Vol. 13
Ausgabe 1.08

Deutschland € 9,80
Österreich € 10,80, Schweiz CHF 19,20

eclipse MAGAZIN

eclipse MAGAZIN

www.eclipse-magazin.de

Mit CD
Alle Infos S. 3

Tools, Open Source & more

- Eclipse Visual Editor Project
- Apache Tomcat 6.0
- ATLAS Transformation Language
- Eclipse Modeling Framework (EMF)
- Eclipse Graphical Modeling Framework (GMF)
- Graphical Editing Framework (GEF)
- Standard Widget Toolkit (SWT)

BONUS:
Artikelserie aus dem Java Magazin „OSGi applied“
+ Bonusartikel aus dem Eclipse Magazin Vol. 12

Professionell Testen

Best Practices: GUI-Tests für Eclipse RCP

» Embedded Eclipse:
Model Driven Development bei eingebetteten Systemen

Praxis
Build-Prozess von Eclipse Plug-ins

Rich Clients
Neu seit Eclipse 3.3: Eclipse DataBinding

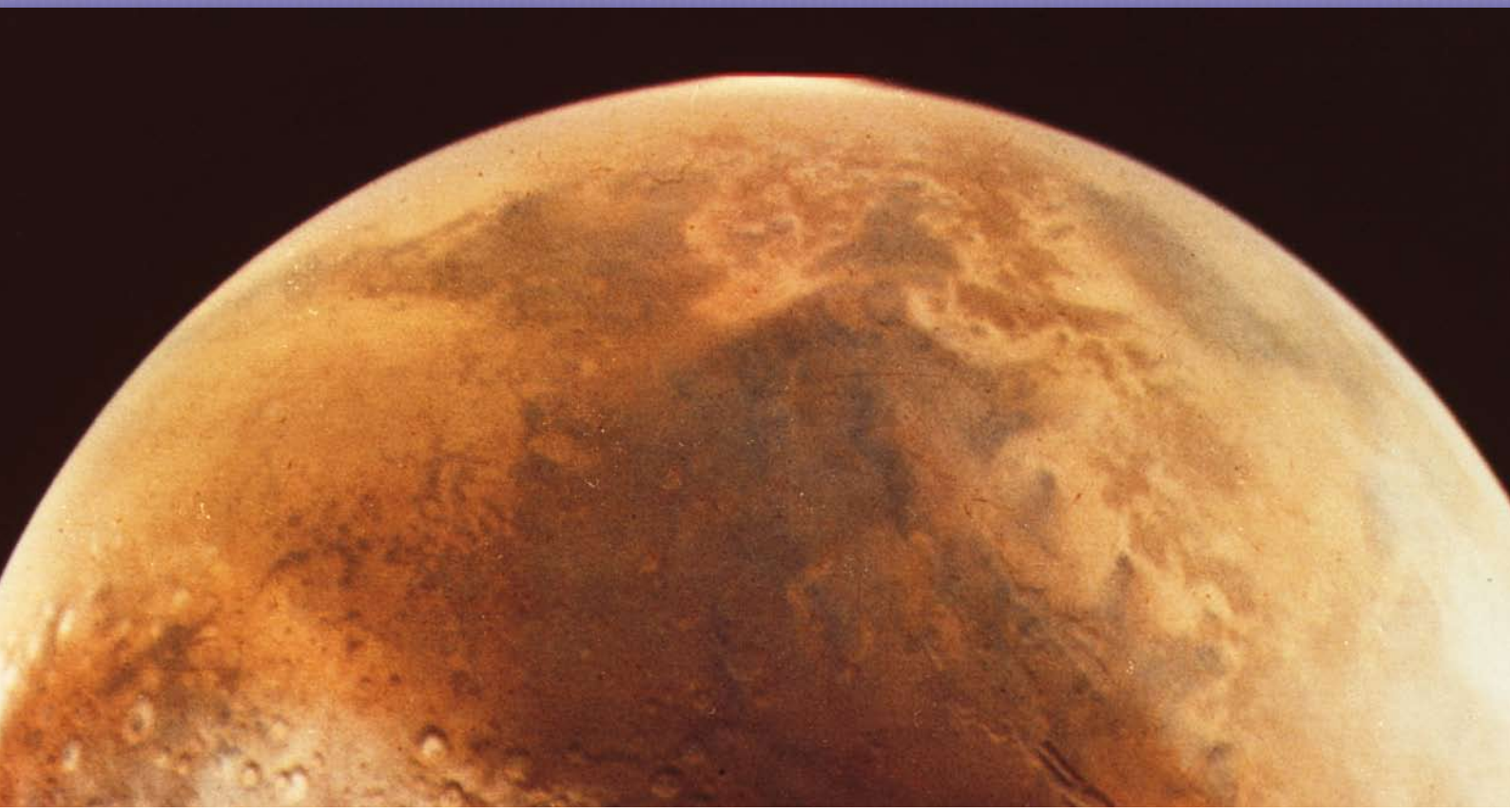
Model Driven
Dokumentationen generieren

Plattformen
Eclipse und NetBeans Platform im Vergleich

D 68864

Datenträger enthält nur Lehr- oder Infoprogramme

4 196603 209806 01



Dynamische Erzeugung eines Generators für Modelldokumentationen

Mercurius – der Götterbote mit EMF und oAW

>> PASCAL ALICH, PETER FRIESE, ANDRÉ LAHS UND HOLGER SCHILL

Die Plattform GMF ermöglicht es, die für die modellgetriebene Softwareentwicklung wichtigen grafischen Editoren zu generieren. Häufig ist es erforderlich, das Modell zu dokumentieren. Möchte man die Dokumentation nicht per Hand auf dem aktuellen Stand halten, sollte auch hier generiert werden. Bisher war für jedes Metamodell ein neuer Generator zu erstellen, doch das neue Konzept „Mercurius" vereinfacht dies und greift Ideen von GMF auf.

