von juha-pekka tolvanen und steven kelly

mehr zum thema:
www.metacase.com
www.gpce.org

DOMÄNENSPEZIFISCHE MODELLIERUNG

Indem die domänenspezifische Modellierung bei der Spezifikation einer Lösung die Konzepte der Anwendungsdomäne (einschließlich ihrer Semantik) direkt verwendet, wird der Abstraktionsgrad beim Programmieren erhöht. Die Endprodukte werden aus diesen High-End-Spezifikationen generiert. Diese Automatisierung ist deshalb möglich, weil sowohl die Sprache als auch die Generatoren die Anforderungen von lediglich einem Unternehmen und einer Anwendungsdomäne erfüllen müssen. Der Artikel beschreibt die domänenspezifische Modellierung und erläutert, wie solche Sprachen und Generatoren implementiert werden können.

Mit der domänen-getriebenen Entwicklung (*Domain-Driven Development*) wird das Ziel verfolgt, Code und Problemdomäne näher zusammenzubringen. Bestrebungen, eine stärkere Übereinstimmung zwischen Code und Anwendungsbereich herzustellen, gibt es bereits seit Jahrzehnten: Ausgehend von Assembler hat sich der Abstraktionsgrad der Programmiersprachen immer weiter erhöht. Dennoch finden sich auch in neueren Programmiersprachen nur mäßige Verbesserungen, um Abstraktionen von der Problemdomäne besser zu unterstützen. Java und C# unterscheiden sich hier kaum von C++.

Die domänenspezifische Modellierung (*Domain-Specific Modeling – DSM*) ist eine Technik, die die domänenen-getriebene Entwicklung unterstützt. DSM bietet einen Ansatz, um den Abstraktionsgrad beim Programmieren weiter zu erhöhen. Im Idealfall verwendet DSM dieselben Konzepte wie die Problemdomäne und folgt dabei auch denselben Regeln wie diese. Tatsächlich ist jede Domäne anders als andere. Jede Domäne hat ihre eigenen speziellen Abstraktionen, Konzepte und Regeln. Sehen wir uns ein paar Beispiele an.

Wenn wir ein Portal für den Produktvergleich und den Abschluss von Versicherungen entwickeln, warum sollen wir dann nicht die Terminologie der Versicherungsbranche direkt in unserer Entwurfssprache verwenden? Mit Begriffen wie „Risiko", „Bonus" und „Schaden" können wir die Gegebenheiten von Versicherungen besser erfassen, als dies mit Java-Klassen möglich ist. Eine versicherungsspezifische Sprache kann außerdem garantieren, dass die modellierten Produkte auch tatsächlich den dortigen Regeln entsprechen: Eine Versicherung ohne Prämie ist kein gutes Produkt – daher sollte es nicht möglich sein, ein solches Produkt zu modellieren.

Wenn wir beispielsweise mobile Applikationen für ein Gerät entwickeln, warum sollen wir dann nicht die Begriffe der Benutzungsschnittstelle und mobilen Dienste direkt in der Entwurfssprache verwenden. Es ist doch viel nahe liegender und natürlicher mit Begriffen wie „Liste", „SMS" oder „View" über die Anwendungslogik nachzudenken als in C-Code. Und wenn wir Systeme für Sprachkommunikation entwickeln, dann orientieren sich Mikrocontroller-Begriffe wie „Menü", „Prompt" und „Voice Entry" viel enger an der Problemdomäne als Assembler-Befehle.

Der Artikel beschreibt die domänenspezifische Modellierung und erläutert, wie mit diesem Ansatz Entwurfsabstraktionen enger an einer vorgegebenen Problemdomäne vorgenommen werden können. Bei DSM geht es aber nicht nur darum, Entwürfe zu erstellen. In vielen Fällen – wenn DSM oben auf einer Plattform oder einem Framework aufsetzt – können die Endprodukte direkt aus diesen High-Level-Spezifikationen erzeugt werden. Bei allen oben genannten Beispielen kann ein domänenspezifischer Generator die Modelle lesen und daraus den vollständigen Code erzeugen. Die Generierung des kompletten Codes ist deshalb möglich, weil sich sowohl die Sprache als auch die Generatoren nur auf eine einzige Problemdomäne und deren Implementierung beziehen müssen.

