Software-Modellierung ohne Kunstgriffe

Mit domänenspezifischer Modellierung zu hundertprozentiger Code-Generierung

Die domänenspezifische Modellierung erhöht den Abstraktionsgrad beim Programmieren, indem sie bei der Spezifikation einer Lösung die Konzepte der Anwendungsdomäne (einschließlich ihrer Semantik) direkt verwendet. Die Endprodukte werden aus diesen High-Level-Spezifikationen generiert. Diese Automatisierung ist dadurch möglich, dass sowohl die Sprache als auch die Generatoren die Anforderungen von lediglich einem Unternehmen und einem Anwendungsbereich erfüllen müssen.

Von Dr. Steven Kelly

Beim Software-Engineering wird ständig versucht, die Produktivität der Entwickler zu steigern. Die erfolgreichsten Produktivitätsverbesserungen konnten in der Vergangenheit dadurch erreicht werden, dass sich das Abstraktionsniveau, auf dem Probleme bei der Systementwicklung gelöst werden, immer weiter erhöht hat. Der Übergang von Assembler hin zu Programmiersprachen der dritten Generation hat zu massiven Produktivitätssteigerungen von 400 % geführt. Die automatische Erzeugung von Assembler-Code auf der untersten Ebene durch eine höhere Programmiersprache spielt dabei eine zentrale Rolle. Das ist „echte“ Code-Generierung, nicht zu verwechseln mit einer Übersetzung von Modellen in Code, wo die Modelle dieselben Konzepte besitzen wie der Code. Wenn man eine Klasse in einem Diagramm vorliegen hat und diese dann in eine Code-Klasse transformiert, wird dadurch weder das Abstraktionsniveau angehoben noch wird die Produktivität wesentlich verbessert.

Die domänenspezifische Modellierung ist ein Ansatz, um den Abstraktionsgrad beim Programmieren weiter zu erhöhen. Im Idealfall verwendet die domänenspezifische Modellierung dieselben Konzepte wie das Arbeitsgebiet bzw. das Produkt und folgt dabei auch denselben Regeln. Jede Domäne ist anders und hat ihre eigenen Abstraktionen, Konzepte und Regeln. Dies lässt sich an ein paar Beispielen erläutern:

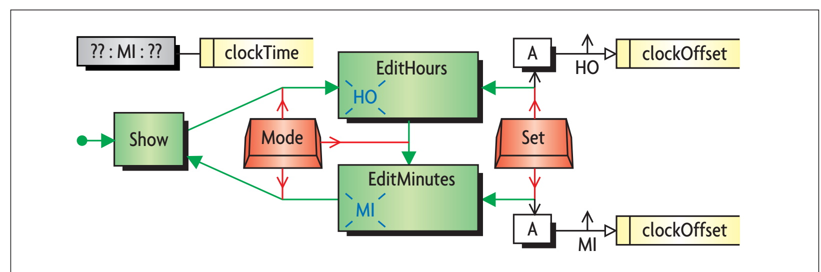

Bild 1. Beispiel für domänenspezifische Modellierung anhand einer Uhren-Applikation.

Wenn beispielsweise Applikationen für ein mobiles Gerät entwickelt werden, warum sollen dann nicht die Begriffe der Benutzerschnittstelle und der mobilen Dienste direkt in der Entwurfssprache verwendet werden? Es ist doch viel naheliegender und natürlicher, mit Begriffen wie „Liste“, „SMS“ oder „Lesen“ über die Anwendungslogik nachzudenken als in C-Code. Und wenn ein System für die Sprachkommunikation entwickelt wird, dann orientieren sich Mikrocontroller-Begriffe wie „Menü“, „Abfrage“ und „Sprachnachricht“ viel enger am Endprodukt als Assembler-Befehle. Eine Modellierung, die die Besonderheiten eines sprachgesteuerten Systems berücksichtigt, kann darüber hinaus garantieren, dass die modellierten Systeme auch gültig sind: Ein Menü ohne Benutzereinträge oder die Behandlung von Timeouts führen mit Sicherheit nicht zu einem guten Sprachkommunikationssystem; daher sollte es nicht möglich sein, ein Menü zu entwerfen, in dem Einträge fehlen.