Die Erstellung von aktueller Dokumentation eines Softwaresystems ist aufwändig und teuer, und in zeitkritischen Phasen eines Projekts bleibt manuell gepflegte Dokumentation leicht auf der Strecke [1]. Eine Dokumentation ist allerdings ein wichtiges Artefakt eines Softwaresystems: Sie gibt aus verschiedenen Perspektiven einen Einblick in das System, z.B. auf das fachliche Modell, und dient damit als Kommunikationsgrundlage zwischen Entwicklern und Anwendungsexperten. In iterativ-inkrementellen Projekten veraltet Dokumentation jedoch sehr schnell und verliert damit an Wert für die Beteiligten. In der modellgetriebenen Softwareentwicklung (MDSD) sind semantisch reiche Domänenmodelle die Grundlage der Entwicklung, da aus diesen Teile des Softwaresystems generiert werden. Die Domänenmodelle sind stets zum System synchron. Sie werden mit Domain Specific Languages (DSL) beschrieben, die für bestimmte Projekte oder Produktlinien individuell erstellt werden, z.B. in Form eines UML-Profils oder mithilfe einer Metamodellierungssprache wie Ecore aus dem Eclipse-Projekt EMF [2]. Diese Modelle werden häufig als Diagramm dargestellt. Diagramme sind in ihrer Detailliertheit beschränkt, damit die Übersichtlichkeit gewahrt bleibt.

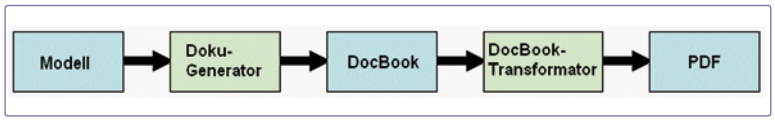


Abb. 1: Ablauf der Dokumentationsgenerierung

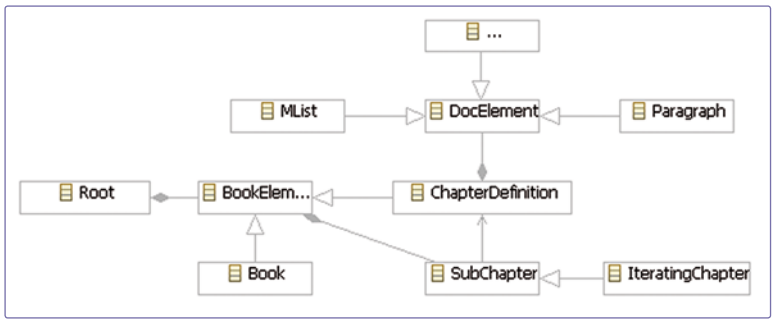


Abb. 2: Mercurius-Metamodell

In Textdokumenten hingegen können Softwaresysteme teilweise besser und detaillierter beschrieben werden. Mithilfe eines auf die DSL zugeschnittenen Generators können ohne Probleme aus den Modellen reichhaltige, aktuelle und damit nutzbare Textdokumentationen erzeugt werden. Obwohl die grundlegende Arbeit stets die gleiche ist, muss ein solcher Dokumentationsgenerator für jede DSL individuell erstellt werden.

