

Domänenspezifisch entwickeln

Modellcode

Eine vollständig modellbasierte Codegenerierung ist möglich, falls sowohl die Modellierungssprache als auch der Codegenerator an die Anforderungen nur eines Unternehmens und einer Domäne angepasst sind.

Die Modellierung soll den Schwerpunkt von der Implementierung auf das Design verlagern. Modelle dienen dem besseren Verständnis von Vorgängen, können jedoch auch als Eingabe für Codegeneratoren fungieren. Dadurch wird die Entwicklung automatisiert, wodurch sich eine höhere Produktivität und Qualität einstellen und Komplexität besser verborgen werden kann. Viele momentan verfügbare Modellierungssprachen basieren jedoch auf der Implementierungswelt und bieten nur begrenzte Möglichkeiten, die Abstraktionsstufe des Designs zu erhöhen und eine vollständige Codegenerierung zu erzielen.

UML verwendet beispielsweise Programmierkonzepte (Klassen, Rückgabewerte usw.) direkt als Modellierkonstrukte. Es sind keine echten Generierungsmöglichkeiten damit verbunden, ein Rechtecksymbol zur Abbildung einer Klasse in einem Diagramm zu verwenden und daraus eine entsprechende textbasierte Darstellung in einer Programmiersprache herzuleiten. Die Abstraktionsstufe der Modelle und des Codes sind vielmehr identisch! Da-

her müssen Entwickler oft Modelle zur Beschreibung des Verhaltens und der Funktion einer Software erstellen, die sie einfacher gleich direkt als Programmcode verfassen könnten. Beschränkte Möglichkeiten zur Codegenerierung zwin-

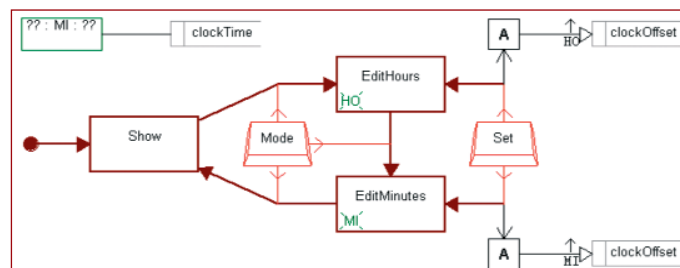


Bild 1: State Machine mit Erweiterungen für Uhrendomäne

gen Entwickler, nach der Designphase mit manueller Programmierung zu beginnen. Dies widerspricht der Idee einer umfassenden Codegenerierung und schränkt mögliche Produktivitätsvorteile ein. Es entstehen auch Round-Trip-Probleme: Falls sie stärker codeorientiert sind, visualisieren Modelle lediglich statische Strukturen. Bei einer stärkeren Modellorientierung wird viel Energie auf die Umformulierung des generierten Codes und die Aktualisierung derjenigen Modelle verwandt, bei denen es sich nicht um Klassendiagramme handelt. Dieselben Informationen gleichzeitig an zwei Orten zu pflegen (Code und Modelle), bringt fast zwangsläufig Schwierigkeiten mit sich. Die Herausforderungen

der Codegenerierung lassen sich auf ähnliche Weise lösen wie in der Vergangenheit bei Programmiersprachen: durch eine erneut höhere Abstraktion. Modelle sollten Code nicht visualisieren, sie sollten vielmehr stärker als Programmiersprachen abstrahieren. Entsprechend war es günstiger, auf C zu wechseln und dadurch eine höhere Abstraktionsstufe zu erreichen, als Assembler-Code zu visualisieren! Die höhere Abstraktion unterscheidet sich jedoch zwischen Anwendungen und Produkten. Jede Domäne beinhaltet ihre eigenen spezifischen Konzepte und Korrektheitsbedingungen. Daher müssen Modellierungssprachen spezifisch an jede Domäne angepasst sein. Domänenspezifische Modellie-

Sprachkommunikationssystemen wesentlich enger an der Anwendungsdomäne als Assembler-Mnemoniks. Spezifikationssprachen höherer Ebenen verwenden diese domänenspezifischen Konzepte direkt als Modellierkonstrukte. Da die Sprache den Abstraktionen und der Semantik der Domäne folgt, entsteht für Entwickler der Eindruck, direkt mit Domänenkonzepten zu arbeiten. Typischerweise werden diese Domänenkonzepte bereits genutzt. Sie sind bekannt, natürlicher und spiegeln bereits die zu Grunde liegenden Computermodelle wider, die für den Entwurf der Produkte benötigt werden. Der Code (Assembler, 3GL, objektorientiert usw.) lässt sich trotzdem aus diesen stärker abstrahierten Spezifikationen generieren. Der Grundstein einer automatisierten Codegenerierung mit Modellen besteht in einer ausschließlichen Übereinstimmung zwischen den Anforderungen eines einzelnen Unternehmens sowie der Sprache und der Generatoren.