Domänenspezifische Modellierungssprachen sind High-Level-Spezifikationssprachen, die domänenspezifische Konzepte direkt als Modellierungskonstrukte verwenden. Da die Sprache sich an den Abstraktionen und der Semantik der Domäne orientiert, können die Modellierer direkt mit den Domänenkonzepten arbeiten. Die Konzepte des Arbeitsbereichs sind im Allgemeinen bekannt und werden bereits angewendet, sind „natürlicher“ als Code und spiegeln die zugrundeliegenden Verarbeitungsmodelle wider, die benötigt werden, um die Produkte zu entwerfen. Der Code (z.B. Assembler, C, objektorientiert) kann nach wie vor aus diesen High-Level-Spezifikationen generiert werden. Der Entwicklungsprozess wird dadurch automatisiert, was wiederum eine Steigerung der Produktivität, eine Verbesserung der Qualität und eine Kapselung der Komplexität zur Folge hat.

Aber warum soll die Code-Generierung aus domänenspezifischen Modellen erfolgreich sein, wo es sich doch schon so oft gezeigt hat, dass der aus Modellen generierte Code eine große Enttäuschung sein kann? Der wichtigste Grund für den Erfolg der domänenspezifischen Modellierung ist, dass sowohl die Modellierungssprache als auch die Generatoren die Anforderungen von lediglich einem Unternehmen erfüllen müssen. In jüngster Zeit sind offene und individuell anpassbare Technologien verfügbar, mit denen Entwickler sowohl die Entwurfssprachen

als auch die Code-Generatoren so verändern können, dass sie die speziellen Anforderungen der Software-Entwicklung in ihrem Unternehmen oder Projekt erfüllen. Erfahrene Entwickler in einer Firma können die Entwurfssprachen und Generatoren daher an eine spezielle Domäne anpassen – neue Produkte können dann mit domänenspezifischen Sprachen entworfen und direkt aus den Modellen generiert werden.

Mit der Begrifflichkeit der Anwendung

Angenommen, eine Firma stellt Uhren her und die Entwickler sind verantwortlich für die Uhren-Applikationen, z.B. Stoppuhr und Weltzeit. Bevor irgendwelche Features implementiert werden können, müssen die Entwickler sich über den Funktionsumfang der Uhren klar werden. Das tun sie anhand der Begriffe und Regeln, die man von Uhren kennt: Knöpfe, Weckzeit, Minuten, Sekunden, Zustände und Benutzeraktionen. Die domänenspezifische Methode verwendet dieselben Konzepte direkt in der Modellierungssprache. *Bild 1* zeigt ein Beispiel einer solchen Modellierungssprache. Das Modell repräsentiert das Feature für die Einstellung der Uhrzeit: die Aktionen, die ein Benutzer durch das Drücken von Knöpfen auslösen kann, die leuchtenden Display-Elemente und die Aktionen, die zum Verändern der Uhrzeit führen. Entsprechend beginnt die Definition der Sprache damit, dass die Terminologie und die Konzepte für die Modellierungssprache (z.B. Knopf, Icon) identifiziert werden. Diese werden in einem ausführbaren Metamodell spezifiziert, das als Ergebnis ein domänenspezifisches Modellierungswerkzeug liefert.

Ergänzend zu den Begriffen der Anwendungsdomäne wird für die Modellierung ein Verarbeitungsmodell benötigt: Im vorliegenden Fall ist dies ein Zustandsautomat (State Machine), ein typisches Modell für eingebettete Software. Das Modell des domänenspezifischen Zustandsautomaten beschreibt die Applikation auf der Ebene der Anwendungsdomäne („Uhren“) und nicht auf der wesentlich detaillierteren Implementierungsebene (d.h. dem Programm-Code). Das Modell arbeitet auf einer höheren Ebene; aus diesem Grund muss viel weniger modelliert werden als in Sprachen, die den Schwerpunkt auf der Code-Visualisierung haben. Die Modelle enthalten ausreichend Informationen, damit Code-Generatoren den gewünschten Code erzeugen können.