Der Aufwand für den Bau von Dokumentationsgeneratoren könnte durch die Mittel der MDSD deutlich reduziert werden, indem DSL-Spezifika in einem Modell beschrieben werden und der individuelle Dokumentationsgenerator aus diesem Modell generiert wird. Um die Machbarkeit dieses Vorhabens zu belegen, wurde an der FH Nordakademie [3] im Rahmen eines Wahlpflichtkurses ein studentisches Projekt unter der Leitung von Prof. Dr. Frank Zimmermann und in Kooperation mit Dr. Marko Boger sowie Peter Friese von der Gentleware AG [4] initiiert. Ziel war die Entwicklung einer Plattform, mit der für beliebige in Ecore definierte DSLs Dokumentationsgeneratoren generiert werden können. Weitere Rahmenbedingungen für das Projekt waren die Verwendung von openArchitectureWare (oAW) [5] für den Bau der Generatoren und die Nutzung von DocBook [6] als Zwischenausgabeformat des Dokumentationsgenerators.

openArchitectureWare (oAW)

openArchitectureWare (kurz: oAW) ist eine auf der Eclipse-Plattform basierende Sammlung von Werkzeugen für die modellgetriebene Softwareentwicklung:

- Ein Workflow-Mechanismus ermöglicht die Definition komplexer Abläufe von Transformationen.
- Xtend reichert Metamodelltypen durch Extensions mit Logik an und ermöglicht Modell-zu-Modell-Transformation.
- Check ist das Pendant zur Object Constraint Language (OCL).
- Xpand als mächtige Template-Sprache für diverse Ausgabeformate (Modell-zu-Text-Transformation).
- Das Recipe Framework prüft Artefakte außerhalb der Generierung (z.B. abgeleitete Java-Klassen) auf Korrektheit und meldet die Ergebnisse an Eclipse als Entwicklungsumgebung zurück.

Im Rahmen des in diesem Artikel dargestellten Projekts wurden der Workflow-Mechanismus, Xtend und Xpand verwendet. Mehr Infos zu oAW finden Sie unter [5] und [7].

Ein Dokumentationsgenerator

Bevor mit dem Bau einer Plattform zur Generierung von Dokumentationsgeneratoren begonnen werden kann, ist es sinnvoll, sich zunächst mit der Funktionsweise des Endprodukts zu beschäftigen. Wie in Abbildung 1 dargestellt, soll ein Dokumentationsgenerator ein Modell als Eingabe erhalten. Dieses Modell soll durch den Generator im ersten Schritt in ein DocBook-Zwischendokument transformiert werden, das dann in einem zweiten Schritt wiederum durch einen DocBook-Transformator in ein beliebiges Ausgabeformat, z.B. PDF, umgewandelt wird. Neben diesem grundlegenden Ablauf der Dokumentationserzeugung existierte die Anforderung der möglichen Modifikation der Dokumentation durch den Anwender, sodass er z.B. vor der endgültigen Transformation einzelne Kapitel verschieben oder ein- und ausblenden kann.

Die Domäne „Dokumentationserzeugung“

Für den Bau von Dokumentationsgeneratoren ist zunächst ein Verständnis über den Aufbau von Dokumentationen und der Darstellung von Modellen einer DSL in textueller Form notwendig. Eine Dokumentation, respektive ein Textdokument, besteht zunächst einmal aus einer Menge von ineinander geschachtelten Kapiteln. Die Schachtelung der Kapitel wird als Gliederung oder grobe Struktur eines Dokuments verstanden. Diese Gliederung wird nachfolgend als Makrostruktur einer Dokumentation bezeichnet. Eine Dokumentation, die ausschließlich aus einer Makrostruktur besteht, ist für die Leser allerdings nicht sehr aussagekräftig. Es fehlt in dieser Dokumentation an Inhalt. Inhalt wird in Dokumenten durch Elemente wie Absätze, Listen, Tabellen und auch Abbildungen in den einzelnen Kapiteln dargelegt. Diese Elemente definieren den Aufbau von Kapiteln, der im Folgenden als Mikrostruktur bezeichnet wird.

Ein Modell einer DSL besteht aus Elementen. Welche Elemente in einem Modell vorkommen und in welchen Beziehungen diese Elemente stehen, definiert das Metamodell. Beispielsweise besteht ein UML-Modell aus Paketen, Klassen, Assoziation und vielen weiteren Elementen. Um ein UML-Modell zu dokumentieren, muss zunächst überlegt werden, wie die Struktur dieses Modells in eine Makrostruktur einer Dokumentation übersetzt werden kann. Dazu ist es nötig, festzulegen, welche Elemente aus dem Modell, respektive dem Metamodell, in der Dokumentation als Kapitel dargestellt werden sollen und welche nicht. Beispielsweise soll für jedes Paket und jede Klasse des Modells ein Kapitel bestehen, nicht aber für Attribute oder Operationen. Außerdem muss definiert werden, wie diese Kapitel ineinander geschachtelt werden, z.B. soll ein Kapitel aus Unterkapiteln für jedes enthaltene Paket und für jede enthaltene Klasse bestehen. Neben der Definition der Makro-