Folgende Problemdomänen eigenen sich besonders gut für DSM: die Entwicklung von Produktfamilien, die plattformbasierte Entwicklung und Produktkonfiguration.

Eine Lösung zur domänenspezifischen Modellierung

In DSM repräsentieren die Modellelemente Konzepte aus der Anwendungswelt und nicht aus der Programmierwelt.

die autoren

Dr. Juha-Pekka Tolvanen (E-Mail: jpt@metacase.com) ist CEO der Firma MetaCase in Finnland. Er ist weltweit als Methodenberater tätig und Autor zahlreicher Artikel zum Thema Methoden-Engineering.

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

Die Modellierungssprache orientiert sich an den Abstraktionen und der Semantik der Domäne, sodass die Modellierer direkt mit den Domänenkonzepten arbeiten können. Richtlinien und Regeln der Anwendungsdomäne können als *Constraints* in die Sprache eingebunden werden, sodass es im Idealfall nicht möglich ist, ungültige oder unerwünschte Entwurfsmodelle zu spezifizieren. Die enge Anbindung der Sprache an die Problemdomäne bringt eine Reihe von Vorteilen mit sich; die meisten dieser Vorteile treten auch bei anderen Methoden auf, mit denen versucht wird, einen höheren Abstraktionsgrad zu erreichen: gesteigerte Produktivität sowie bessere Kapselung der Komplexität und verbesserte Systemqualität.

Ganz allgemein lässt sich sagen, dass ein höherer Abstraktionsgrad eine verbesserte Produktivität bewirkt. Das bezieht sich nicht nur auf die die für das Design benötigte Zeit und Ressourcen, sondern auch auf den Wartungsaufwand. In der Regel kommen Änderungsanforderungen nicht aus der

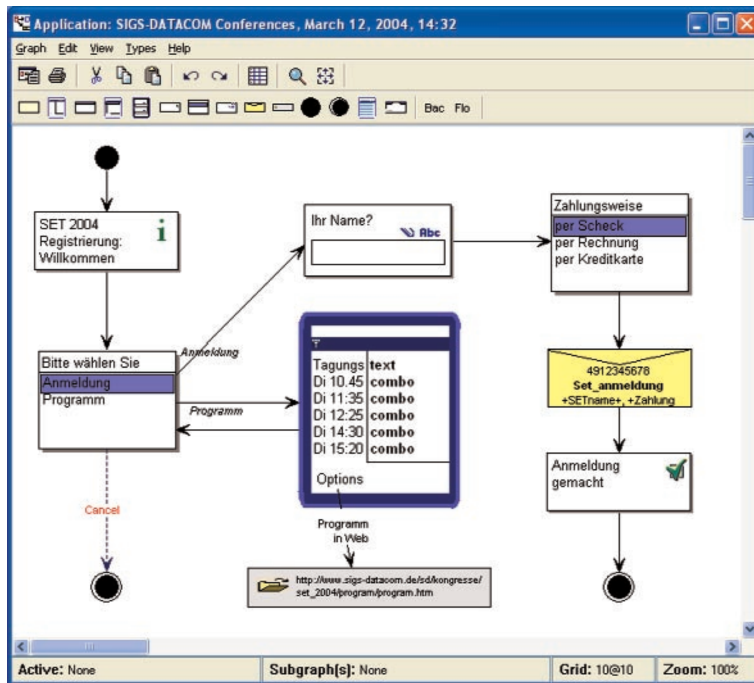

Abb. 1: Registrierung bei der Konferenz „SET 2004"

Implementierungsdomäne sondern aus der Problemdomäne, sodass es wesentlich leichter ist, diese Änderungen in einer Sprache vorzunehmen, die sich der Konzepte der Anwendungswelt bedient. Darüber hinaus ist es in einigen Domänen möglich, dass Nicht-Programmierer komplette Spezifikationen erstellen und anschließend Generatoren starten, die dann den Code erzeugen.

