Architekturgetriebene Generierung für objektorientierte Softwaresysteme

Der Beitrag beschreibt einen neuartigen Ansatz zur Entwicklung von objektorientierten Softwaresystemen. Die architekturgetriebene Generierung verbindet zwei moderne Konzepte des Software-Engineering. Gemäß der Regel, dass das Ganze mehr ist als die Summe seiner Teile, können Codegenerierung und technische Architekturen derart miteinander verschmolzen werden, dass sich der Nutzen der beiden Technologien potenziert.

Codegenerierung ist eine Technologie, die bereits seit langer Zeit zur Verfügung steht. Ein Codegenerator ist z. B. der Teil eines Object Request Brokers (ORB), der aus CORBA-IDL Proxy-Kassen erzeugt. Bisher haben Generatoren noch keine breite Verwendung in der Praxis gefunden und dies, obwohl sie mit fast jedem OO Modellierungswerkzeug mitgeliefert werden.

In jüngster Zeit haben objektorientierte Architekturen zunehmend Aufmerksamkeit auf sich gezogen. Aktivitäten auf diesem Gebiet sind beispielsweise

- Enterprise Java Beans (EJB),
- die Common Object Request Broker Architecture (CORBA) und ihre Dienste
- das Distributed Component Object Model (DCOM) sowie
- Produkte für persistente Objekte.

Eine gute Übersicht über einige dieser Technologien ist in [Edw99] zu finden.

In den folgenden Abschnitten wird gezeigt, wie beide Technologien - Codegeneratoren und Architekturen - miteinander kombiniert werden können. Im letzten Abschnitt wird dann eine Anwendung dieser Vorgehensweise in einem Projekt, das von der SIGNAL IDUNA und der Firma Rösch Consulting in Kooperation durchgeführt wurde, beschrieben. Eine Besonderheit dieses Projekts ist, dass auf Seite des Servers COBOL (also eine nicht objektorientierte Programmiersprache) verwendet wurde. Dennoch konnte ohne Einschränkungen objektorientiert entwickelt werden, inklusive Vererbung und Polymorphismus. Dies unterstreicht nochmals die Leistungsfähigkeit des hier gezeigten Konzepts.

Codegenerierung

Unter Codegenerierung wird im Folgenden die automatische Erstellung von Quellcode durch ein Werkzeug verstanden (vgl. [Roe96]). Eine typische Eingabe für das Werkzeug ist ein UML-Massenmodell, eine typische Ausgabe eine Menge von Java-Code.

Wie in [Hub99] beschrieben, gibt es mit den üblichen OO-Werkzeugen zwei mögliche Verfahrensweisen für Design und Implementierung:

- Ein OO-Modell wird erstellt und von Hand ohne Generator codiert. Vorteil bei dieser Verfahrensweise ist die höhere Abstrahierung des Modells von der Implementierung. Nachteile sind der höhere Codierungsaufwand und mögliche Inkonsistenzen zwischen Modell und Code.
- Ein OO-Modell wird erstellt und über den integrierten Generator in Quellcode umgewandelt. Eine manuelle Codierung ist dann nicht mehr notwendig, aber das OO-Modell ist quellcode-äquivalent, d. h. jedes Element der Programmiersprache findet sich auch als ein Element im Modell wieder und umgekehrt.

Ein generierbares Modell ist im letzteren Fall nur eine grafische Darstellung des Quellcodes. Sobald das Modell eine gewisse Größe erreicht, geht auch die Übersichtlichkeit verloren.

Benötigt wird ein Codegenerator, der unter Berücksichtigung architekturspezifischer Regeln ein Modell auf Code abbildet und dabei bereits möglichst viele Methodenimplementierungen mitgeneriert. Dadurch erhält man sowohl ein ausschließlich fachlich motiviertes OO-Modell als auch eine automatisierte Codeerstellung.

Im Folgenden wird geschildert, welche Eigenschaften ein Generator haben muss und wie er entworfen und verwendet wird.

Anforderungen an einen Codegenerator

Aus der oben gegebenen Beschreibung lassen sich bereits einige Anforderungen an einen Generator ableiten. Die folgende Liste fügt weitere sinnvolle Anforderungen hinzu.

