Codegeneratoren: Domänenspezifische Automatisierung in der Praxis industrieller Softwareentwicklung

Dr. Martin Jung, develop group, 14. November 2008

Zusammenfassung:

Die Technologien, die die Grundlage zur automatischen Generierung von Programmcode bilden, werden stetig verbessert und sind mittlerweile auf einem Stand angekommen, der einen gewinnbringenden Einsatz im Rahmen industrieller Projekte verspricht.

Der Blick in den tatsächlichen Projektalltag zeigt jedoch, dass Modellierung nur vereinzelt betrieben wird und meist nicht auf die Fachdomäne zugeschnitten ist. Generatoren werden – wenn überhaupt – selten durchgängig eingesetzt, und in fast allen Fällen wird das Generat von Hand weiterverarbeitet.

Der Artikel zeigt an kleinen Beispielen aus der Projektpraxis, wie Generatoren erfolgreich eingesetzt werden können. Die oft beobachteten Schwierigkeiten beim Einsatz von Generatoren liegen dabei eher in der für ihre Einführung ausgewählten Strategie, nicht an tatsächlichen technischen Unzulänglichkeiten.

1 Ausgangslage

Bei der Entwicklung großer Softwaresysteme treten häufig wiederkehrende Tätigkeiten auf, deren manuelle Durchführung fehlerträchtig ist. In bestimmten Bereichen, wie beispielsweise der Entwicklung von Persistenzschichten, von Benutzeroberflächen oder Kommunikationsinfrastrukturen häufen sich solche Aufgaben, weswegen man dort verstärkt über die Automatisierung durch Generatoren nachdenkt. Auch hinsichtlich komplexerer Datenstrukturen, die verarbeitet oder visualisiert werden sollen, bietet sich ein automatisiertes Vorgehen an, mit dem die Datenstrukturen für die Darstellung konvertiert oder Bildschirmmasken zu ihrer Bearbeitung generiert werden.

In industriellen Projekten findet sich allerdings erstaunlich selten ein generativer Entwicklungsansatz, was in deutlichem Kontrast zu der Anzahl positiver Erfahrungsberichte steht, die man in den einschlägigen Zeitschriften oder auf Fachtagungen (z. B. [1]) liest bzw. hört. Gründe für diese Diskrepanz liegen

- in einer Unsicherheit hinsichtlich der Risiken und Einsparpotentiale,
- in der Unkenntnis der tatsächlichen Möglichkeiten, die Codegeneratoren bieten,
- oder in schlechten Erfahrungen mit dem Einsatz eines Generators.

Mangel an technologischen Möglichkeiten, die die Generatoren mitbringen, kann als Ursache ausgeschlossen werden – positive Erfahrungsberichte (z.B. [2]) zeigen, dass die Generatoren mittlerweile genügend leistungsfähig sind. Zum einen gibt es hochspezialisierte Generatoren, die für ihre Anwendungsdomäne elegante, hoch abstrahierende Konstrukte anbieten, hinter denen der automatisch erzeugte Programmcode so gut wie vollständig verschwindet. Zum anderen gibt es Generatorensysteme, die standardisierte Eingabesprachen verwenden und konfigurierbare Schablonenmechanismen zur Ausgabe verwenden. Aufgrund der Flexibilität solcher Systeme können sie mit relativ wenig Aufwand für ein ganz spezielles Projektumfeld angepasst und gepflegt werden. Für beide Arten von Generatoren wird im Folgenden ein Beispiel aus Projekten, die von der develop group im Kundenauftrag durchgeführt werden, demonstriert.

Es verbleiben also die beiden anderen Faktoren: Unsicherheit hinsichtlich der Einsparpotentiale bzw. schlechte Erfahrungen beim ersten Einsatz. Auf beide wird nach der Vorstellung der Beispiele genauer eingegangen.

2 Arten von Generatoren

Wie bereits angedeutet, werden zwei Arten von Generatoren unterschieden:

Spezialisierte Generatoren, die nur für einen bestimmten Teil des zu entwickelnden Zielsystems verwendet werden, diesen allerdings vollständig abdecken. Die Eingabe für den Generator ist typischerweise stark abstrahiert, so dass sie mit wenig zusätzlichem Aufwand von einem Domänenexperten erstellt werden kann. Bestehende Entwurfsdokumente sind zumeist eher technischer Natur und daher nur bedingt verwendbar. Für den vom Generator erzeugten Code ist keine bzw. kaum manuelle Überarbeitung mehr notwendig. An den Grenzen des vom Generator erzeugten Programmteils müssen Vorkehrungen getroffen werden, damit die Annahmen des Generators hinsichtlich der Umgebung nicht gebrochen werden und der generierte Code nicht falsch aufgerufen wird.

Vielzweck-Generatorensysteme, die eine mächtige Eingabesprache und variierbare Ausgabeformate beherrschen. Typischerweise wird bei der Verwendung eine Untermenge der Eingabesprache gewählt, und die Ausgabeschablonen werden dem Projektumfeld bzw. dem Einsatzgebiet des Generators angepasst, so dass ein projektspezifisches Spezialwerkzeug resultiert. Beim Einsatz solcher Vielzweck-Generatoren, die von sich aus keine Annahmen über das Projektumfeld und ihre Ein- bzw. Ausgabedaten mitbringen, muss die Strategie zur Einführung und zum Einsatz im Projekt detaillierter festgelegt werden als bei den spezialisierten Generatoren. Von Vorteil ist, dass bestehende, bereits formalisierte Architektur- und Entwurfsmodelle mit geringem Überarbeitungsaufwand als Eingabe für den Generator dienen können. Einen schematischen Überblick über Vielzweckgeneratoren gibt Abbildung 1:

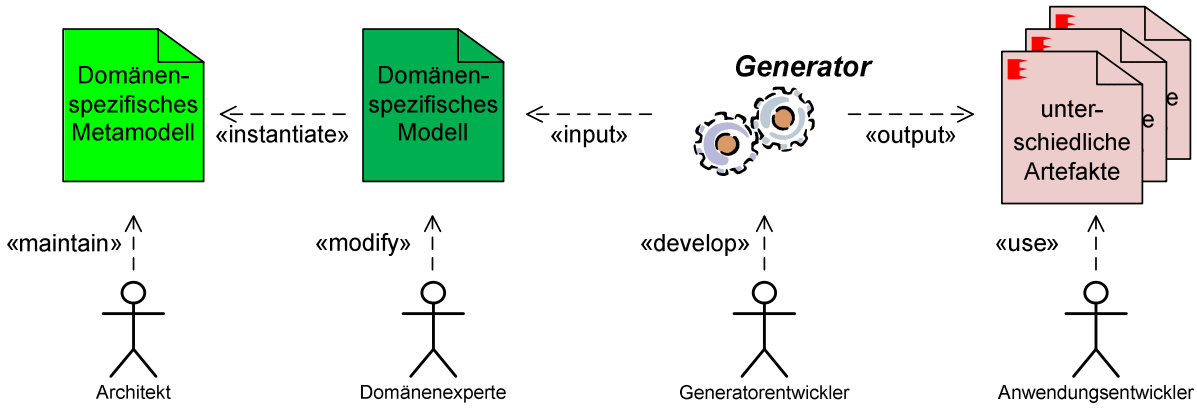

Abbildung 1: Überblick über Aktoren und Elemente beim Einsatz eines Vielzweckgenerators

2.1 Beispiel 1: Persistenzschichtgenerierung in der Medizintechnik

Für die Persistenz werden im Umfeld objektorientierter Software objektrelationale Abbildungen verwendet, die es erlauben, die Datenstrukturen der objektorientierten Software in relationalen Datenbanken abzuspeichern. Zur Beschreibung der Abbildung verwendet man so genannte Mappings, die auf die Strukturen der Software angepasst sein müssen und exakt beschreiben, wie die Inhalte der Objekte auf Relationen in der Datenbank abgebildet werden. Die develop group leistet bei der kundenspezifischen Entwicklung eines Backend-Systems zur Verwaltung medizintechnischer Bilddaten Unterstützung. Dort wird – neben dem Speichern der Bilddaten selbst – Metainformation der Bilder gesammelt,

verarbeitet und abgespeichert. Die Persistierung der Metadaten folgt dem in der Java-Welt etablierten JDO-Standard [3], wobei für jede zu persistierende Java-Klasse ein solches Mapping angegeben wird. Der verwendete Codegenerator (KODO von BEA [4]) modifiziert dann mit Hilfe der Mappings den Bytecode der jeweiligen Java-Klasse, indem zusätzliche Funktionalität injiziert wird, die für die Persistierung der Klassen-Daten beim Aufruf von set-Methoden bzw. für die Aktualisierung der Daten aus der Datenbank bei get-Methoden sorgt.

Dies ist ein Beispiel für einen relativ schmalen Ausschnitt der Softwareentwicklung, der durch den Codegenerator vollständig erledigt wird. Allerdings existieren nur wenige Möglichkeiten zur projektspezifischen Anpassung, die Eingabedaten des Generators müssen genau bedient werden. Das Vorgehen des Generators ist objektbasiert, es wird also jeweils ein Objekt gespeichert bzw. geladen. Im angesprochenen Projekt werden größere, semantisch zusammenhängende Objektbäume jeweils gemeinsam erzeugt, aktualisiert oder gelöscht. Also muss bei Änderung eines Objektes die so genannte „persistente Hülle“ – die komplette semantische Einheit – eines Objektes ermittelt und bearbeitet werden. Dieses Vorgehen war in der eingesetzten Version des Persistenzgenerators nicht vorgesehen, daher waren die Eingabedaten des Generators nur sehr mühsam zu erstellen und in der Wartung sehr aufwändig.

Im Endeffekt wurden dadurch die Schwierigkeiten, die Eingabedaten von Hand zu pflegen, so groß, dass das Verfahren von den Entwicklern abgelehnt wurde. Um dennoch die Vorteile des Generators nutzen zu können, wurde ein weiterer generativer Schritt vorgeschaltet, der aus einem Domänenmodell die Java-Quellcodeklassen des Datenmodells und die darauf abgestimmten Eingabedaten für den Persistenzgenerator herstellt. Der zusätzliche Generator sorgt dafür, dass beide Seiten der generierten Persistenzschicht,

- der Programmcode des logischen Datenmodells und
- die Eingabedaten für den Persistenzgenerator,

aus einer Quelle kommen.

Verallgemeinernd kann man für den Einsatz spezialisierter Generatoren plädieren, wenn sie exakt oder nur mit geringer Abweichung in das Projektszenario passen. Ansonsten empfiehlt sich der Einsatz eines Vielzweckgenerators wie im nächsten Beispiel beschrieben.

2.2 Beispiel 2: Komplexe Stammdaten und deren automatische Bearbeitung im Rahmen der KFZ-Steuergerätediagnostik

In einem anderen Projekt der develop group ist ein Altsystem zur Diagnose von KFZ-Steuergeräten (Electronic Control Units, **ECU**s) auf Grund von zu aufwändiger Wartung zu erneuern, dabei eine moderne Architektur aufzubauen und in erster Iteration lediglich die bestehende Funktionalität weiterhin zu gewährleisten. Die Schwierigkeit in diesem Projekt besteht in der hohen Zahl unterschiedlicher ECU-Varianten, die sich in Adressierung, Funktionsumfang und Interaktionsmodell unterscheiden. Die Anwendung ist ein relativ einfaches Dialogsystem zur Diagnose eingebetteter Steuergeräte in Kraftfahrzeugen. Die Rohdaten für die Diagnose werden von Domänenexperten in einem langwierigen und fehleranfälligen Prozess ermittelt, im Altsystem als Excel-Dateien abgespeichert und dann in das Diagnosesystem importiert. Der Inhalt der Excel-Dateien ist allerdings nicht automatisiert prüfbar. Der Import geschieht deswegen von Hand; fehlerhafte Daten, widersprüchliche Angaben oder Unvollständigkeiten in den Excel-Dokumenten werden dabei vom Datenbankexperten in Zusammenarbeit mit den ECU-Experten behoben.