struktur muss auch die Mikrostruktur beschrieben werden. Dazu spezifiziert man, welche Elemente in Form von Absätzen, Listen oder dergleichen dargestellt werden sollen. Im UML-Beispiel könnten z.B. alle Attribute einer Klasse in Form einer Liste und der Klassenkommentar als Absatz im Kapitel der Klasse dargestellt werden. Unabhängig davon, ob die Dokumentation eines Modells manuell oder automatisch erstellt wird, benötigt man Transformationsregeln, welche die Übersetzung von Modellelementen in Dokumentationselemente beschreiben müssen. Aus Sicht der MDSD sind diese Transformationsregeln der Kern der Domäne „Dokumentationserzeugung“.

Modellierung eines Dokumentationsgenerators

Ein Dokumentationsgenerator ist eine Umsetzung der Transformationsregeln zur Übersetzung von Modellelementen in Dokumentationselemente. Für die Generierung eines Dokumentationsgenerators müssen diese Transformationsregeln bekannt sein. Ein Modell einer beliebigen DSL beschreibt Sachverhalte der jeweiligen Domäne, doch beinhaltet diese keine Informationen zur Erzeugung einer Dokumentation. Eine direkte Erweiterung der DSL um diese Aspekte ist aufgrund des Prinzips „Separation of Concerns“ nicht sinnvoll. Die GMF-Plattform [9] zur Erstellung von grafischen Editoren steht dem gleichen Problem gegenüber. EMF-Metamodelle enthalten keine Informationen über eine mögliche grafische Notation. Diese bildet jedoch die Grundlage für die Erstellung eines grafischen Editors und definiert die Darstellung eines Elements in einem Diagramm. Diese Informationen werden in GMF über separate Modelle konfiguriert, die eine Zuordnung von Metamodellelementen zu Notationselementen, wie z.B. Rechtecke oder Linien, definieren. Anhand der Notationsdefinition wird ein grafischer Editor in Form eines Eclipse Plug-ins generiert. Analog zum Ansatz aus GMF, separate Modelle zur Beschreibung eines grafischen Editors zu nutzen, wird in diesem Projekt ein eigenes Modell zur Beschreibung der benötigten Transformationsregeln verwendet. Darin sind ausreichend Informationen zur Generierung eines Dokumentationsgenerators gegeben, mit dem aus einem Modell eine initiale Dokumentation erzeugt werden kann.

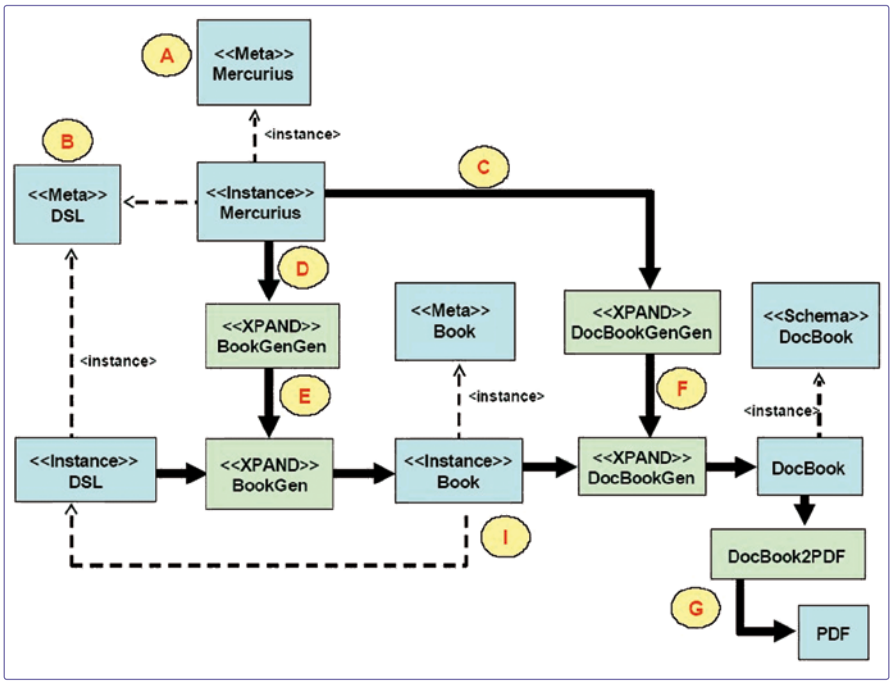


Abb. 3: Von einer DSL zur fertigen Dokumentation

Eine Kernaufgabe des Projekts war nun die Entwicklung einer DSL zur Beschreibung von Dokumentationsgeneratoren. Diese DSL, nachfolgend als „Mercurius“ bezeichnet, definiert die Transformationsregeln zur Übersetzung von Modellelementen in Dokumentati-