Im letzten Schritt der Spracherzeugung wird die Semantik des Zustandsautomaten erweitert, um domänenspezifische Regeln zu integrieren. Andere Optionen, die für den vorliegenden Anwendungsbereich nicht relevant sind, werden weggelassen. Für das Beispiel der Uhren-Anwendung gibt es zwei Erweiterungen der generischen Semantik des Zustandsautomaten:

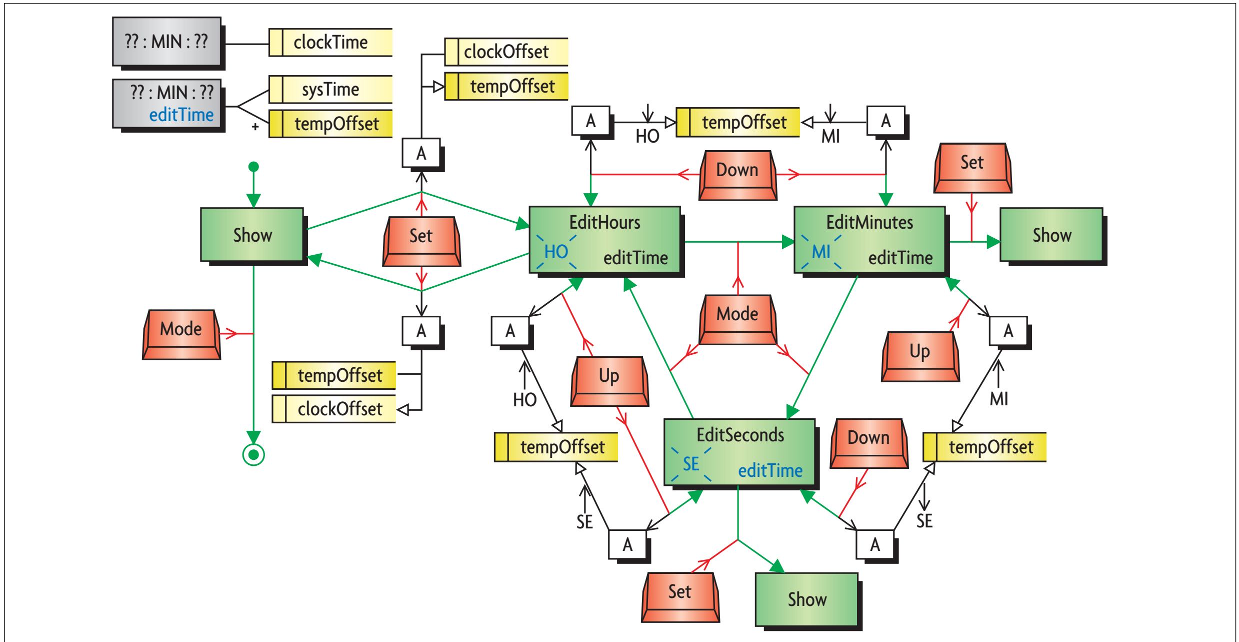

Bild 2. Eine komplexere Variante der Uhren-Anwendung aus Bild 1, mit erweiterten Regeln und Befehlen.

- Erstens können die Zustandsübergänge nur während der Interaktion mit dem Benutzer ausgelöst werden, wenn ein bestimmter Knopf gedrückt wird.
- Zweitens dürfen die Aktionen, die während eines Zustandsübergangs ausgeführt werden, nur Daten vom Typ „Zeiteinheit" verändern.

Außerdem ist die Menge möglicher Operationen begrenzt: Eine Operation kann Zeiteinheiten nur setzen, addieren oder subtrahieren oder diese durchlaufen. Folgendes ist an dieser Stelle noch einmal zu betonen: Sollten zu einem späteren Zeitpunkt weitere Anforderungen hinzukommen, kann die Menge möglicher Operationen erweitert werden oder es können neue Objekttypen definiert werden. Mit den erweiterten Basisoperationen können im vorliegenden Beispiel alle Anforderungen abgedeckt werden, wobei sich die Entwicklung mit Code-Generatoren automatisieren lässt. *Bild 2* zeigt eine weiterentwickelte Variante der in Bild 1 dargestellten Applikation.

Erzeugung von vollständigem, lauffähigem Code

Der Code-Generator spezifiziert, wie Informationen aus den Modellen extrahiert und in Code umgewandelt werden. Dieser Prozess ist abhängig beziehungsweise wird geleitet von der Modellierungssprache mit ihren Konzepten, ihrer Semantik und ihren Regeln sowie von der Syntax, die die Zielplattform als Eingabe erwartet. Um brauchbare Ergebnisse zu bringen, muss der Generierungsprozess vollständig sein: Aus Sicht der Anwendungsentwickler wird vollständig lauffähiger Code generiert, ein manuelles Überarbeiten des Codes ist nicht erforderlich.

