MD^2-DSL — eine domänenspezifische Sprache zur Beschreibung und Generierung mobiler Anwendungen

Henning Heitkötter, Tim A. Majchrzak, Herbert Kuchen

Institut für Wirtschaftsinformatik
Universität Münster
Leonardo-Campus 3, 48149 Münster
{heitkoetter,tima,kuchen}@ercis.de

Abstract: Entwickler mobiler Anwendungen, sogenannter Apps, stehen einer heterogenen Landschaft an mobilen Plattformen gegenüber, die sich hinsichtlich ihrer Programmierung stark unterscheiden. Häufig sollen zumindest die weit verbreiteten Betriebssysteme iOS und Android unterstützt werden. Dabei sollen Apps dem nativen, plattformtypischen Aussehen und Verhalten genügen oder zumindest nachempfunden werden. Bestehende Cross-Plattform-Lösungen im mobilen Umfeld unterstützen letztere Anforderung nicht, so dass Entwickler oftmals gezwungen sind, dieselbe App parallel für mehrere Plattformen nativ zu entwickeln. Dies erhöht den Entwicklungsaufwand erheblich, zumal die Implementierung auf einem niedrigen Abstraktionsniveau erfolgt. Dieser Beitrag stellt das Framework MD^2 und die domänenspezifische Sprache MD^2-DSL vor, die es ermöglicht, Apps mit vorwiegend geschäftlichem Hintergrund auf einem gehobenen Abstraktionsniveau prägnant zu beschreiben. Der modellgetriebene Ansatz MD^2 generiert aus den textuellen MD^2-DSL-Modellen iOS- und Android-Apps. Die Sprache MD^2-DSL enthält Konstrukte zur Abbildung und Umsetzung typischer Anforderungen an Apps und stellt somit eine Alternative zur wiederholten Implementierung einer App auf verschiedenen Plattformen dar.

1 Einführung

Mit der steigenden Beliebtheit von mobilen Endgeräten wie Smartphones und Tablet-Computern rücken auch die Anwendungen für diese in den Fokus. Mobile Applikationen – kurz Apps – verleihen den Geräten Vielseitigkeit und erlauben, die Leistung der Hardware auszunutzen. Die Nachfrage nach Apps ist groß. Aus diesem Grund beschäftigen sich auch Unternehmen zunehmend mit der Entwicklung von Apps.

Problematisch ist die Fragmentierung des Marktes: mit Android, Blackberry, iOS, Symbian und Windows Phone gibt es fünf Plattformen, die signifikante Anteile haben [Gar12]. Aufgrund erheblicher Unterschiede etwa bezüglich Schnittstellen und Programmiersprachen muss für jede Plattform einzeln entwickelt werden, wenn Apps eine hohe Leistung bieten und eine native Benutzerführung aufweisen sollen. Gerade letztere ist für Unternehmen bedeutsam, wie wir durch die Arbeit mit unseren Praxispartner erfahren haben.

Modellgetriebene Softwareentwicklung (MDSD) [SV06] wird in Unternehmen mittler-

weile erfolgreich eingesetzt. Methoden und Werkzeuge werden durch Forschungstätigkeiten stetig verbessert. Die Grundidee ist dabei, ein fachliches Problem bzw. die dazu intendierte Lösung durch Software als Modell zu beschreiben. Aus diesem lässt sich Code generieren – bei geeigneter Anwendung auch für mehrere Plattformen. Wir haben daher mit MD^2 ein modellgetriebenes Cross-Plattform-Framework für die Implementierung mobiler Applikationen entwickelt. Apps werden in der eigens entwickelten domänenspezifischen Sprache MD^2-DSL spezifiziert und zu nativem Code transformiert.

Einen Überblick über den MD^2-Ansatz gibt bereits [HMK13]. Hier werden auch die verwendeten Generatoren genauer erläutert. Das vorliegende Paper fokussiert auf die zugehörige Domain-Specific Language (DSL) und stellt diese detailliert dar, während die Generatoren nur kurz angesprochen werden.

