Krieg der Welten

Domänenspezifische Modellierung für vollständige Code-Generierung

Juha-Pekka Tolvanen

Domänenspezifische Modellierung hebt das Abstraktionsniveau über das Niveau der Programmierung hinaus, indem die Lösung durch die unmittelbare Nutzung von Domänenkonzepten spezifiziert wird. Die Endprodukte werden aus diesen Spezifikationen auf höchstem Niveau generiert. Dieser Grad der Automatisierung wird dadurch ermöglicht, dass sowohl die Modellierungssprache als auch der Code-Generator nur die Anforderungen einer Firma und Domäne erfüllen müssen. Dieser Artikel stellt die domänenspezifische Modellierung (DSM) vor und erklärt, wie solche Sprachen und Generatoren implementiert werden können. Abschließend wird die DSM mit der MDA verglichen.

UML für Code-Generierung?

Die Generierung von vollständigem Code aus Modellen ist seit vielen Jahren ein Ziel der Industrie. Modelle helfen dabei, das zu lösende Problem besser zu verstehen und zu dokumentieren. Sie können darüber hinaus auch die Eingabe für Code-Generatoren sein. Dies automatisiert die Entwicklung und führt zu verbesserter Produktivität, Qualität und Kapselung von Komplexität.

Bedauerlicherweise basieren viele aktuelle Modellierungssprachen auf der Code-Welt und bieten nur bescheidene Möglichkeiten zur Erhöhung der Abstraktion des Designs und zur Erreichung von vollständiger Code-Generierung. Beispielsweise benutzt UML direkt Programmierkonzepte (Klassen, Rückgabewerte etc.) als Modellkonstrukte. Das Benutzen von rechteckigen Symbolen zur Darstellung von Klassen in einem Diagramm und deren äquivalente Textdarstellung in einer Programmiersprache führen zu keinen echten Generierungsmöglichkeiten – das Niveau der Abstraktion in Modellen und im Code ist dasselbe!

Als Konsequenz daraus finden sich Entwickler beim Erstellen von solchen Modellen zur Beschreibung von Funktionalität und Verhalten wieder, welche einfacher direkt als Code geschrieben werden könnten. Die begrenzten Möglichkeiten zur Code-Generierung zwingen Entwickler nach dem Design zur manuellen Programmierung. Dies führt zu *Roundtrip*-Problemen, denn die gleiche Information an zwei Stellen zu verwalten, nämlich im Code und in Modellen, ist Garant für Probleme.

Domänenspezifische Modellierung als Lösung

Die Herausforderungen der Generierung können in ähnlicher Weise gelöst werden, wie dies in der Vergangenheit mit Programmiersprachen gemacht wurde: indem die Abstraktion weiter erhöht wird. Modelle sollten nicht dazu erdacht werden, um Code zu visualisieren, sondern um übergeordnete Abstraktionen oberhalb von Programmiersprachen zu beschreiben. Gleichermaßen war es in der Vergangenheit besser, zu C zu wechseln und die Abstraktion zu erhöhen, als Assembler-Code zu visualisieren!

Bei der domänenspezifischen Modellierung (DSM) repräsentieren die Elemente des Modells Dinge in der Welt der Domäne, nicht in der Code-Welt. Die Modellierungssprache folgt den Abstraktionen und der Semantik der Domäne und ermöglicht den Modellierern so, direkt mit Domänenkonzepten zu arbeiten. Die Regeln der Domäne können als Beschränkungen in die Sprache integriert werden, was es idealerweise unmöglich macht, ungültige oder unerwünschte Design-Modelle zu spezifizieren. Die passgenaue Ausrichtung der Sprache auf eine bestimmte Problemdomäne bietet etliche Vorteile. Viele von ihnen sind auch bei anderen Wegen in Richtung Abstraktionserhöhung zu finden: verbesserte Produktivität, besseres Verbergen von Komplexität und bessere Systemqualität.

Das höhere Abstraktionsniveau variiert jedoch zwischen Anwendungen und Produkten. Jede Domäne beinhaltet ihre eigenen spezifischen Konzepte und Gültigkeitsbedingungen. Deshalb müssen Modellierungssprachen domänenspezifisch sein. Wenn wir ein Portal für den Vergleich und Verkauf von Versicherungsprodukten entwickeln, warum sollten wir nicht die Versicherungsterminologie direkt in der Modellierungssprache benutzen? Begriffe wie „Risiko“, „Bonus“ und „Schaden“ erfassen die Gegebenheiten von Versicherungen besser, als es Java-Klassen vermögen. Eine versicherungsspezifische Sprache kann auch sicherstellen, dass die modellierten Produkte tatsächlich den Regeln des Versicherungswesens entsprechen: Eine Versicherung ohne eine Prämie ist kein gutes Produkt, also sollte es unmöglich sein, solch ein Produkt zu modellieren.