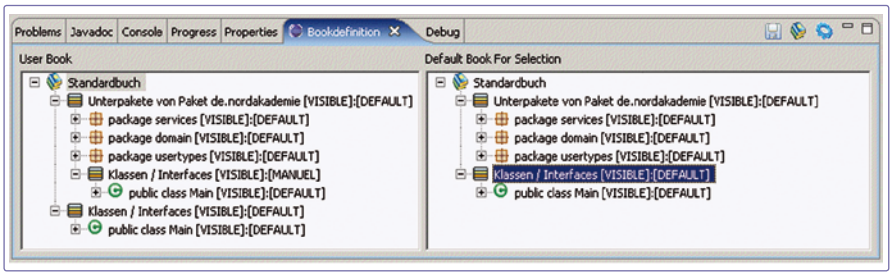


Abb. 4: View zur Bearbeitung eines Book-Modells

onselemente. Dazu legt Mercurius durch *ChapterDefinitions*, wie im Metamodell von Mercurius in Abbildung 2 dargestellt, fest, welche Modellelemente durch ein Kapitel in der Dokumentation repräsentiert werden sollen. *ChapterDefinitions* referenzieren dazu das Element aus dem Metamodell der DSL, deren Modelle dokumentiert werden sollen. Dieses Metamodell soll nachfolgend als Ausgangsmetamodell bezeichnet werden. Ein Exemplar von Mercurius ist daher auf der Metaebene des Ausgangsmetamodells anzusiedeln. Mercurius selbst ist aufgrund der bereits beschriebenen Anforderungen in Ecore definiert. Ecore erlaubt die Verknüpfung von Modellen verschiedenster Metaebenen, wodurch eine Referenzierung des Ausgangsmetamodells in einem Exemplar von Mercurius keine Hürde darstellt.

Innerhalb einer *ChapterDefinition* wird die Mikrostruktur, also der Aufbau des Kapitels, definiert. Die Mikrostruktur wird durch *DocElements*, wie etwa Absätze, Listen, Tabellen etc., beschrieben. *DocElements* beziehen sich dazu auf die Eigenschaften des referenzierten Elements, also auf Attribute oder andere assoziierte Elemente. Beispielsweise könnte in einem Mercurius-Modell für die UML eine *ChapterDefinition* für das Element *Klasse* erstellt werden. Die Mikrostruktur eines Kapitels für das Element *Klasse* könnte aus einem Absatz mit dem Klassenkommentar und aus einer Auflistung der Attribute der Klasse bestehen. Für die Formatierung und Strukturierung von Textausgaben, z.B. den Listeneinträgen oder Absatztexten, wurde ein Platzhaltermechanismus eingeführt. Dieser Platzhaltermechanismus erlaubt es, Textbausteine mit Platzhaltern zu definieren, die später bei der Transformation durch Werte aus dem Modell, z.B. Klassen- oder Attributnamen, ersetzt werden.

Die Makrostruktur der Dokumentation wird über die Konstrukte *BookElement* und *SubChapter* beschrieben. Das *BookElement Book* stellt das Wurzelelement einer Dokumentation dar und referenziert aus dem Ausgangsmetamodell ein Element, welches auch den Charakter eines Wurzelelementes besitzt, z.B. die Elemente *Model* oder *Package*, in der UML. Ein *Book* enthält zur weiteren Strukturierung *SubChapter*. *SubChapter* werden für das Wurzelelement selbst oder für assoziierte Elemente, die über eine Navigation vom Wurzelelement erreicht werden können, definiert. Beispielsweise können in einem *Book*, dessen Wurzelelement ein *Package* ist, *SubChapter* für die im *Package* enthaltenen Klassen, Interfaces und Packages erstellt werden. Außerdem wird jedem *SubChapter* eine *ChapterDefinition* zugeordnet, welche zur Beschreibung der Mikrostruktur der Kapitel verwendet werden soll. Die für ein *SubChapter* ausgewählte *ChapterDefinition* muss natürlich typkompatibel zum referenzierten Element sein. Die Zuordnung von *SubChapter* zu einem *Book* beschreibt aber nur die oberste Ebene der Makrostruktur. Aus diesem Grund können auch einer *ChapterDefinition SubChapter* für assoziierte Elemente zugeordnet werden. Durch diese indirekte Rekursion kann eine tief geschachtelte und zur Struktur des Ausgangsmetamodells korrespondierende Makrostruktur einer Dokumentation beschrieben werden.