Eine solche vollständige Code-Generierung ist schwierig – wenn nicht sogar unmöglich – zu erreichen, wenn der Generator (und die Modellierungssprache, die den Input liefert) so konzipiert wurde, dass er sich für fast alle Situationen eignet. Vollständige Code-Generierung ist dann möglich, wenn sowohl die Sprache als auch die Generatoren die Anforderungen von lediglich einem Unternehmen erfüllen müssen. Das bedeutet, dass der Code-Generator optimal mit der Modellierungssprache, von der er die Eingaben erhält, und mit der Zielplattform, auf der der erzeugte Code laufen wird, zusammenarbeiten muss.

Während Modelle die Daten als Input für den Code-Generierungsprozess zur Verfügung stellen, finden die eigentliche Navigation in den Designinformationen sowie deren Gewinnung in Anlehnung an das Metamodell statt. Daher sollte das Metamodell sich an einem Verarbeitungsmodell orientieren, das „natürlich" für das zu entwickelnde Produkt ist. Meistens sind bestimmte domänen- oder codegeneratorspezifische Änderungen bzw. Erweiterungen am zugrundeliegenden Modell erforderlich. Dies macht man, um sicherzustellen, dass in den Modellen auch alle wesentlichen statischen und dynamischen Aspekte des Produkts als Input für den Code-Generator erfasst werden. Im Uhren-Beispiel ist der Zustandsautomat das Verarbeitungsmodell. Er wurde dann um Konzepte wie „Zeiteinheit" erweitert, um den Anforderungen der Domäne und der Code-Generierung zu entsprechen.

Die Rolle des Plattform-Frameworks

Eine Plattform stellt eine wohldefinierte Menge von Diensten zur Verfügung, die der Code-Generator benutzen kann. Beispielsweise kann der generierte Code die Plattform-Komponenten und ihre Dienste direkt aufrufen. Trotzdem ist es oft gut, zusätzlichen Framework-Code oder -Komponenten zu definie-

ren, um die Code-Erzeugung zu vereinfachen. Ein solches Framework kann oben auf der Plattform aufsetzen – in Form von Bibliotheken, Komponenten und Code-Templates. Das Framework ist nicht notwendigerweise eine zusätzliche Last, die nur vom Code-Generator benötigt wird. Meistens verwendet die zugrundeliegende Software-Architektur ohnehin bereits diverse Bibliotheken, Komponenten oder andere wiederverwendbare Teile, die die Code-Generierung ebenfalls unterstützen können.

```
01 typedef enum { Start, EditHours, EditMinutes, Show, Stop } States;
02 typedef enum { None, Mode, Set } Buttons;
03
04 int state = Start;
05 int button = None; /* pseudo-button for following buttonless transitions */
06
07 void runWatch()
80 {
09   while (state != Stop)
10   {
11     handleEvent();
12     button = getButton(); /* waits for and returns next button press */
13   }
14 }
15
16 void handleEvent()
17 {
18   int oldState = state;
19   switch (state)
20   {
21     case Start:
22       switch (button)
23       {
24         case None:
25           state = Show;
26           break;
27         default:
28           break;
29       }
30     case EditHours:
32       switch (button)
33       {
34         case Set:
35           roll(clockOffset, HOUR_OF_DAY, 1, displayTime());
36           break;
37         case Mode:
38           state = EditMinutes;
39           break;
40         default:
41           break;
42       }
43     case EditMinutes:
44       switch (button)
45       {
46         case Mode:
47           state = Show;
48           break;
49         case Set:
50           roll(clockOffset, MINUTE, 1, displayTime());
51           break;
52         default:
53           break;
54       }
55     case Show:
56       switch (button)
57       {
58         case Mode:
59           state = EditHours;
60           break;
61         default:
62           break;
63       }
64     default:
65       break;
66   }
67   button = None; /* follow transitions that do not require buttons */
68   if (oldState != state) handleEvent();
69 }
```