Spezifikationen auf einem signifikant höheren Abstraktionsniveau als traditioneller Quellcode oder Klassendiagramme zu halten, bedeutet, dass die Spezifikation weniger Arbeit macht. Da die Sprache lediglich für eine spezielle Anwendungsdomäne Gültigkeit besitzen muss – in der Regel nur innerhalb eines Unternehmens – kann die DSM sehr schlank ausfallen. Sie enthält keinerlei Techniken oder Konstrukte, die für die Entwickler unnötige Arbeit mit sich bringen würden.

DSM reduziert die Notwendigkeit, neue Semantiken lernen zu müssen. Die Konzepte der Problemdomäne sind in der Regel gut bekannt und werden bereits angewendet, wohl-definierte Semantiken existieren und werden als „natürlich" angesehen, da sie intern entstanden sind. Die domänenspezifische Semantik muss ohnehin beherrscht werden – warum soll man ihr also nicht einen besonderen Stellenwert einräumen? Die Entwickler müssen so keine zusätzlichen Semantiken (z.B. UML) erlernen und ständig zwischen Domänen- und UML-Semantik wechseln bzw. Abbildungen zwischen beiden vornehmen. Dieses überflüssige Mapping beansprucht Zeit und andere Ressourcen, ist fehleranfällig und wird von allen Designern durchgeführt – manche können es besser als andere, aber meistens machen alle es irgendwie anders.

Dass sich mit DSM qualitativ bessere Systeme bauen lassen, hat vor allem zwei Gründe: Zum einen können in DSM Korrektheitsregeln der Anwendungsdomäne verankert werden, sodass es schwer oder sogar unmöglich ist eine falsche oder unerwünschte Spezifikation zu schreiben. Übrigens ist das auch eine recht preiswerte Methode, um Fehler zu eliminieren: je früher, um so besser. Zum anderen ermöglichen Generatoren Abbildungen auf ein niedrigeres Abstraktionsniveau (in der Regel Code) und das so erzeugte Ergebnis muss anschließend nicht mehr editiert werden. Das ist übrigens einer der der Eckpfeiler diverser erfolgreicher Entwicklungen, die Programmiersprachen vollzogen haben. Oder haben Sie schon einmal jemanden gesehen, der Assembler-Code manuell editiert und versucht hat, diesen mit dem entsprechenden C++-Code zu synchronisieren? In der UML hat beispielsweise die Möglichkeit der Code-Visualisierung zu *Roundtrip*-Problemen geführt: Ist jemand eher code-fixiert, dann stellen für ihn die Modelle lediglich statische Klassenstrukturen dar. Ist jemand hingegen eher modellorientiert, so wird er viel Mühe darauf verwenden, um generierten Code umzuschreiben und alle anderen Modelle und Klassendiagramme auf dem neuesten Stand zu halten. Wenn man dieselben Informationen an zwei verschiedenen Stellen – im Code und in den Modellen – hält – sind Probleme vorprogrammiert.

Und schließlich sind Spezifikationen, die in der Domänenterminologie geschrieben sind, in der Regel besser lesbar, verständlich, erinnerbar und validierbar und es fällt leichter, mit anderen über sie zu diskutieren. Diese klaren Vorteile wollen wir im Folgenden anhand eines Beispiels verdeutlichen.