- Der Generator sollte nicht nur Methodenrümpfe, sondern - sofern möglich - auch die komplette Implementierung generieren. Kandidaten für eine automatische Generierung sind beispielsweise Methoden zum Setzen und Abfragen von Attributen.
- Damit eine wiederholte Generierung ohne Verlust möglich ist, sollten manuell er-

stellter und generierter Code nur an kontrollierten Stellen vermischt werden. Wird generierter Code von Hand angepasst, ist eine Neugenerierung unmöglich. Dies ist zu vermeiden, denn dieser Code muss von Hand gewartet werden.

- Der Generator muss anpassbar sein. Man benötigt volle Kontrolle über die generierten Klassen und Methodenimplementierungen. Die Anpassungen sollten leicht und mit minimalen Aufwand möglich sein.
- Die Konsistenz zwischen OO-Modell und Quellcode sollte im gesamten Lebenszyklus des Produkts garantiert werden.
- Es sollte jede beliebige Zielsprache generierbar sein.
- Der Generator sollte unabhängig von Architekturen, Entwicklungsumgebungen und Modellierungswerkzeugen einsetzbar sein. Das Werkzeug dient lediglich als eine Datenquelle für die Generierung und kann ausgetauscht werden.
- Jede beliebige Beschreibungssprache sollte als Datenquelle verwendet werden können. Mögliche Quellen wären z. B. UML Kassenmodelle oder eine DCOM-IDL-Datei.
- Der Generator sollte bei einer wiederholten Generierung nur die Dateien erzeugen, die von einer Änderung betroffen sind. Manuell erstellte Teile bleiben erhalten und nicht mehr benötigte Dateien werden gelöscht.

Verwenden Sie einen Codegenerator? Wie viele dieser Anforderungen erfüllt er? Erfahrungen aus Projekten haben gezeigt, dass der überwiegende Anteil der Generatoren nur wenige dieser Anforderungen erfüllt. Die meisten Generatoren sind fest in ein Modellierungswerkzeug integriert und nur beschränkt anpassbar. Die Anpassung verursacht erheblichen Aufwand und meistens wird nur eine einzige Zielsprache unterstützt. Für ältere Programmiersprachen, wie COBOL, PL/1 oder Natural, gibt es am Markt kaum geeignete Produkte.

Ein Codegenerator sollte also losgelöst vom Modellierungswerkzeug sein. Da die meisten Werkzeuge durch Skriptsprachen anpassbar sind, muss dennoch nicht auf eine komfortable Bedienung verzichtet werden.

Ein leistungsfähiger Codegenerator

Die Erstellung eines Generators nimmt viel Zeit in Anspruch. Das Werkzeug ist dann oft schwer zu warten und anzupassen. Für diese komplexen Aufgaben werden durchdachte Konzepte benötigt: Stanzformen und Metamodell-Transformationen

Stanzformen

Um konstante Teile einer Klasse zu generieren, kann der Generator diese mittels print-Befehlen erzeugen. Die Entwicklung des Generators wird dann allerdings sehr zeitaufwendig und der resultierende Code ist unübersichtlich und wartungsunfreundlich. Vorteilhafter ist es, die konstanten Teile der Zielsprache wie gewohnt in eine Datei zu schreiben und zusätzlich eine Generatorsprache in spezielle Markierungen einzubetten. Eine solche Datei wird Stanzform genannt und von einem Interpreter ausgeführt. Die eingebetteten Befehle steuern den Generator, konstanter Zielcode wird unverändert ausgegeben. Variable Daten - wie z. B. der Name der zu generierenden Klasse - kann die Generatorsprache aus einer sogenannten Metamodellinstanz ziehen, die zuvor aus der Eingabe erzeugt wurde. Abbildung 1 zeigt ein stark vereinfachtes Beispiel einer Stanzform. Befehle für den Generator werden mit einem vorangestellten #-Zeichen eingegeben.

```
package #thePackage->getName() ;
public class #theClass->getName()
   extends #theClass->getSuperclass->getName()
{
#FOR theOperation IN theClass->getOperations()
#INCLUDE "Operation.template"
#endfor
```

Abbildung 1 : Stanzform

Metamodell-Transformationen

Ein UML-Klassendiagramm ist eine Instanz des UML-Metamodells (vgl. [OMGUML]). Soll aus diesem Modell z.B. Java-Code erzeugt werden, wird eine Instanz des Metamodells der Java-Programmiersprache benötigt. Es muss also eine Transformation vom UML- in das Java-Metamodell durchgeführt werden. Die Transformation könnte direkt in der Stanzform implementiert werden. Beispielsweise würde eine Stanzform für einen CORBA-Proxy auf ihre UML-Metamodellinstanz zugreifen, die Transformation durchführen und den Proxy-Code erzeugen.