Ausschnitt aus dem Metagenerator DocBookGenGen

```
«DEFINE DocBookTemplate FOR mercurius::Chapter»
«o()»DEFINE «this.name“ FOR «this.rootElement.
                        getFullyQualifiedName()»«c()»
    «EXPAND DocElement FOREACH this
                                        .docElement»
«o()»ENDDEFINE«c()»
...
«ENDDEFINE»
...
«DEFINE DocElement FOR mercurius::MList»
  «o()»IF «this.selectedChild.name».size > 0«c()»
  <simplelist>
  «o()»FOREACH «this.selectedChild.name» AS _
                                attribute__variable«c()»
    <member>«this.contentLabel.performTemplate
              (“_attribute__variable“)“</member>
  «o()»ENDFOREACH«c()»
  </simplelist>
  «o()»ENDIF«c()»
«ENDDEFINE»
...
«DEFINE DocElement FOR mercurius::Paragraph»
  <para>«this.contentLabel.performTemplate
                                    (“this“)»</para>
«ENDDEFINE»
```

Ausschnitt aus spezialisiertem DocBookGen

```
...
«DEFINE MClassChapter FOR mini::MClass»
  <para>Die Klasse«this.name» wird hier
          beschrie-ben. «this.description»</para>
  «IF attribute.size > 0»
  <simplelist>
  «FOREACH attribute AS _attribute__variable»
    <member>«_attribute__variable.visibility»
          «_attribute__variable.type»»_attribute_
                              _variable.name“</member>
  «ENDFOREACH»
  </simplelist>
  «ENDIF»
«ENDDEFINE»
...
```

Von einer DSL zur fertigen Dokumentation

Mit Mercurius steht nun eine DSL zur Verfügung, anhand derer Dokumentationsgeneratoren für beliebige Metamodelle in einem Modell beschrieben werden können. Für ein beliebiges Ausgangsmetamodell kann ein Mercurius-Modell definiert werden. Aus diesem Mercurius-Modell kann nun ein Dokumentationsgenerator generiert werden, der ein Exemplar des Ausgangsmetamodells in ein DocBook-Dokument überführt. Eine weitere Anforderung war jedoch, dass der Nutzer des Dokumentationsgenerators die Makrostruktur der Dokumentation zur Laufzeit und vor der Erzeugung des DocBook-Dokuments, z.B. durch das

Verschieben von Kapiteln, beeinflussen kann. Um diese Anforderung erfüllen zu können, reicht ein Dokumentationsgenerator, der direkt aus dem Modell ein DocBook-Dokument erzeugt, nicht aus.

Die Idee zur Lösung des Problems bestand in der Verwendung eines Zwischenmodells, welches die Makrostruktur der Dokumentation aufnimmt und durch den Anwender modifiziert werden kann. Ein solches Zwischenmodell wird anschließend nach DocBook überführt. Dazu wurde ein weiteres Metamodell mit dem Namen *Book* eingeführt. Ein Book-Modell besteht aus ineinander geschachtelten Kapiteln. In jedem Kapitel ist vermerkt, welches konkrete Modellelement damit verbunden ist, und aufgrund welcher *ChapterDefinition* die Mikrostruktur des Kapitels erzeugt werden soll. Durch die Einführung dieses Zwischenmodells sind allerdings jetzt zwei Generatoren erforderlich. Der erste Generator überführt das zu dokumentierende Modell in ein Book-Modell. Dieser Generator wird nachfolgend als *BookGen* bezeichnet. Der zweite Generator, *DocBookGen*, hat die Aufgabe, das Book-Modell in das DocBook-Format zu transformieren. Anhand des Book-Modells weiß *DocBookGen*, wie Kapitel im DocBook ineinander geschachtelt werden sollen. *DocBookGen* selbst enthält für jede *ChapterDefinition* ein Template zur Generierung der Mikrostruktur. Im Book-Modell ist vermerkt, welche *ChapterDefinition* zur Erzeugung einer Mikrostruktur eines Kapitels verwendet werden soll. Anhand dieser Information wählt *DocBookGen* das richtige Template zur Generierung aus.