Ein Beispiel

Keine Domäne ist wie die andere und so ist auch jedes DSM-Beispiel anders. Wir verwenden im Folgenden eine Domäne, die allgemein gut bekannt ist: mobile Telefon-Anwendungen. Die spezielle Implementierungsplattform ist die verbreitete Smartphon-Plattform „Series60" (vgl. [Nog03]) mit ihrem „Amaretto"-Framework (vgl. [Nog04]). Das Framework ermöglicht die Entwicklung und Auslieferung von Unternehmensanwendungen auf Mobiltelefonen. Entsprechend werden – in dieser Domäne – im Rahmen des Entwicklungsprozesses die Konzepte und Regeln von mobilen Unternehmensanwendungen, die auf Series60 und Amaretto basieren, angewendet. In **Abbildung 1** ist dargestellt, wie die DSM-Sprache diese direkt in Modellierungskonzepte umsetzt.

Das Designmodell basiert direkt auf den Begriffen der Domäne, wie z. B. „Note", „Pop-ups", „SMS", „Form" und „Query". Diese Begriffe sind typisch für Mobiltelefon-Dienste und deren Benutzungsschnittstellen-*Widgets*. Wenn Sie sich mit Telefon-Anwendungen – z.B. Telefonbuch oder Kalender – auskennen, dann haben Sie vermutlich bereits verstanden, was die in **Abbildung 1** dargestellte Anwendung macht. Ein Benutzer kann sich per SMS bei einer Konferenz anmelden, eine Zahlungsart auswählen, Informationen über Programm und Redner einsehen oder das Konferenzprogramm über das Web durchblättern. Das Modell fußt auf dem webbasierten Registrierungssystem, das das Unternehmen SIGS-DATACOM für die Konferenz SET 2004 eingesetzt hat.

Wie man in dem Modell sehen kann, sind sämtliche Implementierungskonzepte verborgen (zu diesem Zeitpunkt ist es auch nicht nötig, sie zu kennen). Die Entwickler können sich darauf konzentrieren eine Lösung zu finden, indem sie die Konzepte der Domäne verwenden. Da in den Beschreibungen alle geforderten statischen und verhaltensbezogenen Aspekte der Anwendung erfasst sind, kann die Applikation vollständig aus den Modellen ▶

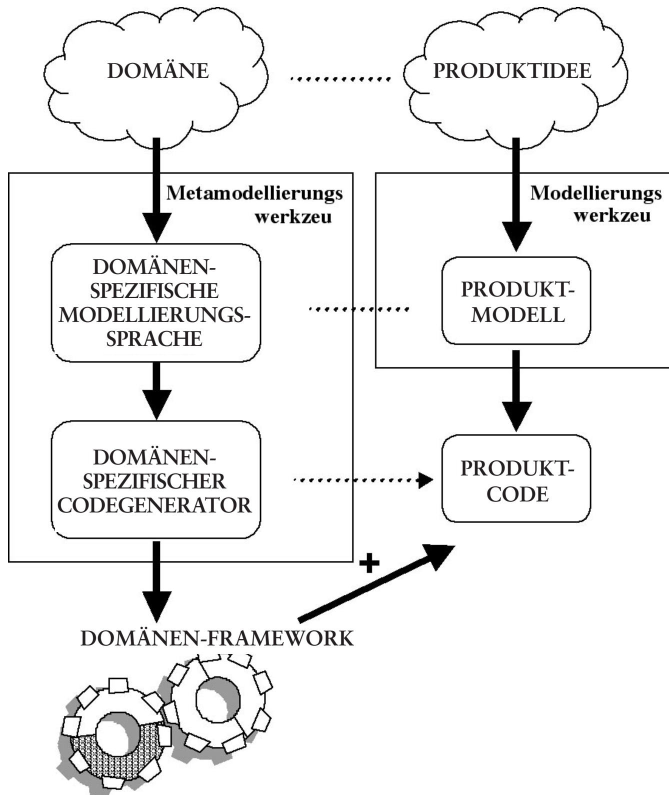

Abb. 2: Architektur einer DSM-Umgebung

generiert werden. Der erzeugte Code benutzt die von Series60 und Amaretto-Framwork angebotenen Dienste. Im Anschluss an das Design besteht keine Notwendigkeit, die Lösung auf Implementierungskonzepte in Code oder auf UML-Modelle, die den Code visualisieren, abzubilden.