Bei der Erneuerung sollte dieser sowohl fehlerträchtige als auch langwierige Prozess der Datenbeschaffung, -einbringung und -verarbeitung durch Automatisierung unterstützt werden. Den Datenermittlern soll es dabei weiterhin möglich sein, weitgehend unabhängig von den Softwareentwicklern zu arbeiten. Zunächst wurde mit allen beteiligten Parteien ein Domänenmodell entwickelt, um einen gemeinsamen Sprachgebrauch und eine gemeinsame Auffassung der Fachdomäne zu erreichen. In diesem Modell sind nicht nur die Datenstrukturen hinterlegt, sondern auch Regeln, die die Inhalte einer Modellinstanz kontrollieren und deren Auswertung inkorrekte Modelle identifizierbar macht.

Die Modelle der Steuergeräte werden als Eingabe für den Generator verwendet, der alle weiteren für das Produkt benötigten Artefakte daraus herstellt:

- Programmcode für die Diagnose,
- SQL-Code, um die Datenbank mit Parameterinformationen für die Diagnose zu füllen, und
- Dokumentation sowohl für die Entwicklung (API-Doku) als auch für die Endbenutzer (ECU- und Funktionslisten).

Basierend auf einem durch die Architekten erstellten ECU-Metamodell, das für alle drei weiteren Rollen der Entwicklung (vgl. Abbildung 1) gleichermaßen geeignet ist, werden Editoren und der eigentliche Generator implementiert. Die Datenermittler können mit Hilfe von Editoren, die die Einhaltung der Regeln des Modells erzwingen, ihre ECU-Datenmodelle erfassen. Die Editoren sind auf Basis des Eclipse Graphical Modeling Frameworks [5] implementiert. Ihre Ausgabe ist ein domänenspezifisches Modell, also in diesem Fall ein ECU-Modell. Dieses domänenspezifische Modell bildet wiederum die Eingabe für den Generator, der auf Basis des openArchitectureWare-Projekts [6] entwickelt ist. Er erzeugt folgende konkrete Ausgabeartefakte:

- SQL-Datendefinitionen für die Datenbank,
- Quelldateien der Programmiersprache zur Erweiterung der manuell erstellten Kernanwendung,
- HTML-Seiten zur Dokumentation der erfassten Daten und
- Testtreiber, die es ermöglichen, die Daten und Klassen der neu generierten ECU-Beschreibung einem automatisierten Test zu unterziehen.

Um die steuergerätespezifischen Teile des Programms, die von diesem Generator erzeugt werden, reibungslos in das restliche Programm einbinden zu können, wurden zusätzliche Vorkehrungen getroffen. Die Anwendung selbst ist modellgetrieben entwickelt, und für die Teile der Anwendung, die mit dem Generator erstellt werden, wird eine explizite Kopplung über Entwurfsmuster (hauptsächlich *Factory* und *Command*) bereits im Modell verankert. Das in Abbildung 1 gezeigte allgemeine Schema lässt sich also für das Projekt speziell anpassen: Wie in Abbildung 2 zu sehen, wird ein für alle verbindliches ECU-Metamodell geschaffen und vom Architekten gepflegt. Der Architekt erstellt zudem das Modell der gesamten Anwendung. Die Entwicklung ist in drei Teams aufgeteilt:

- ECU-Experten, die die Datenanalyse betreiben und ihre Ergebnisse in Form von ECU-Modellen in die Entwicklung einspeisen,
- Generatorenentwickler, die den Generator an sich sowie Werkzeuge für den Umgang mit dessen Eingabe- und Ausgabeartefakten erstellt,
- Anwendungsentwickler, die den generischen Teile der Anwendung implementieren.

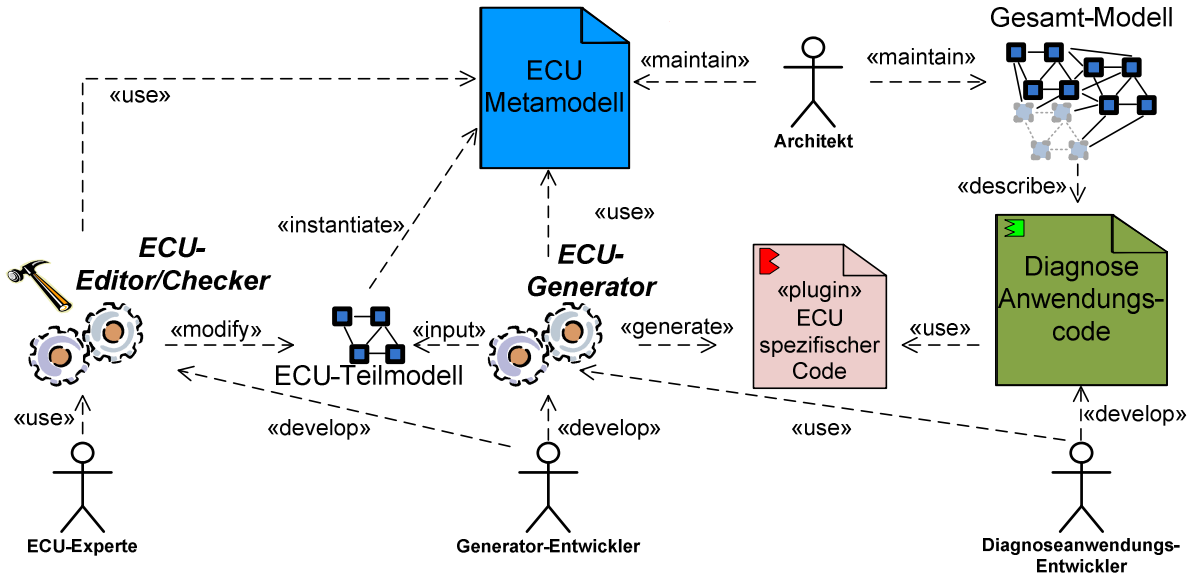

Abbildung 2: Überblick über den ECU-Codegenerator

Das Interaktionsmodell der Diagnosesoftware mit dem Steuergerät ist denkbar einfach. Eine Anfrage wird zum Steuergerät geschickt, und aus der Antwort des Steuergerätes müssen die zur Weiterverarbeitung relevanten Daten extrahiert werden. Die Anfragedaten sowie weitere Parameterinformationen zur Generierung entsprechender Sende- und Empfangsmethoden sind Inhalt der ECU-Teilmodelle. Falls ein Steuergerät vom üblichen Zustandsmodell abweicht, kann seine Zustandsmaschine ebenfalls im Modell hinterlegt werden. Durch die Kopplung über Entwurfsmuster kann die Diagnoseinteraktion in der Anwendung selbst generisch programmiert werden, die genannten Teile der Programmlogik werden aus dem Modell generiert. Tatsächlich wird Programmcode