Diese Domänenkonzepte sind typischerweise schon bekannt und werden benutzt, sind natürlicher und reflektieren schon die für das Produktdesign benötigten darunter liegenden Rechenmodelle. Warum sollten gleichermaßen bei der Entwicklung von Internettelefonie-Diensten nicht schon die relevanten Konzepte und Regeln der Domäne in der Modellierungssprache unterstützt werden? Es ist viel natürlicher bei Telefonservice-Logik mit „Vermittlung“, „Anruf“, „Nachrichtenübermittlung“ etc. zu arbeiten als mit Code. Der Code (Assembler, 3GL, objektorientiert etc.) kann schließlich nach wie vor aus diesen High-Level-Spezifikationen generiert werden. Entscheidend für eine erfolgreiche automatische Code-Generierung aus Modellen ist, dass sowohl Sprache als auch Generator nur den Ansprüchen einer Firma genügen müssen.

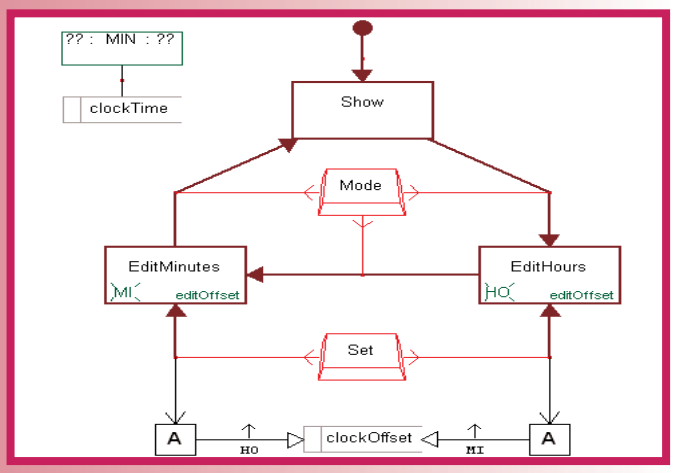

Abb. 1a: DSM zum Modell von Uhrenanwendungen

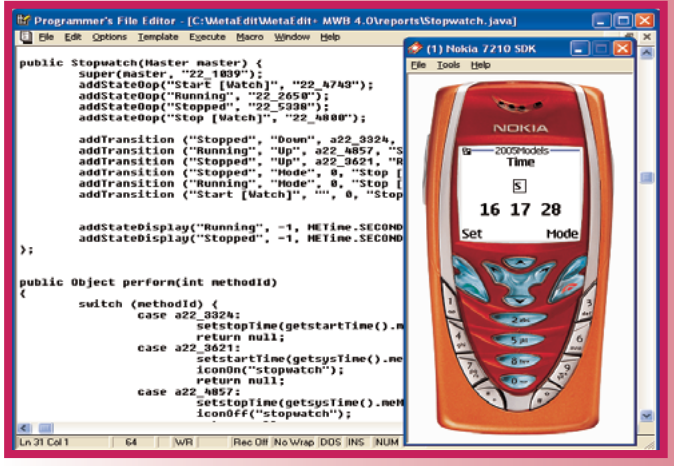

Abb. 1b: Generierter Code und dessen Ausführung auf dem Zielgerät

Ein Beispiel für eine DSM-Sprache

Um DSM detaillierter zu veranschaulichen, nehmen wir uns eine Anwendungsdomäne, die jedem bekannt ist: Anwendungen von digitalen Armbanduhren, wie Zeit, Stoppuhr und Weltzeit. Diese Anwendungen sind in Mobiltelefonen mittels MIDP implementiert, einem Satz von Java APIs, die eine standardisierte Laufzeitumgebung für mobile Informationsgeräte bereitstellen.