Die Modellierungssprache beinhaltet darüber hinaus Domänen-Regeln, um den Entwickler daran zu hindern, unzulässige Entwürfe zu erstellen. So kann beispielsweise bei Series60/Amaretto, nachdem eine SMS verschickt wurde, nur ein Benutzungsschnittstellen-Element oder ein Telefondienst ausgelöst werden. Entsprechend erlaubt die DSM nur einen „Fluss" (*flow*)[1]) von einem SMS-Element. Aus diesem Grund muss sich der Anwendungsentwickler nicht mit den Details der Series60/Amaretto-Architektur und -Codierung befassen. Wenn Sie die Benutzungsschnittstelle des Telefons und seine Dienste verstanden haben, können Sie Anwendungen entwickeln.

Wie DSM implementiert wird

Um die erwähnten Vorteile wie verbesserte Produktivität, Qualität und Kapselung der Komplexität zu erreichen, benötigen wir eine domänenspezifische Sprache und Generatoren. In der Vergangenheit hätte man darüber hinaus noch das zu Grunde liegende Toolset implementieren müssen. Dies war einer der Hauptgründe dafür, dass sich DSM bisher noch nicht durchgesetzt hat, denn das Implementieren von CASE-Tools gehört bei den meisten Firmen wohl eher nicht zu den Kernkompetenzen.

Inzwischen beschränkt sich die erforderliche Arbeit auf das Definieren der Sprache und der Generatoren, seit offene, metamodell-basierte Tools für das Modellieren und Generieren von Code verfügbar sind. Die typische Architektur solcher Meta-CASE-Tools zeigt Abbildung 2 (vgl. [Poh00]).

Diese Tools haben einen dualen Charakter: Auf der linken Seite sieht man die Teile, die benötigt werden, um die Umgebung zu erzeugen – eine Aufgabe, die von Methodenentwicklern wahrgenommen wird. Auf der rechten Seite ist dargestellt, wie die Ingenieure, die Produkte entwickeln, diese Umgebung verwenden. Um die DSM-Definition zu erläutern, verwenden wir Series60/Amaretto als Beispiel.

Die Modellierungssprache für eine Domäne definieren

Die Definition einer Modellierungssprache umfasst drei Aspekte: die Domänenkonzepte, die Notation, die verwendet wird, um diese in grafischen Modellen darzustellen, und die Regeln, denen der Modellierungsprozess unterliegt. Eine komplette Modellierungssprache zu definieren wird als schwierige Aufgabe angesehen: Das trifft mit Sicherheit zu, will man eine Sprache für jedermann schreiben. Aber die Aufgabe stellt sich erheblich einfacher dar, wenn man sie lediglich für *eine* Problemdomäne in *einem einzigen* Unternehmen ausführen muss.

Der zentrale Ansatzpunkt, um die Konzepte einer Domäne herauszuarbeiten, ist das Fachwissen eines Domänenexperten bzw. eines kleinen Expertenteams. In der Regel ist ein solcher Experte ein erfahrener Entwickler, der für die Domäne bereits eine Reihe von Produkten entwickelt hat. Möglicherweise hat er die den Produkten zu Grunde liegende Architektur entwickel oder er war verantwortlich für den Aufbau der Komponenten-Bibliothek. Er kann die Domänenkonzepte leicht identifizieren, indem er seine eigene Terminologie, bestehende Systembeschreibungen und Dienste von Komponenten verwendet.