- zum Laden der Steuergerätespezifika aus der Datenbank und
- zum Erzeugen entsprechender ECU-spezifischer Diagnoseklassen

nach dem *Factory*-Pattern erzeugt. Nach dem *Command*-Pattern werden die Steuergerätefunktionen umgesetzt, sowie gegebenenfalls Zustandsmaschinen für die Steuergeräte nach dem *State*-Pattern. Für alle zu erzeugenden Artefakte werden Schablonen gefertigt, die deren Struktur festlegen. Der Generator fügt dann nachträglich Parameter aus den jeweiligen Modellen ein.

Im Projekt werden die Möglichkeiten des Generatorenframeworks von *openArchitectureWare* weitgehend genutzt, insbesondere die Möglichkeiten von „**`check`**“ und „**`xpand`**“. „**`check`**“ ist ein Mechanismus, den das Generatorenframework anbietet, um die Inhalte des Eingabemodells nochmals zu prüfen, bevor sie vom eigentlichen Generator verarbeitet werden. Diese Möglichkeit sollte auch verwendet werden, um die Schablonen für den Generator von Fehlerbehandlungen frei und damit übersichtlicher zu halten. Das Template-Subsystem „**`xpand`**“ schließlich wird verwendet, um alle Ausgabeartefakte zu erzeugen:

- die Datendefinition (SQL),
- die Programmlogik (java),
- die Testgerüste (java, ant) und
- die Dokumentation (html).

3 Einbettung des Generators in die Entwicklung

Generatoren werden dazu verwendet, einzelne Schritte des Entwicklungsprozesses zu automatisieren. Solange die Schritte ohne Generator manuell erledigt werden, können Ein- und Ausgabe formlos sein und eventuell sogar Unvollständigkeiten bzw. Widersprüche enthalten. Ein Projektmitarbeiter hat sozusagen hervorragende Fehlertoleranzmechanismen, indem er bei den Produzenten unklarer Eingabeartefakte nachfragt und die Eingabe oder eben seinen eigenen Handlungsalgorithmus überarbeitet. Beispielsweise könnte folgendes Szenario auftreten: Im Programmcode für die ECU-Diagnose fehlt die Information, mit welchem Protokoll eine bestimmte ECU angesprochen werden muss; also geht man auf den verantwortlichen ECU-Experten zu und fragt nach. Sofern also allgemein Bedarf an Klärung auftritt, kann dieser Bedarf gedeckt werden – ein Merkmal, das manuell erzeugte Artefakte den maschinell erzeugten voraus haben. Abbildung 3 zeigt dieses Prinzip bei der manuellen Entwicklung: Der Verantwortliche (B) einer bestimmten Tätigkeit kann sich bei seinem Vorgänger (A) Klarheit über seine Eingabe verschaffen und die Tätigkeit dann ausführen. Für Nachfolger im Prozess (C) steht der Bearbeiter jederzeit für Erklärungen zur Verfügung.

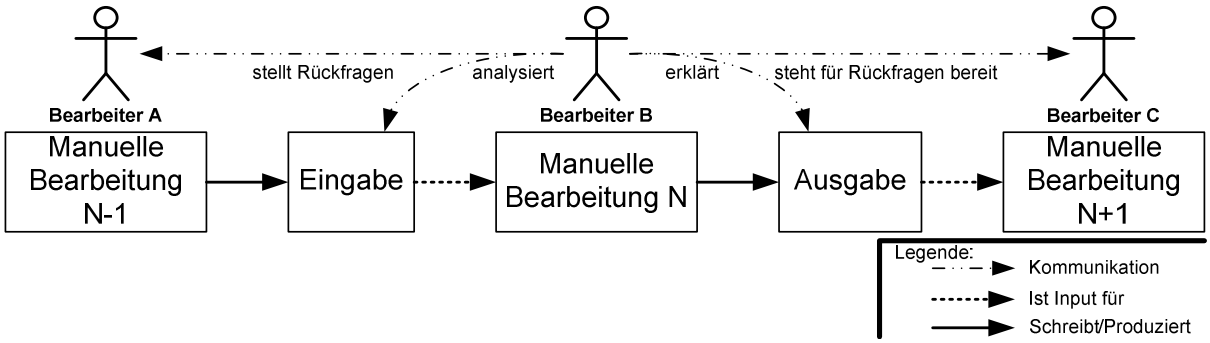

Abbildung 3: Schema eines manuellen Entwicklungsschritts

Ein Generator hingegen arbeitet nur auf Basis exakter Eingaben, bei fehlerhaften Eingaben wird er nicht richtig funktionieren, und falls die Ausgabe des Generators unverständlich ist, so kann man bei ihm keine zusätzliche Information bekommen. Insbesondere die Kontrolle der Ausgabeartefakte (vgl. Abbildung 4) kann manuell nicht geleistet werden, ohne den Vorteil der Geschwindigkeit des Generators zu gefährden. Fehlt im ECU-Datenmodell, wie oben angesprochen, eine Protokollinformation und wird der ECU-Diagnosecode generiert, so tritt bei den Testläufen ein Fehler auf. Die Ursachenanalyse wird schwierig und teuer, da mehr Zeit zwischen Fehlerursache und -erkennung vergeht und zusätzlich mehr Personen gebunden werden (ECU-Experten und Generatorenentwickler). Im konkreten Projekt (Beispiel 2, Abschnitt 2.2) wurde für solche Situationen eine Umgebung geschaffen, die zu jedem generierten Artefakt eine sofortige Navigation zu den manuell erstellten, die Fehlerursache enthaltenden Artefakten erlaubt. Zusammen mit einem Debugger, der die Analyse des generierten Codes zur Ausführungszeit mit Navigation in die ursprünglichen Artefakte ermöglicht, wurden damit gute Erfahrungen gemacht. Ohne Werkzeugunterstützung muss man durch das Generat (das nicht für manuelle Bearbeitung gedacht ist) hindurch bis zum letzten manuell erstellten Artefakt navigieren, und dessen Ersteller befragen. Beim Einsatz eines Generators müssen also ausreichende Werkzeuge zur Prüfung von Eingaben und Ausgaben des Generators geschaffen werden.