Dieser Beitrag ist wie folgt aufgebaut: Kapitel 2 gibt eine Einführung in domänenspezifische Sprachen. In Kapitel 3 beschreiben wir die Architektur unseres modellgetriebenen Ansatzes MD^2. Als Hauptteil des Beitrags stellt Kapitel 4 MD^2-DSL vor und erläutert, wie mit der Sprache Apps spezifiziert werden. Daran schließt sich in Kapitel 5 ein kurzer Einblick darein an, wie unser Ansatz aus den textuellen Modellen iOS- und Android-Apps generiert. Kapitel 6 diskutiert unseren Ansatz und insbesondere MD^2-DSL vor dem Hintergrund relevanter Literatur und vergleichbarer Ansätze im Umfeld von Programmiersprachen. Das Fazit in Kapitel 7 schließt den Beitrag ab.

2 Domänenspezifische Sprachen für die Anwendungsentwicklung

Domänenspezifische Sprachen (DSL) sind formale Sprachen, die für die Nutzung in einem bestimmten Problemkontext – der Domäne – entworfen wurden. In Anlehnung an [Fow10, Kap. 2.1] lassen sie sich grob wie folgt charakterisieren:

- DSLs werden von Menschen genutzt, um Computer für bestimmte Aufgaben zu instruieren. Viele DSLs können als Programmiersprachen mit einem sehr hohen Abstraktionsniveau angesehen werden.
- Jede DSL ist zugeschnitten auf eine bestimmte *Domäne*, die durch technische Vorgaben (z.B. Verwendung eines bestimmten Frameworks) und/oder inhaltliche Aspekte (z.B. E-Shop) bestimmt ist.
- Eine DSL ist typischerweise nicht Turing-mächtig.

DSLs lassen sich i.d.R. (auch von wenig technikaffinen Anwendern) leicht erlernen. Sie bieten eine kompakte Darstellung sowie eine hohe Ausdruckskraft in Bezug auf die Problemdomäne.

In unserem Fall liegt die Verwendung einer DSL nahe: es sollen so genannte *Business Apps* entwickelt werden, also mobile Applikationen, die einem geschäftlichen Zweck dienen. Insbesondere sind dabei datengetriebene Applikationen gemeint, die aus Formular-basierten Eingabe- und Ausgabemasken aufgebaut sind. Typischerweise implementieren

sie die *CRUD*-Funktionalität (Create, Read, Update, Delete) und greifen auf gerätespezifische Funktionen wie GPS zu. Diese Domäne ist einerseits im Umfang beschränkt, bildet aber andererseits fast alle für Unternehmen relevanten Szenarien ab.

DSLs kommen häufig bei modellgetriebener Softwareentwicklung zum Einsatz. Hierbei wird das zu erstellende Programm zunächst als textuelles oder grafisches Modell beschrieben. Anschließend folgt eine Reihe von Transformationsschritten, die schließlich zur Generierung von Quelltext führen. Je nach Ansatz und Vorgehensweise kann dieser Quelltext dann noch modifiziert werden; idealerweise stellt er aber bereits das Zielprogramm dar. DSLs bieten sich an, um kompakte aber gleichzeitig semantisch *reiche* Modelle zu erzeugen. Da mit dem Modell von der Zielplattform abstrahiert werden kann, bietet sich dieser Ansatz insbesondere für die Cross-Plattform-Entwicklung an.

Der Wunsch, Cross-Plattform-Entwicklung wesentlich zu vereinfachen, und die Eignung der Domäne Business Apps führten zu unserer Entscheidung, MD^2 als modellgetriebenes Framework mit der eigenen DSL MD^2-DSL zu implementieren. Dazu wurden zunächst mit Partnern in der Industrie typische Anforderungen an Business Apps herausgearbeitet. Anschließend folgte die Entwicklung eines Prototypen. Wir unterstützen zunächst die Code-Generierung für Tablets unter Android und iOS. Durch diese nicht-konzeptionelle Einschränkung kann ein Erfahrungsschatz aufgebaut werden, bevor wir die arbeitsintensive aber wenig Erkenntnisse liefernde Erweiterung für zusätzliche Plattformen angehen.