Listing 1. C-Code, der aus dem Zustandsdiagramm in Bild 1 generiert wurde

```
01 // All this code is generated directly from the model.
02 // Since no manual coding or editing is needed, it is
03 // not intended to be particularly human-readable
04
05 public class SimpleTime extends AbstractWatchApplication {
06
07   // define unique numbers for each Action (a...) and DisplayFn (d...)
08   static final int a22_1405   = +1; //+1+1
09   static final int a22_2926   = +1+1; //+1
10   static final int d22_977    = +1+1+1; //
11
12
13   public SimpleTime(Master master) {
14     super(master);
15
16     // Transitions and their triggering buttons and actions
17     // Arguments: From State, Button, Action, To State
18     addTransition („Start [Watch]“, „“, 0, „Show“);
19     addTransition („Show“, „Mode“, 0, „EditHours“);
20     addTransition („EditHours“, „Set“, a22_2926, „EditHours“);
21     addTransition („EditHours“, „Mode“, 0, „EditMinutes“);
22     addTransition („EditMinutes“, „Set“, a22_1405, „EditMinutes“);
23     addTransition („EditMinutes“, „Mode“, 0, „Show“);
24
25     // What to display in each state
26     // Arguments: State, blinking unit, central unit, DisplayFn
27     addStateDisplay(„Show“, -1, METime.MINUTE, d22_977);
28     addStateDisplay(„EditHours“, METime.HOUR_OF_DAY, METime.MINUTE, d22_977);
29     addStateDisplay(„EditMinutes“, METime.MINUTE, METime.MINUTE, d22_977);
30   };
31
32   // Actions (return null) and DisplayFns (return time)
33   public Object perform(int methodId)
34   {
35     switch (methodId) {
36       case a22_2926:
37         getclockOffset().roll(METime.HOUR_OF_DAY, true, displayTime());
38         return null;
39       case a22_1405:
40         getclockOffset().roll(METime.MINUTE, true, displayTime());
41         return null;
42       case d22_977:
43         return getclockTime();
44     }
45     return null;
46   }
47 }
```

Listing 2. Für die Uhren-Anwendung generierter Java-Code

Der zentrale Punkt bei der Entwicklung eines Code-Generators ist, wie die Modellkonzepte auf den Code abgebildet werden. Der Output wird nicht als konkreter Code definiert, sondern vielmehr als ein Beispiel oder Template. Im einfachsten Fall erzeugt jedes bei der Modellierung verwendete Symbol bestimmten festen Code, einschließlich der Werte, die in die Eigenschaftsfelder (property fields) bei den jeweiligen Symbolen eingetragen sind. Der Generator kann auch – abhängig von den Werten am Symbol, den Beziehungen, die es zu anderen Symbolen hat, oder anderen im Modell enthaltenen Informationen – unterschiedlichen Code erzeugen.

Die Definition des Generators sollte so direkt und einfach wie möglich erfolgen. Dies kann dadurch erreicht werden, dass innerhalb des Code-Generators keine Varianten oder Implementierungsdetails behandelt werden. Das Framework und die Komponentenbibliothek können diese Aufgabe einfacher erledigen und dadurch den Abstraktionsgrad auf der Code-Seite erhöhen. Auch vereinfacht es die Definition des Generators, wenn die domänenspezifischen Modelle Regeln (Correctness Constraints) umfassen: Der Generator muss dann nicht prüfen, ob die Eingaben korrekt sind. Saubere Modularisierung und Wiederverwendung helfen eine Menge beim Implementieren des Code-Generators. Wenn beispielsweise die Behandlung von Varianten verschiedener Zielplattformen in gesonderte Module ausgelagert wird, dann ist es ganz einfach, die Plattform-Unterstützung zu erweitern: Beim Hinzufügen einer neuen Plattform sind lediglich neue Versionen dieser Module, nicht des ganzen Generators erforderlich.

Beispiel für einen Code-Generator

Wie werden die bisher diskutierten Teile der Lösung nun zusammengefügt, um eine hundertprozentige Code-Generierung zu erreichen? Zum besseren Verständnis wird dies wieder am Beispiel der Uhren-Applikation gezeigt: *Listing 1* zeigt den Code, der aus dem in Bild 1 dargestellten Zustandsautomaten generiert wurde. Die Code-Generierung ist vollständig, da aus Sicht des Modellierers kompletter Code erzeugt wurde und ein manuelles Nachbearbeiten nicht erforderlich ist.

Nach der Initialisierung der Variablen wird die Funktion runWatch() aufgerufen, die den Input dem Teil der Applikation, der den Großteil der Arbeit erledigt, übergibt (Zeilen 19 bis 66). Für jeden Zustand gibt es Knöpfe, die Zustandsübergänge auslösen können. Deshalb wurde der Zustandsautomat als einfache geschachtelte Switch-Anweisung implementiert. Ein Zustandsübergang kann außerdem Aktionen auslösen: Im Uhren-Beispiel sind die einzigen Aktionen, die diesbezüg-

lich auftreten können, einfache arithmetische Operationen auf Zeiteinheiten. Das Setzen von Zeit-Variablen ist eine simple Zuweisung, das Durchlaufen von Zifferpaaren nach oben oder unten ist als Funktion in unserem Framework implementiert. Wenn eine solche Funktion benötigt wird, setzt der Code-Generator lediglich einen entsprechenden Prozeduraufruf ab (wie z.B. in den Zeilen 35 und 50).