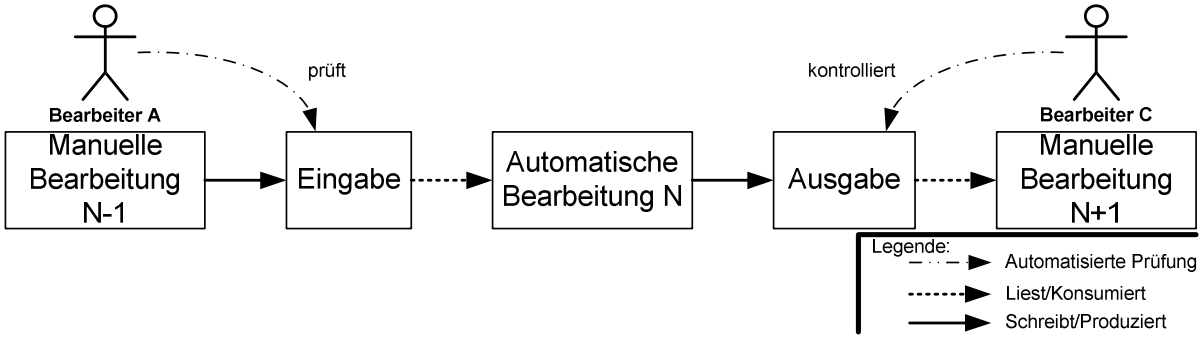

Abbildung 4: Schema eines automatisierten Entwicklungsschritts

Sowohl die Eingaben als auch die Ausgaben des Generators müssen demnach prüfbar sein, und zwar weitgehend automatisiert, um möglichst effizient zu bleiben. Das bedeutet aber im Projektgeschäft auch, dass für jeden Schritt, der automatisiert werden soll,

- der Generator selbst,
- sein Eingabeformat zusammen mit entsprechenden Prüfroutinen sowie
- sein Ausgabeformat zusammen mit Tests für die Korrektheit der Ausgabeartefakte

erstellt und gepflegt werden müssen. Dabei muss berücksichtigt werden, dass der Vorteil eines Generators nur realisiert werden kann, wenn die zugehörigen Prüfroutinen ebenfalls automatisch ablaufen. Gerade dieser zusätzliche Aufwand für generierte Artefakte stellt eine weitere Herausforderung bei der Einführung eines Generators in Projekten dar.

Ist der Generator mit solchen zusätzlichen Prüfmechanismen ausgestattet, kann er mit wenig Risiko in das laufende Projekt eingebunden werden. Lediglich an der Grenze zwischen generierten und manuell erstellten Artefakten sind noch Vorkehrungen zu treffen, um Probleme im Entwicklungsprozess, speziell bei der Versionierung, zu vermeiden. Es existieren verschiedene Ansätze, wie mit der Vermischung von generiertem und manuell erzeugtem Programmcode umgegangen werden soll. Quintessenz von vergleichenden Untersuchungen hierzu ist es, beide Artefaktarten strikt zu trennen und nicht z.B. mit Hilfe von so genannten „Protected Regions“ zu vermischen (vgl. dazu auch [7]). Beherzigt man diese Trennung, ist die restliche Anwendung von der Architektur her so umzustellen, dass die generierten Teile über eine definierte Schnittstelle eingebunden werden. Dazu dient neben einer Abhängigkeitsanalyse zwischen generiertem und manuell erstelltem Code der Einsatz von bewährten Entwurfsmustern, wie z.B. der im Rahmen der ECU-Diagnoseanwendung genannten. Die Abhängigkeitsanalyse ist unter zwei Aspekten zu führen:

- Welche Teile des generierten Codes werden vom manuellen Programmcode aufgerufen?
- Welche Teile des manuellen Programmcodes darf das Generat verwenden?

Neben dem üblichen Verfahren, manuell erstellten Code abhängig vom Generat zu modellieren, kann auch eine Abhängigkeit des generierten Codes von anderem, manuell erstellten Code unter Umständen sinnvoll sein. Beispielsweise ist dies für kleinere Toolkit-Funktionen oder sehr veränderliche Programmabschnitte effizienter als eine vollständige Generierung. Solange man an der Stelle auf die Zyklenfreiheit der Abhängigkeiten achtet, ist auch das kein Problem.

3.1 Analyse des Projekts vor Einführung des Generators

Bevor in ein laufendes Projekt ein Generator eingeführt wird, muss geprüft werden, ob sich die Einführung lohnt. Faktoren für eine solche Prüfung sind in Abbildung 5 zusammengefasst.

Zunächst scheint angesichts des Overheads ein Einsatz umso vielversprechender, je häufiger der Generator angewendet wird. Zwei Faktoren beeinflussen die Häufigkeit:

- die Änderungsfrequenz der Eingaben und
- die Anzahl der unterschiedlichen Eingabeartefakte („Einsatzbreite“) im Projekt.

Die reine Zeitersparnis durch den Generator ist dabei das häufigste Argument, stärker wiegt jedoch meist das Potenzial, bei fehleranfälligen, sich wiederholenden Tätigkeiten Fehler zu vermeiden. Einen gewissen Reifegrad vorausgesetzt, wird der Generator seine Aufgabe fehlerfrei erledigen und aufwändige Fehlerkorrekturzyklen vermeiden. Ändert sich im Gegensatz dazu das Ein/Ausgabeschema des Generators häufig (d.h. das Domänenmodell, auf dem der Generator beruht, ist nicht für längere Zeit stabil zu halten), so muss mit viel Aufwand für die Pflege des Generators gerechnet werden.