In unserem Telefon-Beispiel kommen die Domänenkonzepte von den Benutzungsschnittstellen-Elementen (z.B. „Form", „Pup-up"), der Menü- und Knöpfe-Struktur (z. B. benutzerdefinierbare Schlüssel, Menü) und den zu Grunde liegenden Diensten (z. B. „SMS", „Web-Browsing"). Indem wir diese Konzepte in die Modellierungssprache einbringen und weiter verfeinern, können wir den begrifflichen Teil der Modellierungssprache festlegen. Das Ziel dabei besteht darin, dass die ausgewählten Begriffe exakt mit der Domänen-Semantik übereinstimmen.

Die Domänenkonzepte allein machen allerdings noch keine Modellierungssprache aus: Wir benötigen außerdem das Domänenwissen darüber, wie die Konzepte zusammenhängen. Für unser Beispiel entscheiden wir uns für „Application Flow" als grundlegendes Verarbeitungsmodell. Dieses Modell beschreibt, wie die Domänenkonzepte während der Ausführung der Anwendung miteinander interagieren (ein Beispiel findet sich in Abb. 1). Wir führen die Begriffe „Start" und „Stop" und „Directed Flows" – mit und ohne Bedingungen – ein. Bedingungen benötigt man dann, wenn der Benutzer von mehreren Möglichkeiten eine auswählt, z. B. über eine Liste oder über ein Menü. Wir ergänzen außerdem Konzepte für Bibliothekscode und schaffen dadurch eine Möglichkeit, Erweiterungen von Drittanbietern einzubinden

[1]) *In Abb. 1 stellt der Pfeil von einem Element zu einem anderen einen solchen „Fluss" dar.*

und mit Parametern aufzurufen.

Im nächsten Schritt werden die Basiskonzepte um die Domänenregeln erweitert. Typischerweise schränken die Regeln die Verwendung der Sprache dadurch ein, dass sie festlegen, welche Arten von Verbindungen zwischen den Konzepten zulässig sind. Sie können spezifizieren, wie bestimmt Konzepte wieder verwendet und wie Modelle angeordnet werden. In unserem Telefonbeispiel definieren die Regeln, welche Benutzungsschnittstellen-Elemente ihre eigenen Menüs haben, welche von ihnen über benutzerdefinierte Knöpfe erreichbar sind und wie verschiedene Telefondienste während der Anwendungsausführung aufgerufen werden dürfen.

Die Regeln werden zusammen mit den Konzepten mit einem Metamodell kodifiziert und formalisiert. Ein Metamodell beschreibt die Modellierungssprache, ähnlich wie ein Modell ein System beschreibt (vgl. [Bri96]). Das Metamodell wird von dem Meta-CASE-Tool instanziiert und erzeugt ein domänenspezifisches Modellierungswerkzeug, das die Konzepte und Regeln der Modellierungssprache beinhaltet.

Für den notationellen Teil der Modellierungssprache definieren wir Symbole als grafische Repräsentationen der Modellierungskonzepte. Da unsere Beispieldomäne mit Benutzungsschnittstellen verwandt ist, können wir viele Symbolen ähnlich wie die von Telefon-*Widgets* gestalten.

Das Domänen-Framework definieren

Das Domänen-Framework stellt die Schnittstelle zwischen generiertem Code und der zu Grunde liegenden Plattform zur Verfügung. Manchmal wird keine eigener Framework-Code benötigt: Der erzeugte Code kann die Plattform-Komponenten direkt aufrufen, deren Dienste ausreichend sind. Aber trotzdem ist es häufig hilfreich, eigenen Framwork-Hilfscode oder Komponenten zu definieren, um die Codegenerierung zu vereinfachen. Möglicherweise gibt es solche Komponenten bereits aus früheren Entwicklungen oder Produkten, die dann lediglich ein wenig überarbeitet werden müssen.

In unserem Beispiel bietet Amaretto bereits eine ganze Reihe von Dienste an, die sich auf einem höheren Level befinden als die von Series60. Das Domänen-Framwork fügt hier lediglich zwei Funktionen hinzu: einen Dispatcher, um den „Fluss" der Applikationslogik auszuführen, und ein View-Management für Anwendungen mit mehren Sichten.