Die gewonnenen Erkenntnisse und die dargestellten Ideen münden in die Architektur der Plattform zur Generierung von Dokumentationsgeneratoren, die in Abbildung 3 dargestellt ist. In diesem Diagramm sind die benötigten Komponenten, deren Abhängigkeiten und der gesamte Workflow enthalten. Der Workflow beginnt mit der Definition der Transformationsvorschriften durch ein Mercurius-Modell (A) für ein Ausgangsmetamodell (B). Das Mercurius-Modell dient als Eingabe für den Metagenerator *BookGenGen* (D), der den spezifischen Generator *BookGen* (E) generiert. Auch geht das Mercurius-Modell in den Metagenerator *DocBookGenGen* (C) ein, der den spezifischen Generator *DocBookGen* erzeugt (F). Die Definition des Mercurius-Modells und der Einsatz der Metageneratoren erfolgt zur Entwicklungszeit des Dokumentationsgenerators. Zur Ausführungszeit des Dokumentationsgenerators beginnt der Workflow mit der Bearbeitung des Exemplars des Ausgangsmetamodells, z.B. ein UML-Modell. Dieses wird durch den *BookGen* (E) in das Book-Modell überführt und kann durch den Anwender modifiziert werden. Das modifizierte Book-Modell wird durch den *DocBookGen* (F) in ein valides DocBook transformiert. Das DocBook-Exemplar wird abschließend durch einen geeigneten Transformator (G) beispielsweise in das PDF-Format umgewandelt.

Eine Änderung des Ausgangsmodells erfordert bei dieser Vorgehensweise einen erneuten Durchlauf des Workflows

DocBook

DocBook ist ein XML-Format für Dokumente wie Bücher, Artikel und Dokumentationen. DocBook wurde in diesem Projekt als Zielsprache gewählt, weil es sich im Wesentlichen auf die Struktur eines Dokuments beschränkt und die Darstellung desselben außen vor lässt. Zudem existiert eine Vielzahl frei verfügbarer Transformatoren, die aus DocBook-Dateien PDF-, Word- und HTML-Dateien sowie Eclipse Help erzeugen können. Damit ist die Anforderung eines beliebigen Zielformats im Wesentlichen abgedeckt. In diesem Artikel wird daher nur auf die Generierung von DocBook-Dateien aus Modellen eingegangen. Der interessierte Leser sei bezüglich der weiteren Transformation auf [8] verwiesen. Um DocBook erzeugen zu können, müssen die Elemente aus dem Ursprungsmodell mit DocBook-Elementen wie Kapiteln, Absätzen und Listen in Beziehung gebracht werden.

zur Aktualisierung der Dokumentation. Bei einer erneuten Generierung des Book-Modells würden prinzipiell die Änderungen des Anwenders verloren gehen. Damit dies nicht geschieht, wurde ein Synchronisationsmechanismus zwischen der neuen und der alten Version des Book-Modells eingeführt, der aus Platzgründen nicht weiter erläutert wird.

Umsetzung

Bei der Umsetzung des Mercurius-Projekts stand natürlich die Generierung der Generatoren *BookGen* und *DocBookGen* im Vordergrund. Die dafür benötigten Metageneratoren *DocBookGenGen* und *BookGenGen* wurden dazu in XPAND, der Modell-zu-Text-Transformationssprache von oAW, entwickelt. Die Eingabe für diese Metageneratoren ist ein Exemplar des Mercurius-Metamodells für ein spezielles Ausgangsmetamodell. Das Ergebnis der Transformation sind die speziell dem Ausgangsmetamodell angepassten Generatoren *BookGen* und *DocBookGen* in Form von XPAND Templates. Listing 1 zeigt einen Ausschnitt aus dem Metagenerator *DocBookGenGen*, mit dem beispielsweise das Template aus Listing 2 generiert werden kann. Prinzipiell ist die Generierung eines Generators unproblematisch, da es sich um eine reine Modell-zu-Text-Transformation handelt. Die einzige Hürde bestand darin, dass Generator und Generat in der gleichen Sprache sind, und dass XPAND den schematischen Code innerhalb der Templates nicht gesondert, z.B. durch Anführungszeichen, markiert. Der XPAND-Code, der generiert werden soll, muss aber von dem XPAND-Code des Metagenerators unterscheidbar sein, da sonst syntaktische und semantische Fehler entstehen. Durch die Einführung von XTEND-Funktionen *«o()»* und *«c()»*, welche die öffnenden und schließenden Guillemets zur Auszeichnung der XPAND-Syntax als Konstanten zurückgeben, konnte diese Trennung realisiert werden.