Geringe Fehleranfälligkeit	Hohe Fehleranfälligkeit
Schlechte Werkzeugunterstützung	Gute Editoren für Ein/Ausgabeartefakte
Eingabeartefakte informal	Eingabeartefakte formalisiert
Seltene Ausführung	Häufige Ausführung
Domänenmodell schnelllebig	Domänenmodell stabil
Einsatz nicht sinnvoll	**Einsatz empfehlenswert**

Abbildung 5: Entscheidungsfaktoren für den Generatoreinsatz (Gewichtung steigt von oben nach unten)

Schließlich sind zwei weitere Überlegungen bei der Abwägung für oder wider den Einsatz anzustellen:

- die Frage nach der Formalisierung von Ein- und Ausgabedaten des Generators, sowie
- die Frage nach der weiteren Integration des Generators in das Projekt- bzw. Werkzeugumfeld.

Sind die Artefakte, die der Generator als Eingabe verwendet, bislang schwach spezifiziert und lediglich auf Grund guter Kommunikation der Teammitglieder untereinander verständlich, muss viel Aufwand für die Formalisierung der Eingaben eingeplant werden. Im ECU-Diagnoseprojekt waren die Daten vor Einführung des Generators über mehrere heterogene Datenquellen verstreut (u.a. Excel-Dokumente, SQL-Server Einträge, direkt im Programmcode). Da der Generator hier im Rahmen seiner Eingabe-Prüfroutinen einen Abgleich vornehmen muss, entsteht bereits an dieser Stelle ein Vorteil.

Hinsichtlich der Ausgaben des Generators muss man bedenken, dass für sie ein gemeinsames Verständnis der Inhalte innerhalb des Teams erforderlich ist. Der Generator muss also in seine Ausgabe genug Kommentare bzw. ausreichend Information einbringen, so dass die Ausgabeartefakte von allen Projektmitarbeitern, welche die automatisch generierten Artefakte verwenden, leicht verstanden und weiterverarbeitet werden können. Hier kann eine automatisch mit jedem Generatorenlauf erstellte Dokumentation sehr hilfreich sein.

Hinsichtlich der Werkzeugunterstützung muss in erster Linie darauf geachtet werden, dass bei der Erstellung der Generator-Eingabe keine unnötigen technischen Details für den Generator mit gepflegt werden müssen. Es ist wesentlich sinnvoller, an dieser Stelle eine eigene domänenspezifische Sprache zu entwickeln und zu pflegen, als vorgefertigte Sprachen wie die UML in ihrer vollen Breite zu nutzen. Für die Standardsprachen spricht zwar die gute Werkzeugunterstützung, allerdings ist deren Komplexität nicht zu unterschätzen. Die Modellierung kann daher wesentlich komplizierter ausfallen, als sie sein müsste. Mit Hilfe des auch in der ECU-Diagnose verwendeten GMF-Projektes [5] lässt sich eine rudimentäre Werkzeugunterstützung in Form von graphischen Editoren für Modelle direkt aus dem Metamodell einer domänenspezifischen Sprache generieren. So erhält man exakt auf die

Problemdomäne zugeschnittene Editoren. Die Weiterverarbeitung der Modelle (Prüfung der Eingaben, Generierung, etc.) ist mit dem Generatorensystem *openArchitectureWare* problemlos machbar.

3.2 Identifikation des Generatorschritts

Fasst man die Softwareentwicklung als eine Reihe von Aktivitäten auf, so ist ein Generator für mehrere, direkt aufeinander folgende Aktivitäten dieser Art einsetzbar. Den Teil der Entwicklung, den der Generator übernimmt (z.B. Generierung von Schnittstellen-, Infrastruktur- und Datenbankcode) nennt man **Generatorschritt**.

Unter Berücksichtigung der genannten Einflussfaktoren auf den Entwicklungsaufwand eines Generators kann im jeweiligen Projekt ein lohnender Generatorschritt gesucht werden. Dabei sollte ein Entwicklungsschritt gewählt werden, dessen Domäne gut verstanden ist und stabil bleibt (vgl. Kriterien in Abbildung 5). Eine weitere Überlegung bei der Identifikation richtet sich auf den Umbau der Anwendungsarchitektur. Die erwartete Ausgabe des Generators sollte (z.B. mit Hilfe von Entwurfsmustern) bereits in der Architektur des Gesamtsystems gekapselt werden. Dazu ist ein Umbau der Architektur und ein Refactoring der Anwendung (vgl. auch 3.3 im Anschluss) erforderlich. Es ist insbesondere darauf zu achten, dass Ein- und Ausgaben des Generatorschritts in Form von Schnittstellen vereinbart werden. So ist gewährleistet, dass die manuelle Entwicklung nicht von den generierten Implementierungen abhängt und keine Reibungsverluste im weiteren Entwicklungsprozess drohen.

Ist der Generatorschritt vorab festgelegt, weil ein Spezialgenerator eingesetzt wird, so müssen die entsprechenden Gegebenheiten im Projektumfeld geschaffen werden. Wie an Hand des Projekts aus der Medizintechnik beschrieben, kann dies durch vorgeschalteten Einsatz eines Vielzweck-Generators erreicht werden. In jedem Fall ist darauf zu achten, irrelevante Informationen aus den Eingabemodellen auszublenden, um den eigentlichen Generator leicht wartbar zu halten.

3.3 Umbau der Entwicklung

Bei der Einführung eines Generators in ein laufendes Projekt ist schließlich darauf zu achten, das im Team vorhandene domänen- und anwendungsspezifische Wissen der Entwickler zu nutzen. Hierbei muss oft behutsam vorgegangen werden, um bestehende Teamstrukturen durch den geplanten Einsatz nicht zu stören. Auch darf der Generator nicht zu früh zum Einsatz gebracht werden, um die Akzeptanz nicht durch noch schwierige oder fehlerträchtige Verwendungsversuche zu gefährden.