Den Codegenerator entwickeln

Zum Schluss wollen wir die Kluft zwischen Modell- und Codewelt schließen, indem wir den Codegenerator definieren. Der Generator spezifiziert, wie Informationen aus den Modellen extrahiert und in Code transformiert werden. Der Code wird mit dem Framework verknüpft und in ein fertiges ausführbares Programm kompiliert, ohne dass irgendwelche manuellen Arbeiten erforderlich sind (vgl. [Bat00]). Der erzeugte Code ist einfach ein Nebenprodukt auf dem Weg zum Endprodukt, ähnlich wie .o-Dateien bei der C-Kompilierung.

Der zentrale Punkt bei der Entwicklung eines Codegenerators ist, wie die Konzepte aus dem Modell auf den Code abgebildet werden. Das Domänen-Framework und die Komponenten-Bibliothek können diese Aufgabe dadurch vereinfachen, dass der Abstraktionsgrad auf der Codeseite steigt. Im einfachsten Fall erzeugt jedes Modellsymbol einen bestimmten festen Code, einschließlich der Werte, die in dem Symbol als Argumente eingetragen sind. Der Generator kann aber auch – abhängig von den in einem Symbol eingetragenen Werten und den Beziehungen, die das Symbol zu anderen Symbolen hat – unterschiedlichen Code erzeugen.

Ein einfaches Beispiel einer Generator-Definition für einen „Note"-Dialog zeigt Listing 1. Der Note öffnet einen Dialog mit einer Information, wie z. B. „SET 2004 Registrierung: Willkommen" in Abbildung 1. Die Zeilen 1 und 6 bilden lediglich die Struktur für einen Generator. Zeile 2 erzeugt die Signatur der Funktionsdefinition und Zeile 3 einen Kommentar. Die Namensvergabe für Funktionen (*Function Naming*) basiert auf einem internen Namen, den der Generator erzeugen kann, wenn der Entwickler nicht jedem Symbol einen eigenen Funktionsnamen geben will.

In Zeile 4 wird der Aufruf des Amaretto-Dienstes erzeugt. Dieser benutzt die Modelldaten (im Listing hervorgehoben), wie z. B. den Wert der Texteigenschaft des Note-Benutzungsschnittstellenelements (in diesem Fall „SET 2004 Registrierung: Willkommen"). Ähnlich hat der Modellierer den Wert des Note type aus einer Liste verfügbarer „Note"-Typen ausgewählt, wie z. B. info (hier) oder confirmation (verwendet in Abb. 1 in „Anmeldung gemacht"). Die generierte Zeile 4 sieht daher wie folgt aus (Modelldaten sind hervorgehoben):

```
appuifw.note(u"SET 2004 Registrierung: Willkommen",
   'info')
```

```
1 report '_Note'
2   'def ';   subreport; '_Internal name'; run;   '():'; newline;
3   '# Note '; :Text; newline;
4   '   appuifw.note(u"'; :Text; '", "'; :Note type; '")'; newline;
5   subreport; '_next element'; run;
6 endreport
```

Listing 1: *Codegenerator für einen Note-Dialog*

Da der Codegenerator die Abbildung vom Modell auf die Implementierung automatisiert, realisieren alle Entwickler den Note-Dialog ähnlich: nämlich so, wie ein erfahrener Entwickler ihn definiert hat. Schließlich wird in Zeile 5 eine weitere Generator-Definition aufgerufen, die dem Applikationsfluss zum nächsten Telefondienst oder Benutzungsschnittstellenelement folgt. Dieser Generator mit dem Namen _next element wird – ähnlich wie die „Notes" – ebenfalls von anderen Benutzungsschnittstellenelement verwendet.