Typischerweise sind die generierten Templates des Dokumentationsgenerators für einen Produktiveinsatz nicht ausreichend. Der Dokumentationsgenerator soll natürlich aus Eclipse heraus auf eine einfache Art und Weise verwendet werden können. Um dies zu ermöglichen, wird neben den beiden Generatoren auch die benötigte Infrastruktur für die Integration benötigt (hier: ein Eclipse Plug-in). Dieses Plug-in definiert eine *Action* als *ObjectContribution* auf Exemplare des Ausgangsmetamodells, um den *BookGen* zu starten. Das daraus generierte Book-Zwischenmodell wird nach der Generierung in einer *View* bereitgestellt (Abb. 4), mit deren Hilfe der Anwender das Zwischenmodell bearbeiten kann. Durch eine *Action* innerhalb der *View* kann der Anwender den finalen Workflow starten, um aus dem modifizierten Book-Modell ein DocBook-Dokument zu erzeugen und dieses danach in ein zuvor gewähltes Zielformat zu überführen. Die Bestandteile dieses Plug-ins konnten in generische und schematische Komponenten unterteilt werden. Bis zum Projektende gab es keine Anforderung, die das Einfügen von individuellem Code erforderte. Zu den generischen Komponenten des Plug-ins gehören die *View*, mit der das generische Book-Zwischenmodell modifiziert werden kann, und der DocBook-Transformator zur Erzeugung des Zieldokuments. Die übrigen Bestandteile, wie z.B. die *Actions*, die oAW-Workflows, die *Manifest.mf* oder die *plugin.xml*, sind aufgrund der Abhängigkeiten zum Ausgangsmetamodell von Dokumentationsgenerator zu Dokumentationsgenerator verschieden. Sie besitzen jedoch die gleiche Systematik und können somit aus einem Mercurius-Modell abgeleitet werden.

Fazit und Ausblick

Mit der Mercurius-Plattform wird ein Werkzeug zur Verfügung gestellt, mit dem Dokumentationsgeneratoren für beliebige DSLs auf Basis von EMF effizient und ohne individuelle Programmierung erstellt werden können. Der erfolgreichen Übertragung der Ideen aus GMF ist es zu verdanken, dass dieses Werkzeug für beliebige Ausgangsmetamodelle wieder verwendet werden kann. Insbesondere hat sich in diesem Kontext der Einsatz eines separaten Modells zur Beschreibung der zu erzeugenden Dokumentation bewährt, das durch generische Metageneratoren in einen DSL-spezifischen Dokumentationsgenerator überführt werden kann.

Mit dem Mercurius-Projekt konnte gezeigt werden, dass die Generierung von metamodellspezifischen Dokumentationsgeneratoren möglich ist. Der Einsatz der von EMF und GMF bekannten Technik, ein bzw. mehrere Modelle zur Konfiguration der Generatoren bzw. der Laufzeitumgebung zu verwenden, hat sich ebenfalls als tragfähig erwiesen. Das Hauptziel des Forschungsprojekts wurde also erreicht. Einige der anfänglich gestellten Anforderungen konnten jedoch aus Zeitmangel nicht umgesetzt werden und werden im Anschluss im Rahmen der Produktentwicklung bei Gentleware implementiert. Insbesondere handelt es sich hier um das Feature, grafische Darstellungen eines Pakets aus dem GMF-Modell automatisiert zu entnehmen und in die generierte Dokumentation einzubinden. Gentleware wird die Ergebnisse des Projekts im Rahmen der weiteren Produkt- und Projekttätigkeit anwenden und weiterentwickeln. Die Zusammenarbeit mit der Nordakademie hat sich als sehr effizient und fruchtbar herausgestellt.

Pascal Alich ist bei der PPI AG als IT-Berater angestellt. Kontakt: pascal.alich@ppi.de.

Peter Friese ist einer der beiden „Auftraggeber" des dargestellten Projekts und arbeitet als Senior Softwarearchitekt bei Gentleware. Kontakt: peter.friese@gentleware.com.

André Lahs arbeitet als Softwarearchitekt für die Gentleware AG in Hamburg. Er beschäftigt sich schwerpunktmäßig mit MDSD. Kontakt: andre.lahs@gentleware.com.

Holger Schill ist Softwareentwickler bei der Peter Kölln KGaA und hat seine Diplomarbeit über die Generierung von Grails-Anwendungen mit oAW geschrieben. Kontakt: h.schill@koelln.de.

>>Links & Literatur

[1] Stahl et. al: Modellgetriebene Softwareentwicklung, dpunkt.verlag 2007

[2] www.eclipse.org/modeling/emf/

[3] www.nordakademie.de

[4] www.gentleware.com

[5] www.openarchitectureware.org

[6] www.docbook.org

[7] www.eclipse.org/gmt/oaw/

[8] www.sagehill.net/docbookxsl/

[9] www.eclipse.org/modeling/gmf/