Es hat sich als ungünstig erwiesen, den Transformationscode in der Stanzform zu hinterlegen. Der resultierende Code wird für nicht triviale Transformationen unübersichtlich und schwer zu warten. Außerdem ist es nicht möglich, die Stanzform in einem anderen Projekt wiederzuverwenden.

Eine Lösung für dieses Problem ist die Metamodell-Transformation. Diese bildet ein Metamodell auf ein anderes Metamodell ab und wird außerhalb der Stanzform implemen-

tiert. Die Metamodelle selbst können allgemein gehalten und wiederverwendet werden.

Im Generator verwendete Metamodelle werden wie folgt charakterisiert:

- Ein Programmiersprachen-Metamodell bildet die möglichen Konstrukte der Programmiersprache 1-zu-1 auf ein Klassendiagramm ab. Dieses Metamodell kann eingesetzt werden, wenn Code für diese Sprache generiert werden soll.
- Ein UML-Metamodell wird in jedem Generator verwendet, der ein UML-Klassenmodell als Eingabe hat.

Um mit diesen beiden Metamodellen beispielsweise einen Java-Generator zu erstellen, wird eine Transformation des UML-Metamodell auf das Java Metamodell codiert. Abbildung 2 zeigt den Aufbau eines solchen Generators.

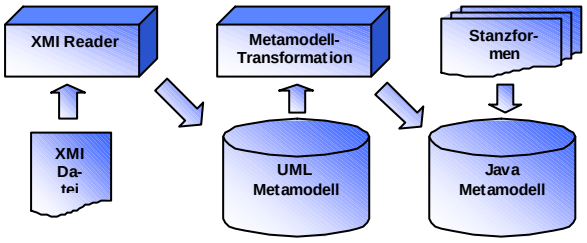

Abbildung 2 : Java Generator

Sollen gleichzeitig mehrere Zielsprachen generiert werden, hat sich die Verwendung einer mehrstufigen Transformation bewährt. Dabei wird zuerst das UML- auf ein Design-Metamodell transformiert, bevor die Zielsprachentransformationen auf das Design-Metamodell angewendet werden.

Ein beliebiger Codegenerator setzt sich dann aus den folgenden Teilen zusammen:

- allgemeine Metamodelle für die Eingaben und Zielsprachen,
- allgemeine Stanzformen für die Zielsprachen,
- architekturspezifische Metamodell-Transformationen und
- architekturspezifische Stanzformen zum Erzeugen von Methodenimplementierungen.

Generierungsfähige Softwarearchitekturen

Das hier beschriebene Generierungsverfahren hat eine Reihe von Vorteilen gegenüber der weitgehend manuellen Erstellung von Software, die heute noch vorherrscht. Diese Vorzüge entfalten sich dann in vollem Maße, wenn bei der Erstellung der Software eine generierungsfähige Softwarearchitektur benutzt wird. Die folgenden beiden Abschnitte beschreiben die zentralen Merkmale einer solchen Architektur. Anhand dieser Merkmale lässt sich erkennen, welche Eigenschaften die Generierungsfähigkeit von Architekturen steigern.

Trennung von Fachlichkeit und technischer Architektur

Anwendungssysteme weisen heute noch sehr oft einen hohen Grad von Abhängigkeit zwischen technischer Architektur und fachlichem Programmcode auf. Aufgaben der technischen Architektur – wie Transaktionsverarbeitung, Persistenz oder Objektaktivierung - sind mit der fachlichen Programmlogik vermischt. In diesen Fällen ist es sehr schwierig, die technische Architektur weiterzuentwickeln, ohne die programmierte Fachlogik zu verändern. Oftmals ist es ebenso schwierig den Architekturcode wiederzuverwenden. Diese Vermischung von Fachlogik und technischer Architektur verhindert in der Regel eine wirtschaftliche und effiziente Generierung. Moderne Softwarearchitekturen definieren deshalb eine Business-Objekt Schicht, die den fachlichen Code von der technischen Architektur trennt.