Seit kurzem stehen offene und benutzeranpassbare Technologien zur Verfügung, die es Entwicklern ermöglichen, sowohl Design-Sprachen als auch Codegeneratoren auszutauschen, um diese den unterschiedlichen Anforderungen der Softwareentwicklung anzupassen. Erfahrene Entwickler können in einem Unternehmen daher die Design-Sprachen und Generatoren an eine spezifische Domäne anpassen. Damit lassen sich die tatsächlichen Produkte mit domänenspezifischen Sprachen entwerfen und direkt aus den Modellen generieren.

Am Beispiel der Entwicklung einer digitalen Armbanduhr mit Funktionen wie Stoppuhr oder Weltzeit lässt sich die Vorgehensweise zeigen. Vor der Implementierung von Funktionen müssen die Ent-

Dr. Juha-Pekka Tolvanen

ist CEO von MetaCase

wickler diese in der Uhren-domäne entwerfen. Dazu müssen sie die Vorgaben und Regeln der Uhr, wie z.B. Schalter, Alarm, Anzeigesymbole, Zustände und Benutzeraktionen verwenden. Die domänenspezifische Methode verwendet exakt dieselben Konzepte direkt in der Modellierungssprache. Als Beispiel stellt das Modell in Bild 1 die Zeiteinstellungsfunktion dar: Aktionen, die ein Benutzer durch Drücken von Schaltern vornehmen kann, blinkende Anzeigeelemente und Aktionen zur Veränderung der Zeit. Dementsprechend beginnt die Definition der Sprache mit dem Identifizieren der Terminologie und der Konzepte der Modellierungssprache (z.B. Schalter, Symbol, usw.).

Diese werden formal in einem Metamodell dokumentiert, das sich in metamodellbasierten Modellierungswerkzeugen umsetzen lässt. Zusätzlich zu der Terminologie folgt die domänenspezifische Modellierungssprache einem Computermodell, das im genannten Beispiel die Statemachine ist – ein typisches, mit eingebetteter Software verwendetes Computermodell. Die Ebene, auf der die Statemachine betrieben wird, und die Regeln, die sie verwendet, stellen jedoch keine Implementierung sondern Produktregeln dar. Da sich das Modell auf einer höheren Ebene befindet, ist eine Modellierung in wesentlich geringerem Umfang als in Sprachen notwendig, die auf eine Visualisierung von

Code ausgerichtet sind. Trotzdem ist die Modellierung gerade so umfassend, dass Codegeneratoren mit ihrer Hilfe den notwendigen Code generieren können. Im letzten Schritt der Spracherstellung erweitern wir die Semantik der Statemachine und schränken sie ein, um uns auf das Konzept der Uhrendomäne zu konzentrieren. In unserem Beispiel existieren in der Statemachine zwei uhrenspezifische Erweiterungen: Übergänge sind nur dadurch auslösbar, dass ein Benutzer einen bestimmten Schalter drückt, und im Laufe der Übergänge vorgenommene Aktionen finden nur auf Zeiteinheiten Anwendung. Auch die möglichen Operationen sind begrenzt: Zeiteinheiten lassen sich ad-

dieren, subtrahieren oder herauf- bzw. herunterzählen. Die möglichen Operationen lassen sich jedoch einfach erweitern oder neue Entitätsarten für Operationen definieren, falls dies später notwendig werden sollte. Mit den oben genannten grundlegenden Operationen können Entwickler in unserem Beispiel jedoch alle momentanen Erfordernisse der Uhrenfamilie abdecken und die Entwicklung mit Codegeneratoren automatisieren.

Der Generator legt fest, wie Informationen aus den Modellen gewonnen und in Code umgesetzt werden. Dieser Prozess hängt von der Modellierungssprache und ihren Konzepten, ihrer Semantik, den Regeln und der für die Zielplattform benötigten Eingabesyntax ab. Damit das Ergebnis verwertbar ist, muss der Generierungsprozess vollständig sein: Aus der Sicht des Anwendungsentwicklers muss vollständiger Code erzeugt werden, ohne dass manuelles Umschreiben notwendig ist. Diese Vollständigkeit war der Grundstein anderer, erfolgreich mit Programmiersprachen erzielter Umbrüche. Haben Sie jemals einen Entwickler gesehen, der Assembler-Code manuell editiert und dann versucht, seinen C-Code damit in Einklang zu halten? Entsprechend sollte der generierte Code schlichtweg ein Nebenprodukt auf dem Weg zum fertigen Produkt sein, wie z.B. ».o«-Dateien bei einer C-Kompilierung.