Bevor irgendwelche neuen Features implementiert werden können, müssen Entwickler diese in der Uhrendomäne modellieren. Dies beinhaltet das Anwenden der Terminologie und Regeln der Uhr, wie Knöpfe, Alarme, Anzeige-Symbole, Zustände und Benutzeraktionen. Die DSM-Sprache wendet genau dieselben Konzepte direkt in der Modellierungssprache an. Die Benutzung bekannter Terminologie führt zu Design-Modellen, die einfacher zu lesen, zu merken, zu validieren und zu warten sind. Ein einfaches Beispiel solch einer Modellierungssprache ist in Abbildung 1a dargestellt. Das Modell repräsentiert das Feature zur Zeiteinstellung: die Aktionen, die der Nutzer über die Betätigung von Knöpfen durchführen kann, die blinkenden Anzeige-Elemente und die Aktionen, die die Zeit verändern.

Die Modellierungssprache, hier basierend auf Zustandsautomaten, beinhaltet auch Domänenregeln, die verhindern, dass Entwickler ungültige Modelle erstellen (z. B. verbinden zweier Uhrenknöpfe). Ein deutlicher Hinweis auf das höhere Abstraktionsniveau ist, dass die Modellierungssprache auf den Regeln der Uhr operiert und diese anwendet, nicht auf der Implementierung. Es sind weder MIDP-Konzepte noch Java-Code im Modell zu sehen! Nach der Erstellung des Modells kann der Entwickler einen Generator starten und die Anwendung auf dem Zielgerät ausführen (Abbildung 1b). Das komplette Beispiel können Sie über [MetaCase] erhalten.

Wie DSM implementiert wird

Um die Vorteile der DSM wie verbesserte Produktivität, Qualität und Kapselung der Komplexität zu erreichen, benötigen wir eine DSM-Sprache und Generatoren. In der Vergangenheit hätten zudem die unterstützenden Werkzeuge implementiert werden müssen. Dies war einer der Hauptgründe dafür, dass sich DSM nicht so schnell durchgesetzt hatte. Schließlich zählt das Implementieren von CASE-Werkzeugen kaum zu den Kernkompetenzen der meisten Firmen.

Seit offene, metamodell-basierte Werkzeuge zur Modellierung und Code-Generierung verfügbar sind, beschränkt sich die erforderliche Arbeit auf das Definieren der Sprache und der Generatoren. Diese Werkzeuge ermöglichen die Bereitstellung einer arbeitsfähigen DSM-Umgebung innerhalb von Tagen anstatt Monaten wie im Falle von Werkzeugen, die zusätzliche manuelle Programmierung erfordern. Noch entscheidender ist, dass die Entwicklungsumgebung auf Änderungen in der Domäne reagieren kann: Diese Werkzeuge können Modelle verwalten, die synchronisiert mit Sprache und Generatorentwicklung sind, was die Design-Arbeit in echten Industrieumgebungen sicher macht.

Die Modellierungssprache für eine Domäne definieren

Das Definieren einer Modellierungssprache beinhaltet drei Aspekte:

- die Domänenkonzepte,
- die zur Darstellung dieser in grafischen Modellen benutzten Notation und
- die Regeln, welche den Modellierungsprozess steuern.

Im Großen und Ganzen wird das Definieren einer kompletten Sprache als schwierige Aufgabe angesehen. Das trifft sicherlich zu, wenn man eine Sprache für jedermann erstellen will. Die Aufgabe vereinfacht sich erheblich, wenn man sie nur für eine Problemdomäne in einer Firma erstellt.

Das Hauptaugenmerk beim Finden von Domänenkonzepten liegt auf dem Fachwissen eines oder eines kleinen Teams von Domänenexperten. Typischerweise ist der Experte ein erfahrener Entwickler, der schon einige Produkte in dieser Domäne entwickelt hat. Er kann die Architektur hinter den Produkten entwickelt haben oder für die Zusammenstellung der Komponentenbibliothek verantwortlich gewesen sein. Er kann problemlos Domänenkonzepte aus ihrer Terminologie, existierenden Systembeschreibungen und Komponentendiensten ableiten.

In unserem Beispiel mit der Uhrenanwendung sind die Domänenkonzepte hergeleitet aus der visuellen Anzeige (z. B. Zeiteinheiten, Symbole etc.), der Kontrollstruktur (Nutzer drückt Knopf, Alarm wird ausgelöst etc.) und darunter liegenden Diensten (d. h. Zeit und ihre Manipulation, Alarm). Indem diese Konzepte in die Modellierungssprache eingebracht und weiter verfeinert werden, erzeugen wir die Spezifikation der Sprache (d. h. das Metamodell). Das Ziel ist hier, die gewählten Konzepte akkurat auf die Semantik der Domäne abzubilden.