Business-Objekt Schicht

Der fachliche Code besteht aus den im Massenmodell modellierten fachlichen Operationen und sogenannten Standardoperationen, die durch die Transformation des Modells während der Generierung erzeugt werden. Beispiele für Standardoperationen sind Operationen zum Setzen und Abfragen von Attributen oder zum Erzeugen und Löschen von Objekten. Die fachlichen Operationen werden manuell implementiert, die Standardoperationen werden generiert. Der generierte Code der Business-Objekt Schicht ist der einzige Ort, an dem sich Fachlichkeit und technische Architektur berühren. Optimierungen und Änderungen der Architektur wirken sich dadurch nur auf die generierten Teile der Business-Objekt-Schicht aus. Der manuell programmierte Teil bleibt gegenüber Änderungen stabil.

Die Trennung von Fachlichkeit und technischer Architektur führt damit zu drei Bestandteilen einer Anwendung:

- Die technische Architektur kapselt die technischen Aufgaben, wie Client/Server-Kommunikation, Transaktionsverarbeitung, Objektaktivierung, Persistenz und Replikation, vollständig vom manuell programmierten Teil der Fachlogik. Dieser technische Architekturteil ist frei von spezifischer Fachlogik und in beliebigen Projekten wiederverwendbar.

- Die manuell programmierte Fachlogik kapselt die gesamte fachliche Komplexität der Anwendung.
- Die generierten Standardoperationen kapseln das Ineinandergreifen von Fachlichkeit und technischer Architektur.

Durch diese Dreiteilung erkennt man, wie die Generierungsfähigkeit von Architekturen zu steigern ist: Wenn es gelingt, wiederverwendbare und kombinierbare Elemente im Systemaufbau zu identifizieren und zu isolieren, besteht eine große Chance, die Erstellung dieser Elemente zu automatisieren und damit die Softwareerstellung wesentlich effizienter zu gestalten. Dieses Rezept lässt sich z. B. auch für Proxies, Skeletons und Datenbankzugriffsprogramme erfolgreich anwenden. In diesem Sinne ist das Generierungsverfahren der Firma Rösch Consulting architekturgetrieben: Erst die Nutzung objektorientierter Modellierungstechniken und die Weiterentwicklung von Softwarearchitekturen haben den wirtschaftlichen Einsatz von Generatoren möglich gemacht und erlauben in Zukunft eine wesentlich schnellere Erstellung von Software bei gleichzeitig höherer Qualität.

Einsatz des Generierungsverfahrens bei SIGNAL IDUNA

Die SIGNAL IDUNA Gruppe ist seit dem 1. Juli 1999 durch den Zusammenschluss der SIGNAL Versicherungen und IDUNA NOVA zu einem Gleichordnungskonzern am Markt präsent. Mit den Versicherungsdienstleistungen für Kranken-, Lebens- Sach- und Unfallversicherungen ist das Unternehmen in den wachstumsstarken Märkten gut positioniert. Daneben werden Finanzdienstleistungen angeboten, wie z. B. über die IDUNA Bausparkasse oder die CONRAD HINRICH DONNER BANK.

In den folgenden Abschnitten wird dargestellt, wie das oben beschriebene Generierungsverfahren für eine spezifische Client/Server-Architektur bei der SIGNAL IDUNA Gruppe angepasst wurde. Am Projekt sind bis zu zwölf Mitarbeiter beteiligt. Das Projekt ist im April 1999 gestartet und wird in seiner ersten Ausbaustufe im Juli 2000 abgeschlossen sein.

Entscheidung für den CORBA-Standard

Mit dem Vorhaben, die Entwicklung von verteilten Anwendungen zu unterstützen, wurde gleichzeitig ein Methodenwechsel zur Objektorientierung vollzogen. Als grundlegende Zielarchitektur wurde 1997 mit CORBA frühzeitig ein Industriestandard gewählt. Wesentliche Entscheidungskriterien für CORBA waren

- die Sprach- und Plattformunabhängigkeit der Architektur,
- die zugrundeliegenden Objektverteilungsmechanismen,
- die Einsetzbarkeit der Architektur für unterschiedliche verteilte Anwendungen im Bereich traditioneller Client/Server-Anwendungen,
- für Internet/E-Commerce-Anwendungen die Möglichkeiten der Integration von Transaktionsmonitoren über Konnektoren und nicht zuletzt
- die Ausrichtung der führenden Hersteller auf diese Technologie.