Zunächst sollten die Ein- und Ausgabedokumente des Generatorschritts im Projekt überarbeitet werden. Struktur und Inhalt der Dokumente müssen dahingehend geändert werden, dass die Eingabedokumente dem Parser des Generators zugänglich sind, die Ausgabedokumente von den Projektmitarbeitern leicht weiter bearbeitet werden können und alle notwendigen Inhalte vorweisen. Die neuen Dokumentenstrukturen werden zunächst manuell verwendet. Dabei kann von den Teammitgliedern geprüft werden, ob alle relevanten Informationen erfasst werden können und ob die resultierenden Artefakte verständlich sind. In der Anfangsphase dieses Praxistests sollte bei der Erstellung der Eingabeartefakte durch die Domänenexperten jeweils ein Generatorenentwickler einbezogen werden, und bei der Verwendung der Ausgabeartefakte durch die Anwendungsentwickler sollte ebenfalls Unterstützung durch das Generatorenteam geleistet werden.

Die Umstellung auf die neuen Artefakte sollte zusammen mit einer geeigneten Werkzeugunterstützung erleichtert werden, die den Domänenexperten hilft, die Eingaben darzustellen, und den Anwendungsentwicklern hilft, die Ausgaben einzubinden und zu testen. Das vermindert die zu erwartende Skepsis im Team und kann (durch neue Features der Werkzeuge) bereits zu einer Effizienzsteigerung führen.

Ist im Projekt ein modellgetriebener Entwicklungsprozess (wie z.B. in [8] beschrieben) etabliert, fällt dieser Schritt besonders leicht. Typischerweise liegt in solchen Entwicklungsprozessen das Architekturmodell (Software-Architekturmodell **SAM**) bereits formalisiert vor und ist damit zu großen Teilen dem Generator bereits zugängig. Auch die Verfeinerungen des SAM, die Entwurfsmodelle der einzelnen Komponenten (Component Design Models, **CDM**) können mit wenig Aufwand angepasst bzw. angereichert werden. Für die genannten Modelle sind dann auch bereits Bearbeitungswerkzeuge im Einsatz, die im jeweiligen Projektumfeld bewährt sind, etwa spezielle UML-Modellierungstools. Zudem sind (wenn wie in [8] verfahren wird) alle Modelle zentral in einem Repository gespeichert, so dass der Generator daraus lesen und wieder zurückschreiben kann, was die Integration weiter vereinfacht. Um im Generator nicht Modellinhalte verarbeiten zu müssen, die im Repository liegen, weil sie für andere Entwicklungsschritte benötigt werden, muss das Modell noch bereinigt werden. Dies wird durch eine vorgeschalte Model-to-Model-Transformation (z.B. mit Hilfe einer QVT-Unterstützung [9] des Modellierungstools) erreicht.

Sobald festgelegt ist, welche Teile der Software letztendlich vom Generator erstellt werden, muss die Architektur des Gesamtsystems überarbeitet werden (vgl. Abbildung 6). Die manuell bzw. automatisiert erstellten Bereiche müssen auf Architekturebene getrennt werden und nur lose durch ausgewählte Mechanismen (z.B. Entwurfsmuster) aneinander gekoppelt werden. Der Umbau der Architektur wird im Regelfall ein Refactoring der Anwendung nach sich ziehen.

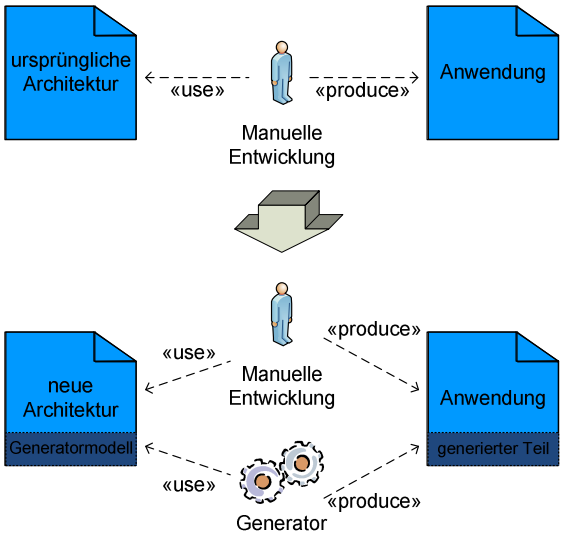

Abbildung 6: Refactoring der Architektur und Kopplung über eine Schnittstellenschicht

Typischerweise werden Generatoren parallel zur eigentlichen Anwendungsentwicklung konzipiert, implementiert und getestet. Der dazu notwendige Aufwand kann relativ leicht der Belastung durch die Produktentwicklung angepasst werden, indem z.B. die Eingabedokumente lediglich für den Generator mit Kommentaren angereichert, in der Entwicklung aber weiterhin manuell eingesetzt werden. So kann der bestehende, manuelle Entwicklungsprozess bis zu einem Zeitpunkt beibehalten werden, der für den Wechsel auf die automatische Generierung günstig ist. Auch Teilergebnisse des Generators (etwa Header-Dateien bzw. Interfaces oder die API-Dokumentation) können von der Anwendungsentwicklung verwendet werden, sofern ein Generatorentwickler (evtl. mit Unterstützung eines Anwendungsentwicklers) die in frühen Phasen noch fehleranfällige Ausgabe des Generators prüft.

Wird dann nach Fertigstellung des Generators von manueller Entwicklung auf automatisierte umgeschaltet, so ist wiederum die Erfahrung der Entwickler gefragt, die das Altsystem und das manuelle

Vorgehen kennen. Nur sie können ausreichend beurteilen, ob das Ergebnis des Generators auch dem entspricht, was aus der manuellen Entwicklung entstanden ist. In einer Übergangsphase sollte also die Kontrolle der Ein- und Ausgaben noch von jeweils einem Mitglied der „angrenzenden“ Teams mit besonderem Augenmerk durchgeführt werden. Um diese Kontrolle zu unterstützen, ist eine stabile Testsuite für den Generator zu konzipieren und deren Pflege und Anwendung im Prozess zu verankern. Die Unittests der bestehenden Software können weiterverwendet werden, und auch ein Back-to-Back Systemtest gegen die manuell erstellte Software ist machbar.