Die Generierung beschränkt sich nicht darauf, Dienste, die von Komponenten zur Verfügung gestellt werden, aufzurufen. Die Komponenten können ihrerseits ebenfalls generierten Code aufrufen; darüber hinaus kann generierter Code auch abstrakte Funktionalität von vorhandenem Code erben und diese unter Verwendung von Entwurfsdaten implementieren bzw. konkretisieren. Der Generator kann auch Code für das Applikatons-Framework erzeugen, der zu langweilig zu erzeugen wäre. Dadurch wird die Modellierungsarbeit verringert und Komplexität verborgen. In unserem Telefonbeispiel muss das Drücken des *Cancel*-Knopfes in einem Dialog beispielsweise normalerweise nicht modelliert werden, denn der Generator kann Default-Code erzeugen, der veranlasst, das zum vorherigen Element zurückgesprungen wird.

DSM anwenden

Basierend auf Produktmodellen anstelle von Codemodellen ist mit DSM eine schnellere Entwicklung möglich. Die dargestellte Entwicklung einer Telefonanwendung ist hierfür ein Beispiel. Berichte über Erfahrungen mit DSM aus der Industrie zeigen größere 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 zehn Mal schneller entwickelt (vgl. [Kel00]); bei der Firma Lucent konnte die Produktivität – abhängig vom Produkt – um das drei- bis zehnfache ▶

gesteigert werden (vgl. [Wei99]). 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 beim Code sondern beim Problem selber. Komplexität und Implementierungsdetails können so verborgen werden und eine bereits bekannte Terminologie rückt in den Vordergrund.
- Dank einer einheitlicheren Entwicklungsumgebung und dadurch, dass nicht mehr so viele Wechsel zwischen den verschiedenen Ebenen erforderlich sind, können eine Konsistenz der Produkte und niedrigere Fehlerraten erreicht werden.
- Das Domänenwissen wird für das Entwicklerteam explizit gemacht und findet sich in der Modellierungssprache und deren Werkzeugunterstützung wieder.

Setzt man die Implementierung einer DSM zu den Gesamtprojektkosten in Beziehung, so stellt diese keine zusätzliche Investition dar, sondern spart im Gegenteil Entwicklungsressourcen: Normalerweise arbeiten alle Entwickler mit den Konzepten der Problemdomäne und bilden diese von Hand auf die Implementierungskonzepte ab. Aber unter den Entwicklern gibt es große Unterschiede. Manche erledigen diesen Job besser, manche schlechter. Man sollte also die erfahrenen Entwickler die Konzepte und ihre Abbildung einmal definieren lassen, dann müssen die anderen dies nicht noch einmal erledigen. Wenn ein Experte den Codegenerator spezifiziert, dann erzeugt dieser Generator Applikationen von besserer Qualität, als dies normale Entwickler von Hand erreichen könnten. ■

Übersetzung aus dem Englischen: Kirsten Waldheim.

Literatur & Links

[Bat00] D. Batory, G. Chen, E. Robertson, T. Wang, Design Wizards and Visual Programming Environments for GenVoca Generators, in: IEEE Transactions on Software Engineering, Vol. 26, No. 5, 2000

[Bri96] S. Brinkkemper, K. Lyytinen, R. Welke, Method Engineering – Principles of method construction and tool support, Chapman & Hall, 1996

[Kel00] S. Kelly, J.-P. Tolvanen, Visual domain-specific modeling: Benefits and experiences of using metaCASE tools, International workshop on Model Engineering, ECOOP 2000

[Nok03] Nokia, Series 60 SDK documentation, version 2.0, 2003 (siehe: www.forum.nokia.com/)

[Nok04] Nokia, Python for Series60: API reference, version 1.0, 2004 (siehe: www.forum.nokia.com/)

[Poh00] R. Pohjonen, S. Kelly, Domain-Specific Modeling, in: Dr. Dobb's Journal, August 2002

[Wei99] D. Weiss, C.T.R. Lai, Software Product-line Engineering, Addison Wesley Longman, 1999