Eine solche vollständige Codegenerierung ist schwierig, falls nicht sogar unmöglich, wenn Generator und Modellierungssprache fast alle Situationen abdecken sollen. Sie ist jedoch möglich, wenn beide lediglich die Anforderungen eines Unternehmens erfüllen müssen. Dies bedeutet, dass der Codegenerator mit der Modellierungssprache, von der er seine Einga-

```

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, MTime.MINUTE, d22_977);
28         addStateDisplay("EditHours", MTime.HOUR_OF_DAY, MTime.MINUTE, d22_977);
29         addStateDisplay("EditMinutes", MTime.MINUTE, MTime.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(MTime.HOUR_OF_DAY, true, displayTime());
38                 return null;
39             case a22_1405:
40                 getClockOffset().roll(MTime.MINUTE, true, displayTime());
41                 return null;
42             case d22_977:
43                 return getClockTime();
44         }
45         return null;
46     }
47 }

```

Listing 1: Aus dem Zustandsdiagramm in Bild 1 generierter Code

ben erhält, und mit der Zielplattform, auf der der generierte Code ablaufen wird, gut zusammenarbeiten muss. Modelle liefern die Eingaben für den Codeerstellungsprozess, die tatsächliche Navigation und der Abruf der Design-Informationen wird jedoch entsprechend dem Metamodell durchgeführt. Deshalb sollte das Metamodell ein Computermodell sein, das mit dem entwickelten Produkt harmoniert. In den meisten Fällen werden für Domänen oder zur Codegenerierung einige Veränderungen oder Erweiterungen des Grundmodells notwendig sein, um sicher zu stellen, dass die Modelle alle wichtigen statischen Merkmale und Verhaltensaspekte des Produkts als Eingaben des Codegenerators erfassen können. In unserem Uhrenbeispiel wählten wir die State-machine als Computermodell und erweiterten sie dann um Konzepte wie Zeiteinheiten, um die Anforderungen der Domäne und der Codegenerierung zu erfüllen. Eine Plattform stellt eine wohl definierte Menge an Dienstleistungen zur Verfügung, die dem Codegenerator als Schnittstelle dienen: Der generierte Code kann z.B. direkt die Komponenten der Plattform und ihre Dienste aufrufen. Oft ist es jedoch vorteilhaft, die Codegenerierung durch die Definition von zusätzlichem Framework-Utility-Code oder Komponenten zu vereinfachen. Ein solches Framework ist in Form von Bibliotheken, Komponenten und Code-Templates auf die Plattform aufsetzbar. Dabei ist das Framework nicht unbedingt eine zusätzliche, nur vom Codegenerator benötigte Bürde. Vielmehr verwendet die zugrunde liegende Softwarearchitektur in den meisten Fällen bereits zahlreiche Bibliotheken, Komponenten oder sonstige wieder verwendbare Be-

standteile, die ebenfalls die Codegenerierung unterstützen.

Der wichtigste Punkt bei der Erstellung eines Codegenerators ist die Art und Weise, wie die Modellkonzepte in Code umgesetzt werden. Die Ausgabe wird nicht als wirklicher Code definiert, sondern eher als ein Beispiel oder ein Template. In den einfachsten Fällen ergibt jedes Modellierungssymbol einen bestimmten, fest vorgegebenen Code, der die als Argumente in die Symbole eingegebenen Werte enthält. In Abhängigkeit von den Werten innerhalb des Symbols, den Beziehungen zu anderen Symbolen oder anderen Informationen des Modells kann der Generator jedoch auch unterschiedlichen Code generieren.

Die Generatordefinition sollte so direkt und einfach wie möglich sein. Dies ist möglich, indem innerhalb des Generators keine Variationen oder Implementierungsdetails verwendet wer-

den. Das Framework und die Komponentenbibliothek können durch eine höhere Abstraktion im Codebereich vereinfachend wirken. Die Generatordefinition wird auch durch domänenspezifische Modelle mit Korrektheitsbedingungen erleichtert, denn der Generator muss in diesem Fall die Korrektheit der Eingabe nicht überprüfen. Eine ordnungsmäßige Modularisierung und Wiederverwendung sind bei der Erstellung des Codegenerators sehr hilfreich. Falls der Entwickler die Variationenbehandlung für unterschiedliche Zielplattformen z.B. in eigenständige Module isoliert, lässt sich die Anzahl der unterstützten Plattformen sehr einfach erweitern, da dann für das Hinzufügen einer Plattform lediglich neue Versionen dieser Module notwendig sind und kein komplett neuer Generator.