CORBA-Adaption mit Java-Clients und COBOL-Server-Framework unter IMS

Im Jahre 1997 existierte bei den SIGNAL Versicherungen keine verteilte Serverlandschaft. Aus Gründen der besseren Systemverwaltbarkeit und Skallierbarkeit wurde als Topologie für Client/Server-Anwendungen eine Drei-Schichten-Architektur mit Thin Clients unter Windows NT und dem Großrechner mit OS/390 als Anwendungs- und Datenserver definiert. Als objektorientierte Programmiersprache für Client und Server soll zukünftig standardmäßig Java eingesetzt werden. Zum Entwurfszeitpunkt der technischen Architektur war das nur für die Client-Seite möglich, da Java unter "Open Edition" in Verbindung mit den erforderlichen CORBA-Services, wie z. B. dem Transaktionsdienst, noch nicht von den Herstellern angeboten wurde.

Wir haben uns entschieden, zur Implementierung der Serverarchitektur COBOL-85 unter IMS einzusetzen. Wesentliche Rahmenbedingung für diese Entscheidung war die Einhaltung des Single-Source-Prinzips, was bedeutet, dass die gesamte Anwendung sowohl online in der beschriebenen Topologie als auch offline auf einem Außendienstlaptop läuft und nur einmal zu codieren ist. Die entwickelte Serverarchitektur kann sowohl für OS/390/IMS als auch für Windows NT/Windows 95 kompiliert werden. Abbildung 3 gibt eine Übersicht über den Online-Einsatz der Architektur.

Realisierung von CORBA-Elementen in der Architektur

Wie aus Abbildung 3 ersichtlich, sind in der Kommunikationsarchitektur CORBA Elemente

realisiert. Die Kommunikation der Objekte erfolgt über Stellvertreter (Proxies). Die Proxy-Schicht auf der Client-Seite stellt sicher, dass die Java-Anwendung frei von technischen Implementierungsdetails über den Standort des Servers während der Laufzeit ist. Im speziellen wird hier gesteuert, ob die Anwendung offline auf einem Laptop läuft oder ob der Java Client mit den Objekten auf dem Großrechner kommuniziert.

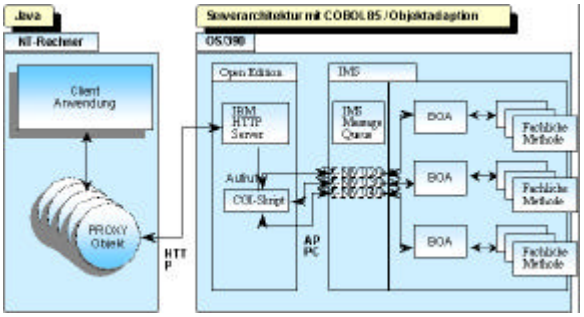

Abbildung 3 : COBOL Architektur

Die Serverarchitektur arbeitet ebenfalls nach den Regeln von CORBA. Schnittstellen und Implementierung sind klar voneinander getrennt. Der Aufruf von Objekten auf der Serverseite erfolgt über die als Skeletons bezeichneten Stellvertreterobjekte. Die öffentlichen Schnittstellen der Objekte werden im OOA-Modell in der Interface Definition Language (IDL) beschrieben. Dadurch wird die Unabhängigkeit von den eingesetzten Programmiersprachen erreicht.

Zwischen der Kommunikationsarchitektur und dem Skeleton befinden sich Object-Adapter (vgl. Abb.3), die zwischen der Welt der CORBA-Objekte und der Welt der Programmiersprachen-Implementierungen vermitteln. Aufgabe eines Objekt-Adapters ist es, CORBA-Objekte und ihre Objekt-Referenzen zu erzeugen, Anfragen von Clients an die jeweiligen Implementierungen weiterzuleiten sowie Aktivierungen und Deaktivierungen von CORBA-Objekten vorzunehmen – also den Lebenszyklus von Objekten abzubilden. Im vorliegenden Serverframework sind Objekt-Adapter implementiert. Die von außen kommenden Objekt-Anfragen, die als Nachricht über die IMS Message Queue ankommen, laufen über den Objekt Adapter und das Skeleton der gewünschten Operation zum Aufruf der Implementierung.

Merkmale der COBOL-Serverarchitektur

Die COBOL-Serverarchitektur unter IMS wurde in Zusammenarbeit mit Rösch Consulting konzipiert und realisiert. Sie stellt ein objektorientiertes Framework dar, in dem unter COBOL alle Elemente der Objektorientierung, auch Vererbung und Polymorphismus, realisiert wurden.
Wesentliches Merkmal der Serverarchitektur sind die Abbildung eines Transaktionskonzeptes mit Zwei-Phasen-Commit sowie eine integrierte Persistenzschicht. Abbildung 4 veranschaulicht das Transaktionskonzept.