Stellen sich dabei Schwierigkeiten im generierten Code heraus, so tritt ein weiterer Vorteil der Vielzweck-Generatoren zu Tage: Da sie für viele unterschiedliche Einsatzszenarien konzipiert wurden, bringen sie oft Entwicklungswerkzeuge mit, um ihre Ausführung zu analysieren. Das oben angesprochene *openArchitectureWare*-Framework beispielsweise bietet Debugger, um die Ausführung des Generators und die Auswertung der Schablonen bei der Fehlersuche schrittweise zu überwachen. Im Projekt zur ECU-Diagnose (siehe 2.2) wurde gerade durch solche Entwicklungswerkzeuge die Akzeptanz des Generators wesentlich verbessert.

Bei der Entwicklung des Generators und der ihn unterstützenden Werkzeuge (Editoren für die Eingabeartefakte, Viewer und Checker für die Ausgabeartefakte, ggf. auch Debugger) ist es wichtig, zwischen den Teams folgendes Rollenverständnis zu etablieren: Sowohl die Domänenexperten als auch die Anwendungsentwickler sind Kunden des Generatorenteams. Die Domänenexperten beziehen Werkzeuge zur Bearbeitung ihrer Daten, die Anwendungsentwickler beziehen Softwarekomponenten bzw. Subsysteme. Grundsätzlich sollten daher auch die Domänenexperten und Anwendungsentwickler, z.B. im Rahmen von Workshops und Zwischenberichten, in die Entwicklung des Generators und der Werkzeuge mit einbezogen werden. So können sie einerseits ihre Kundenrolle einnehmen und z.B. Werkzeug-Features priorisieren, andererseits wird so auch das „Not-Invented-Here“-Phänomen abgemildert.

4 Strategie zur Einführung des Generators

Kaum ein Projekt fängt von vornherein mit einem generativen Ansatz an. Deshalb muss die Einführung behutsam erfolgen um, nicht die Akzeptanz zu verlieren. Hat man die Situation im jeweiligen Projekt analysiert und hält den Einsatz eines Generators für sinnvoll und vielversprechend, so kann man schrittweise nach folgender Strategie vorgehen, wobei vorausgesetzt wird, dass ein modellgetriebener, Repository-basierter Entwicklungsprozess bereits installiert ist:

1. Entwicklung (oder Anpassung) eines Domänen-Metamodells, Entwicklung von Editoren
2. Erstellung von Domänenmodellen mit Hilfe der neuen Editoren
3. Entwicklung von Prüfroutinen für das Repository, um dessen Konsistenz zu sichern
4. Entwicklung von Konversionsroutinen, um bestehende Datenvorräte zu transportieren
5. Bereitstellung von Model-to-Model-Transformationen, um aus dem Repository die für den Generator relevanten Informationen herauszufiltern
6. Entwicklung der Schablonen für die Ausgabe des Generators
7. Refactoring der Architektur, um manuell und automatisch erzeugte Teile zu entkoppeln
8. Entwicklung von Teststrategien und -eingabedaten für die automatisch erzeugten Programmteile
9. Entwicklung der Schablonen für Testgerüste und Einbettung der Generatortests in die der gesamten Anwendung
10. Anpassung des Entwicklungsprozesses durch Einbindung von Prüfroutinen im Repository, Ausführung des Generators und Anwendung der Generatortests

5 Fazit

Bei der Einführung von Generatoren in bestehenden Entwicklungsprojekten muss von vornherein festgelegt werden, welchen Schritt der Generator im Entwicklungsprozess erledigen soll. Diese Menge an Entwicklungsaktivitäten bezeichnet man als Generatorschritt. Für den Generatorschritt sind Ein- und Ausgabedaten vollständig zu bestimmen und in einem automatisch verarbeitbaren Format festzulegen. Sofern für diesen Schritt ein spezialisierter Generator vorliegt, ist diesem der Vorzug zu geben. Bereits bei geringer Abweichung sollte abgewogen werden, ob nicht Konzeption, Realisierung und Einsatz eines Vielzweck-Generators weniger Aufwand bedeutet als die Anpassung des Projektes an den Spezialgenerator. Eine sehr gut geeignete Basis für den domänenspezifischen Einsatz von Vielzweck-Generatoren ist die Kombination aus GMF und *openArchitectureware*.

Ein modellgetriebener Entwicklungsansatz ist die ideale Umgebung für einen Vielzweck-Generator, weil in der Regel die meisten Eingabeartefakte schon formalisiert und automatisiert bearbeitbar vorliegen.

Erfolgsfaktoren für den Einsatz sind die Identifikation eines lohnenden Generatorschritts, die personelle Einbindung von Knowhow-Trägern in die Generatorentwicklung, die Werkzeugunterstützung für die Bearbeitung und Prüfung der Ein- und Ausgabedokumente, sowie die durchgängige Testbarkeit des Projekts.

Links

[1] Code generation conference, http://www.codegeneration.net/conference/programme.php

[2] Tolvanen, Juha-Pekka: Model-Based Code Generation: 20 Example Cases From the Past 10 Years, http://www.sigs-datacom.de/sd/kongresse/oop_2008/program.php?cat=session&ID=50

[3] Java Data Objects **JDO**, http://java.sun.com/jdo/

[4] BEA Kodo, http://www.bea.com

[5] Eclipse Graphical Modeling **GMF**, http://www.eclipse.org/modeling/gmf/

[6] openArchitectureWare, http://www.openarchitectureware.org/

[7] Vlissides, John: Generation Gap Pattern, http://www.research.ibm.com/designpatterns/pubs/gg.html

[8] Al-Hilank, Samir; Jung, Martin: Tool-Unterstützung für die architekturgetriebene Entwicklung, Objektspektrum 03/2007

[9] Meta Object Facility (MOF) Query/View/Transformation **QVT**, http://www.omg.org/spec/QVT/