Doch wie sind die unterschiedlichen Bestandteile einer Codegenerierungs-Lösung für eine 100%-ige Co-

degenerierung zusammenzufügen? Listing 1 zeigt den generierten Code für die in Bild 1 dargestellte State-machine. Die Implementierung ihres elementaren Verhaltens als abstrakte Framework-Klasse verbirgt die Komplexität der State-machine vor dem Generator. Die konkrete State-machine ist dann eine Unterklasse dieser abstrakten Klasse (Zeile 5). Sie wird im Klassenkonstruktor mit den Daten aus den Design-Modellen initialisiert (Zeilen 13 bis 30). Hierbei handelt es sich um ein Beispiel, wie sich ein Framework-Gegenstück für logische Modellkonstrukte implementieren lässt.

Zeilen 33 bis 46 sind ein Beispiel für Code, der für Verhaltensaspekte einer Anwendung generiert wird. Bei jedem Zustandsübergang werden in der Regel im Laufe des Übergangs einige Aktionen ausgeführt. In unserer Uhrensprache sind diese Aktionen auf grundlegende Berechnungen mit Zeiteinheiten beschränkt. Im Ergebnis müssen wir noch die Operationen Plus, Minus und Zählen behandeln, diese sind als einfache Dienste in unserem Framework implementiert. Falls ein solcher Dienst benötigt wird, erstellt der Codegenerator lediglich einen Aufruf auf ihn (z.B. Zeile 37 oder Zeile 40).

Wie bereits erwähnt, beschreiben domänenspezifische Modelle die Funktionen einer Anwendung codeunabhängig auf einer höheren Abstraktionsstufe. Daher können wir für unterschiedliche Plattformen mit denselben Modellen Code generieren. C-Code ließe sich beispielsweise aus denselben Designs heraus erstellen: Lediglich der Generator unterscheidet sich, nicht jedoch das Design der Anwendung. (cg)

Schnelles Design mit Produktmodellen

Basierend auf Produkt- anstatt auf Code-Modellen ermöglicht die domänenspezifische Modellierung eine schnellere Entwicklung. In der Praxis ergeben sich wesentliche Produktivitätsfortschritte, niedrigere Entwicklungskosten und höhere Qualität. So berichtet Nokia, auf diese Weise Mobilfunktelefone bis zu zehnmal schneller als bisher entwickeln zu können, und Lucent spricht von drei- bis zehnfach höherer Produktivität. Die wesentlichen dazu beitragenden Faktoren sind:

- Das Problem wird lediglich einmal auf hoher Abstraktionsstufe gelöst, und der endgültige Code direkt aus dieser Lösung generiert.
- Die Aufmerksamkeit des Entwicklers wechselt von der Codeerstellung auf das Design und damit auf das Problem selbst. Komplexität und Implementierungsdetails treten in den Hintergrund, und die bereits bekannte Terminologie wird betont.
- Stärkere Einheitlichkeit der Entwicklungsumgebung und ein seltener Wechsel zwischen den Ebenen Design und Implementierung führen zu konsistenten Produkten und niedrigen Fehleraten.
- Das Domänenwissen wird dem Entwicklungsteam verdeutlicht und in der Modellierungssprache sowie ihrer Tool-Unterstützung erfasst.

Die Implementierung einer domänenspezifischen Modellierung und Codegenerierung stellt keine zusätzliche Investition dar, sondern spart vielmehr Entwicklungsressourcen: Traditionell arbeiten alle Entwickler mit Domänenkonzepten und setzen sie manuell auf Implementierungskonzepte um. Manchen gelingt dies besser, anderen weniger gut. Daher sollten erfahrene Entwickler die Konzepte definieren und einmal umsetzen, sodass andere Mitarbeiter nicht erneut dazu gezwungen sind. Falls ein Experte den Codegenerator spezifiziert, erstellt der Generator qualitativ hochwertigere Anwendungen, als es normalen Entwicklern auf manuelle Art und Weise möglich ist.

MetaCase

Telefon 0 03 58/14 44 51 40 3
Fax 0 03 58/14 44 51 40 5