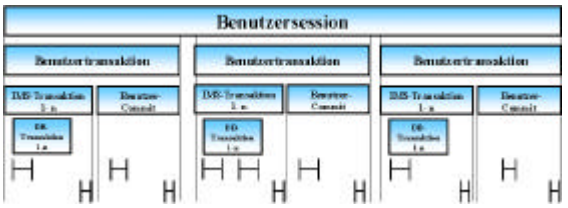

Abbildung 4 : Benutzer-Transaktionen

Die Bearbeitung von Objekten erfolgt stets innerhalb einer Benutzertransaktion. Diese wird auf der Client-Seite durch eine Folge von Dialogschritten abgebildet. Eine (logische) Benutzertransaktion kann mehrere physische Transaktionen umfassen. Jede physische (technische) Transaktion ist zwangsweise durch eine IMS-Transaktion abgesichert. Eine Benutzertransaktion muss explizit durch eine spezielle IMS-Transaktion (genannt Benutzer-Commit, vgl. Abb. 4) beendet werden. Am Ende einer technischen Transaktion werden alle bekannten und aktivierten Objekte in einem persistenten Objekt-Pool abgelegt und zu Beginn der nächsten technischen Transaktion erneut geladen. Das Rückschreiben der Objekte in die entsprechenden Datenbanken erfolgt erst beim Benutzer-Commit. Die Serverarchitektur bildet zu diesem Zweck ein optimistisches Zwei-Phasen-Commit Verfahren ab. Zum Zeitpunkt der Objektaktivierung erfolgt kein Sperren der Datenbankzeilen. Erst beim expliziten Commit wird für jedes geänderte Objekt geprüft, ob es nach der Aktivierung in der Datenbank modifiziert wurde.

Die gesamte Verwaltung der Persistenzschicht, d. h. das Lesen und Schreiben der Datenbanken, erfolgt über das Architekturframework und nicht durch den fachlichen Anwendungscode der Methodenprogramme. Abbildung 5 zeigt die wesentlichen Elemente des Serverframeworks.

Die beschriebene Architektur wurde 1998 erstmalig in einem Pilotprojekt eingesetzt. Das Serverframework sollte verwendet und durch die Anwendungsentwickler um den methodenspezifischen Code ergänzt werden. Dabei hat sich gezeigt, dass zum einen bei den Entwicklern genaue Kenntnisse bezüglich der Schnittstellen des Frameworks erforderlich waren und zum anderen für jede Klasse gleichartige Pro-

gramme erstellt werden mussten, beispielsweise Skeletons, Standardmethoden und Datenzugriffe für die Persistenzschicht. Das zugrundeliegende Mengengerüst legte den Einsatz eines automatisierten Verfahrens, also eines Generators, nahe.

"Rational Rose" für die Erstellung des OOA-Modells

Ausgangspunkt der Generierung ist ein mit dem Analysewerkzeug "Rational Rose" erstelltes Klassenmodell. Obwohl die Methoden mit COBOL-85 unter IMS realisiert werden, gibt es bei der Modellierung des Kassenmodells keine Einschränkungen hinsichtlich der UML-Elemente. Anforderungen an die Generierbarkeit des Modells bestehen hinsichtlich formaler Kriterien. So dürfen die Attribute nur IDL Datentypen enthalten, wobei die Liste der IDL Datentypen um die DB/2 Datentypen Date und timestamp erweitert wurden. Das hat den Vorteil, dass die Datenzugriffsschicht generiert werden kann. Ein generierbares Modell muss formal vollständig sein - so ist für jede Beziehung ein eindeutiger Rollenname innerhalb des Modells zu vergeben. Neben diesen Modellierungskonventionen wurden zusätzliche Eingaben zum Kassenmodell mittels Werkzeugerweiterungen über Rose-Skripte ermöglicht. Beispielsweise werden zu jeder Klasse die fachlichen Schlüssel - bestehend aus einer Untermenge der Attribute der Klasse - eingetragen.

Wegen der Längenbegrenzung von COBOL-Namen müssen die Massen und Methoden des Modells noch zusätzlich technische Namen erhalten. Die Namensvergabe wird maschinell durchgeführt. Durch die Hinterlegung der Namen im Rose-Modell ist eine wiederholte Generierung möglich.

