndexTemplate();
  str = indextemplate.generate(system);
  IFile f = folder.getFile("index.php");
  if (f.exists()) f.delete(true, true, progressMonitor);
  f.create(new ByteArrayInputStream(
      str.getBytes(setCharset)), false,
      progressMonitor);
  progressMonitor.worked(1);
...

  // generate system.php (equals index.php)
  ...

  // generate actor and usecase files
  UseCaseTemplate uctemplate = new UseCaseTemplate();
  ActorTemplate actortemplate = new ActorTemplate();
  for (Iterator iter =
    system.getPortalElements().iterator();
    iter.hasNext();) {
    PortalElement element = (PortalElement) iter.next();
    if (element instanceof Actor) {
      str = actortemplate.generate(element);
    } else { // instanceof UseCase
      str = uctemplate.generate(element);
    }
    f = folder.getFile(
        PHPUtil.toClassName(element.getName())+.php");
    if (f.exists()) f.delete(true, true, progressMonitor);
    f.create(new ByteArrayInputStream(
        str.getBytes(setCharset)), false,
        progressMonitor);
    progressMonitor.worked(1);
  }
} catch (Exception ex) {
  ...
}
```