Allerdings bilden reine Domänenkonzepte allein noch keine Modellierungssprache. Wir benötigen vielmehr Domänenwissen darüber, wie sie zusammengestellt werden können. Hier wird die Semantik mit Hilfe von Zustandsautomaten angereichert und beschränkt, um sich am Konzept der Uhrendomäne auszurichten. Beispielswei-

se können Zustandsübergänge vom Nutzer nur ausgelöst werden, wenn ein bestimmter Knopf gedrückt wird. Die stattfindenden Aktionen während des Übergangs können nur mit Entitäten von Zeiteinheiten arbeiten. Zudem ist der Satz möglicher Operationen beschränkt: Man kann nur Zeiteinheiten hinzufügen oder entfernen bzw. sie hoch- oder herunterzählen. Es muss betont werden, dass zukünftige weitergehende Anforderungen leicht realisiert werden können. Hierzu muss einfach der Satz möglicher Operationen bzw. der Satz verarbeitbarer Entitätstypen erweitert werden.

Um schlussendlich den Notationsteil der Modellierungssprache bereitzustellen, definieren wir Symbole als grafische Repräsentationen der Modellierungskonzepte. Da unsere Beispieldomäne Benutzungsoberflächen-bezogen ist, können wir viele ihrer Symbole auf dem visuellen Erscheinungsbild von Uhren basieren lassen.

Definieren des Domänen-Frameworks

Das Domänen-Framework stellt die Schnittstelle zwischen generiertem Code und der darunter liegenden Plattform bereit. In einigen Fällen wird kein weiterer Framework-Code benötigt: Der generierte Code kann direkt die Plattform-Komponenten aufrufen und deren Dienste sind ausreichend. Häufig lässt sich die Code-Generierung jedoch durch die Bereitstellung zusätzlichen Framework-Utility-Codes oder zusätzlicher Komponenten vereinfachen. Solche Komponenten können schon aus früheren Entwicklungsbemühungen und Produkten existieren und nur ein wenig Anpassung benötigen.

In unserem Uhrenbeispiel brauchen wir zwei grundlegende Komponenten: ein Template zur Bereitstellung der Benutzungsoberfläche für das Midlet und ein Template für Zustandsautomaten, die die Unteranwendungen definieren. Beide können als abstrakte Klassen implementiert werden, die bei Ableitung konkreter Klassen erweitert werden. Beispielsweise wird die abstrakte Klasse für den Zustandsautomaten die Datenstrukturen zur Definition des Zustandsautomaten beinhalten und die abgeleitete Klasse den Code zum Laden der Definitionsdaten. Die konkrete Klasse implementiert zudem die Methoden, welche während der Zustandsübergänge ausgeführt werden.

Entwickeln des Code-Generators

Schlussendlich wollen wir die Lücke zwischen dem Modell und der Quelltext-Welt durch das Definieren eines Code-Generators schließen. Der Generator spezifiziert, wie Information aus den Modellen entnommen und in Quellcode transformiert wird. Die Modellierungssprache mit ihren Konzepten, ihrer Semantik und Regeln und die Eingabe-Syntax, die die Zielplattform erfordert, bestimmen und lenken diesen Prozess. Um benutzbar zu sein, muss der Generierungsprozess vollständig sein: Vollständiger Quellcode wird aus der Modellierungssicht generiert und manuelles Umschreiben des Codes ist unnötig. Diese Vollständigkeit ist auch der Eckpfeiler anderer erfolgreicher Veränderungen bei Programmiersprachen.

Das Hauptaugenmerk beim Bau eines Code-Generators liegt darauf, wie die Modellkonzepte im Quellcode abgebildet werden. Das Domänen-Framework und die Komponentenbibliothek können diese Aufgabe vereinfachen, indem sie den Abstraktionsgrad auf Quellcode-Seite erhöhen. In den einfachsten Fällen erzeugt jedes Modellierungssymbol einen bestimmten festgelegten Code inklusive der Werte, die den Symbolen als Argumente mitgegeben wurden. Der Generator kann auch verschiedenen Code generieren, abhängig von den Werten im Symbol, den Beziehungen zu anderen Symbolen oder anderer Informationen im Modell.