Wie bereits erwähnt, beschreiben domänenspezifische Modelle die Anwendungsfunktionalität unabhängig vom Code auf einer höheren Abstraktionsebene. Daher kann aus denselben Modellen Code für unterschiedliche Plattformen generiert werden. Wie das Beispiel in *Listing 2* zeigt, kann aus demselben Design Java-Code erzeugt werden: Nur der Generator ist ein anderer, das Design der Applikation ist dasselbe.

An Listing 2 fällt als erstes auf, dass der Generator die Komplexität des Zustandsautomaten verbirgt, indem er dessen grundlegendes Verhalten als abstrakte Framework-Klasse implementiert. Der konkrete Zustandsautomat ist dann eine Unterklasse von dieser abstrakten Klasse (Zeile 5) und wird mit den Daten aus dem Designmodell im Klassenkonstruktor initialisiert (Zeilen 13 bis 30). Die Aktionen erhalten willkürliche Namen und werden von der Switch-Anweisung aufgerufen (Zeilen 35 bis 44). Dies ist ein Beispiel dafür, wie man ein Framework implementiert, das einem logischen Modellkonstrukt entspricht.

Mehr Qualität, weniger Aufwand

Domänenspezifische Modellierung beschleunigt die Entwicklung dadurch, dass an Stelle von Code-Modellen Produktmodelle eingesetzt werden. Das in diesem Artikel vorgestellte Beispiel verdeutlicht das. Berichte über Erfahrungen mit domänenspezifischer Modellierung aus der Industrie zeigen große Verbesserungen bei der Produktivität, niedrigere Entwicklungskosten und eine bessere Qualität. Das Unternehmen Nokia gibt beispielsweise Zahlen an, denen zufolge es mit dieser Methode Mobiltelefone bis zu zehnmal schneller entwickelt; bei der Firma Lucent konnte die Produktivität – abhängig vom Produkt – um das Drei- bis Zehnfache gesteigert werden. Die Schlüsselfaktoren hierbei sind:

- Das Problem wird nur einmal – und zwar auf einem hohen Abstraktionsniveau – gelöst und der Code wird direkt aus dieser Lösung erzeugt.
- Das Hauptaugenmerk der Entwickler liegt nicht mehr auf dem Code, sondern auf dem Design, also dem Problem selbst. Komplexität und Implementierungsdetails können so verborgen werden und die Modelle sind beides: Design und Implementierung.
- Dank einer einheitlicheren Entwicklungsumgebung und dadurch, dass nicht mehr so viele Wechsel zwischen den Ebenen Design und Implementierung erforderlich sind, können eine Konsistenz der Produkte und niedrigere Fehlerraten erreicht werden.
- Das Domänenwissen wird für das Entwicklerteam sichtbar gemacht und findet sich in der Modellierungssprache und deren Werkzeugunterstützung wieder.

Die Implementierung von domänenspezifischer Modellierung und Code-Generierung stellt keine zusätzliche Investition dar, wenn man sich den gesamten Zyklus vom initialen Entwurf bis hin zum lauffähigen Code vor Augen hält. Tatsächlich spart dieser Ansatz sogar Entwicklungsressourcen: Normalerweise arbeiten alle Entwickler mit den Konzepten der Problemdomäne und bilden diese von Hand auf die Implementierungskonzepte ab. Aber zwischen den Entwicklern gibt es große Unterschiede: Einige sind Experten, die meisten jedoch nicht. Wenn daher die erfahrenen Entwickler die Konzepte und deren Abbildungen einmal definieren, dann müssen die anderen dies nicht noch einmal tun. Ein von einem Experten definierter Code-Generator wird mit Sicherheit qualitativ bessere Applikationen erzeugen, als „durchschnittliche“ Entwickler dies manuell tun. *jk*

Dr. Steven Kelly
ist Chief Technology Officer der Firma MetaCase. Er verfügt über langjährige Erfahrung bei der Entwicklung von MetaCase-Umgebungen und berät bei ihrer Anwendung für die domänenspezifische Modellierung.
E-Mail: stevek@metacase.com

Übersetzung aus dem Englischen: Kirsten Waldheim (E-Mail: KParitong@aol.com)