Im Kassenmodell wird nur die eigentliche Fachlichkeit der Anwendung abgebildet. Darüber hinaus wurde eine Menge von Standardmethoden definiert, die nicht in das OOA-Modell eingetragen werden. Diese Standardmethoden werden bei der Generierung erzeugt. Hierbei handelt es sich z. B. um Objektmethoden zum Lesen und Setzen von Attributen, um Kassenmethoden (wie Create und Remove von Objekten) und Instanzmethoden (wie selectOne und selectAll).

Transformationen und Stanzformen

Die Modelltransformationen finden zum Zeitpunkt der Generierung statt. Aus dem OOA-Modell, das zuvor auf formale Konsistenz überprüft wurde, wird beim Generieren im ersten Schritt ein Design-Metamodell maschinell erzeugt. Dieses beinhaltet pro Klasse Einträge für jede Methode und jedes Attribut sowie die aufgelösten Assoziationen und Vererbungsbeziehungen. Zu diesem Zeitpunkt sind alle Informationen sprachunabhängig im Design-Metamodell abgelegt. Das Metamodell beinhaltet weiterhin die architekturspezifischen Standardmethoden. Damit wird ein umfangreicher Teil des Architekturdesigns maschinell abgebildet.

Im zweiten Schritt wird das Design auf das Java- und COBOL-Metamodell transformiert. Dabei findet die Umsetzung der IDL Datentypen auf die sprachspezifischen Datentypen gemäß dem CORBA Mapping für Java und COBOL statt. Zuletzt werden der Java und COBOL-Quellcode auf Basis der vorgefertigten Stanzformen erzeugt.

Ein weiteres wesentliches Merkmal der Stanzformen ist, dass sie weitere architekturspezifische Designregeln und unternehmensweite Konventionen umsetzen, wie z. B. Programmierrichtlinien und Namenskonventionen.

Durch die Generierung über Stanzformen wird das Zusammenwirken der generierten Standardmethoden und der fachlichen Methodenrümpfe sichergestellt. Die für eine fachliche Methode erforderlichen Objektdaten werden vom Serverframework über eine Schnittstelle bereitgestellt.

Für die zu generierenden COBOL-Programme wurden ca. 50 Stanzformen und für die Java-Proxies ca. 25 Stanzformen entwickelt. Damit kann jedes beliebige UML Klassenmodell generiert werden, das den Anforderungen an das Generierungsverfahren genügt.

Zusammenfassend kann festgestellt werden, dass das gesamte Design der Anwendung

- für die technischen Architekturelemente - wie Transaktionsverarbeitung, Kommunikation, Persistenz - innerhalb der festen, wiederverwendbaren Architekturprogramme implementiert ist oder aber
- als Standardoperationen durch den Generator erzeugt wird.

Fazit

Architekturgetriebene Generierung macht dort weiter, wo die Möglichkeiten objektorientierter Technologie aufhören. Es ist möglich fachliche und technische Teile voneinander zu trennen. Damit können beide Teile parallel entwickelt werden. Der technische Teil ist unabhängig von der Fachlogik wiederverwendbar.

Literatur

[Edw99] J. Edwards, D. Harkey, R. Orfali, Client/Server Survival Guide, John Wiley & Sons,1999

[Hub99] S. Huber, H. Seidl, Architekturzentriertes Entwickeln mit der UML, in OBJEKTspektrum 4/99, S. 20 [OMG-UML] OMG Unified Modeling Language Standard, Version 1.3

[Roe96] M. Rösch, Generierungstechnik für die Implementierung von Business-Objekten, in OBJEKTspektrum 6/99, S. 28 9

Über die Autoren

Peter Metzen und Constantin Szallies arbeiten als IT Berater.

Dipl.-Math. Bemd Paffenholz (E-Mail: bemd.paffenholz@signal.de) arbeitet in der Abteilung Anwendungsarchitektur der SIGNAL IDUNA Gruppe Versicherungen und Finanzen. Sein Schwerpunkt ist die Entwicklung und Einführung von technischen Architekturen.

Dipl.Inform. Franz Töpler (E-Mail: franz.toepier@signal.de) ist in der Abteilung Anwendungsarchitekturen der SIGNAL IDUNA Gruppe Versicherungen und Finanzen tätig. Er ist Leiter des Projekts "Technische Architektur und Generator für die neue IV".