Ein Beispiel für automatisch generierten Code zeigt Listing 1. Es veranschaulicht die Implementierung des Zustandsautomaten aus Abbildung 1. Listing 1 zeigt, dass die Komplexität des Zustandsautomaten vor dem Generator verborgen wird, indem sein grundlegendes Verhalten als abstrakte Framework-Klasse implementiert wird. Der konkrete Zustandsautomat wird dann als Subklasse von dieser abstrakten Klasse abgeleitet (Zeile 5) und mit Daten aus den Design-Modellen im Klassenkonstruktor initialisiert (Zeilen 13 – 30). Dies ist ein Beispiel für die Implementierung eines Framework-Gegenstückes zu einem logischen Modell-Konstrukt.

Die Zeilen 33 – 46 zeigen, was für Code für Verhaltensaspekte der Anwendung generiert wird. Für jeden Zustandsübergang existiert üblicherweise ein Satz von Aktionen, der während des Übergangs ausgeführt wird. In unserer Uhrensprache sind diese Aktionen beschränkt auf die Abdeckung der grundlegenden Zeiteinheiten-Arithmetik. Damit bleiben drei Operationen übrig, mit denen wir uns beschäftigen müssen: plus, minus und durchlaufen. Diese Operationen sind als primitive Dienste in unserem Framework implementiert. Wenn solch ein Dienst benötigt wird, erzeugt der Code-Generator nur einen Aufruf dafür (z. B. wie in Zeile 37 oder Zeile 40).

Wie schon angesprochen, beschreiben domänenspezifische Modelle die Anwendungsfunktionalität in vom Quellcode unabhängiger Weise auf einem höheren Abstraktionsniveau. Deshalb erlauben es uns die gleichen Modelle, Code für verschiedene Plattformen zu generieren. Zum Beispiel kann aus den gleichen Modellen C-Code generiert werden: nur der Generator ist verschieden, nicht das Anwendungsmodell.

Wie das Beispiel veranschaulicht, funktioniert DSM nur, weil Modellierungssprache und Code-Generator einer einzigen Domäne gewidmet sind. Dadurch kann modellbasierte Code-Generierung vollständig und der generierte Code effizient sein. Selbst wenn sich herausstellt, dass der Code nicht effizient ist, kann dies höchstwahrscheinlich auch direkt über den Generator korrigiert werden.

Wie unterscheidet sich DSM von MDA?

Nachdem wir mit Beispielen die Hauptprinzipien von domänenspezifischer Modellierung demonstriert haben, können wir sie nun mit anderen gängigen modellbasierten Entwicklungsansätzen vergleichen und speziell mit der *Model Driven Architecture* (MDA) der OMG. Ganz grundsätzlich beinhaltet MDA die Umwandlung von UML-Modellen auf einem höheren Abstraktionsniveau in UML-Modelle auf einem niedrigeren Abstraktionsniveau. Gewöhnlich sind dies zwei Ebenen, und zwar plattformunabhängige Modelle (PIMs) und plattformspezifische Modelle (PSMs). Diese PIMs und PSMs sind reines UML und bieten daher keine wirkliche Erhöhung der Abstraktion.

In der MDA werden auf jeder Stufe die Modelle einer detaillierteren Bearbeitung mit *Reverse* und *Roundtrip Engineering* unterzogen und am Ende wird wesentlicher Code aus dem finalen Modell generiert. Die OMG verfolgt mit MDA das Ziel, dasselbe PIM auf verschiedenen Software-Plattformen nutzen zu können. Außerdem will die OMG alle Übersetzungen und Modellformate standardisieren, sodass die Modelle zwischen den Werkzeugen verschiedener Hersteller austauschbar werden. Dies erreichen zu wollen, ist sehr ambitioniert, aber man ist noch viele Jahre davon entfernt. Diese Zielsetzung zeigt jedoch klar die Unterschiede zwischen DSM und MDA und beantwortet die Frage, wann welcher Ansatz anzuwenden ist.

DSM erfordert Domänen-Fachwissen, ein Potenzial, das eine Firma nur erreichen kann, wenn sie kontinuierlich in der gleichen Problemdomäne arbeitet. Dies sind typischerweise eher Produkt- oder Systementwicklungshäuser als Projekthäuser. Hier ist die Plattformunabhängigkeit keine wichtige Anforde-

```
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 1: Aus Zustandsdiagramm von Abbildung 1 generierter Code

rung, obwohl sie mit DSM einfach erreicht werden kann, indem verschiedene Code-Generatoren für verschiedene Software- und/oder Produktplattformen eingesetzt werden. Stattdessen liegt das Hauptaugenmerk der DSM auf der signifikanten Verbesserung der Entwicklerproduktivität.

Mit der MDA richtet die OMG ihren Fokus nicht auf die Nutzung von DSM-Sprachen, sondern auf generisches UML, ihre eigene Standard-Modellierungssprache. Sie versucht nicht, das möglicherweise existierende Domänen-Fachwissen einer Firma einzukapseln, sondern nimmt an, dass dieses Wissen nicht vorhanden oder irrelevant ist. Es scheint daher, dass die MDA, wenn die OMG letztlich ihre selbstgesteckten Ziele erreicht, geeignet für System- oder Anwendungsintegrationsprojekte ist.

MDA setzt ein fundiertes Wissen ihrer Methodik voraus, welches nicht per se in einer Firma vorhanden ist und durch Erfahrung erlangt werden muss. Folglich muss das MDA-Fachwissen angeeignet oder von außerhalb eingekauft werden, wohingegen das Domänen-Fachwissen für die DSM schon in einer Organisation verfügbar ist und angewendet wird. In diesen Situationen ist die Wahl zwischen MDA und DSM oft klar.

DSM nutzen

Domänenspezifische Modellierung erlaubt schnellere Entwicklung basierend auf den Modellen der Problemdomäne und weniger auf Modellen von Quellcode. Unser Uhrenbeispiel veranschaulichte dies schnell. Industrielle Erfahrungen mit DSM [DSM] zeigen bedeutende Verbesserungen der Produktivität, niedrigere Entwicklungskosten und bessere Qualität. Das Unternehmen Nokia gibt beispielsweise an, dass sich die Entwicklung von Mobiltelefonen auf diesem Weg um den Faktor 10 beschleunigt. Bei der Firma Lucent konnte die Produktivität – abhängig vom Produkt – um das drei- bis zehnfache gesteigert werden. Die Schlüsselfaktoren dafür sind:

- Das Problem wird nur einmal – und zwar auf einem hohen Abstraktionsniveau – gelöst und der lauffähige Quellcode wird geradewegs aus dieser Lösung generiert.
- Das Hauptaugenmerk der Entwickler liegt nicht länger auf dem Code, sondern beim Modell, dem Problem an sich. Komplexität und Implementierungsdetails können so verborgen werden und eine bereits bekannte Terminologie rückt in den Vordergrund.
- Durch eine einheitlichere Entwicklungsumgebung und dadurch, dass weniger Wechsel zwischen den Abstraktionsniveaus Modell und Implementierung erforderlich sind, lassen sich eine bessere Konsistenz der verschiedenen Produkte und niedrigere Fehlerraten erreichen.
- Das Domänenwissen wird für das Entwicklungsteam explizit gemacht, indem es in der Modellierungssprache und deren Werkzeugunterstützung festgehalten wird.

Der Einsatz von DSM bedeutet keine zusätzliche Investition, wenn der gesamte Zyklus vom Design bis zum arbeitenden Code betrachtet wird. Vielmehr spart es Entwicklungsressourcen: Traditionell 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 diese Aufgabe besser, manche schlechter. Also lasst die erfahrenen Entwickler die Konzepte und deren Abbildung einmal definieren, dann müssen die anderen dies nicht erneut tun. Spezifiziert ein Experte den Code-Generator, so produziert dieser Anwendungen von besserer Qualität, als es normale Entwickler von Hand könnten.

Literatur und Links

[Fow] M. Fowler, Language Workbenches: The Killer-App for Domain Specific Languages?, `http://martinfowler.com/articles/languageWorkbench.html`

[DSM] DSMforum sammelt Fallstudien zum Einsatz der domänenspezifischen Modellierung, `http://www.DSMForum.org`

[Gree04] Greenfield et al, Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, Wiley, 2004

[MetaCase] das vollständige Beispiel der digitalen Armbanduhranwendung können Sie vom Autor (`jpt@metacase.com`) oder über die Firmenwebseite erhalten, `www.metacase.com/download`

[MIDP] JSR-000118 Mobile Information Device Profile, Final Release, `http://jcp.org/en/jsr/detail?id=118`

[OMG] Object Management Group, `http://www.omg.org`

[PoKe02] R. Pohjonen, S. Kelly, Domain-Specific Modeling, in: Dr. Dobb's Journal, August, 2002

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