3 Konzepte und Architektur von MD^2

Im Folgenden wird kurz in die Grundkonzepte und Architektur von MD^2 eingeführt. Details finden sich in [HMK13]. Die Entwicklung von Apps mit MD^2 erfolgt in drei Schritten. Dabei erfordert nur der erste manuelle Arbeit durch den Entwickler. Dieser beschreibt als erstes die App als textuelles Modell. Im zweiten Schritt wird dieses Modell von einen Code-Generator verwendet, um plattformspezifischen Quelltext sowie zusätzliche Strukturelemente wie etwa XML-Konfigurationsdateien zu erzeugen. Es kommt pro unterstützter Plattform ein eigener Code-Generator zum Einsatz, da die Heterogenität der Plattformen einen plattformunabhängigen Zwischenschritt in der Transformation wenig sinnvoll macht: die zu erwartenden Synergien sind geringer als der Komplexitätsanstieg.

Als drittes muss der Quelltext zu einer App kompiliert werden. Dieser Schritt wird derzeit durch den Entwickler angestoßen, ist aber aufgrund der Nutzung der entsprechenden Entwicklungsumgebungen (Android Developer Tools bzw. Xcode für iOS) komplett automatisiert. Die erzeugten nativen Pakete können auf Endgeräte ausgebracht oder im Simulator debuggt werden. Da die dritte Phase ebenfalls komplett in den Arbeitsablauf integriert werden kann, ist die einzige herausfordernde (und zeitaufwendige) Aufgabe des Entwicklern das Schreiben von MD^2-DSL-Code. Während die ersten beiden Phasen durch MD^2 abgedeckt sind, erfolgt in der dritten Phase der Rückgriff auf die Werkzeuge der Plattformentwickler. Unser Framework stellt für alle drei Schritte Hilfsmittel zu Verfügung. Insbesondere die textuelle DSL MD^2-DSL ist ein wesentlicher Beitrag, da sie auf Business Apps zugeschnitten ist (siehe nächstes Kapitel). Im Rahmen von MD^2 wird auch eine

Entwicklungsumgebung bereitgestellt, die mit den üblichen Funktionen wie Syntaxhervorhebung, Inhaltsassistenz und Validatoren die Entwicklung maßgeblich vereinfacht.

Die Code-Generierung wird automatisch gestartet, sobald das Modell gespeichert wird. Dabei erzeugt der Parser zunächst die abstrakte Syntax des Modells und stellt diese für die weiteren Schritte der Generierung bereit. Ein Preprocessing bereitet das Modell vor. Danach wird das Modell durch die plattformspezifischen Generatoren traversiert, die sukzessive Quelltext erzeugen. Neben den Quelltexten – Java für Android und Objective-C für iOS – werden weitere Dateien generiert. Hierbei handelt es sich vor allem um XML-Dokumente, die z.B. Elemente der Benutzerschnittstelle spezifizieren. Hinzu kommen Konfigurationsdateien für die Entwicklungsumgebungen. Die generierten Apps werden mit statischen Bibliotheken gepackt, die häufig genutzt Funktionen bündeln.

Durch einen weiteren Code-Generator wird ein Server-Backend erzeugt. Es basiert auf dem Datenmodell der App und ist auf JavaEE-Applikationsservern lauffähig. Das Backend ist zwar in der bereitgestellten Form ausführbar, soll aber als Blaupause dienen, um die serverseitige Geschäftslogik zu implementieren bzw. unternehmensinterne Systeme anzubinden. Somit bleiben Apps schlank; bereits zur Verfügung stehende Dienste lassen sich nutzen. Dies entspricht auch dem Wunsch der meisten Unternehmen.

MD^2-DSL und auch die generierten Apps genügen dem Model-View-Controller (MVC) Entwurfsmuster. Folglich ist das textuelle Modell in *Model*, *View* und *Controller* aufgeteilt (vgl. Abb. 3, 5 und 7). In Apps wird der Controller durch ein Ereignis-System realisiert, das die Benutzerinteraktion abbildet sowie auf interne und externe Ereignisse reagiert.

4 MD^2-DSL

Dieses Kapitel stellt die domänenspezifische Sprache MD^2-DSL vor. Das erste Unterkapitel beschreibt, wie die Sprache auf Basis typischer Anforderungen mobiler Anwendungen entwickelt wurde. Es beschreibt ferner die Komponenten der Sprache und ihre Implementierung. Die folgenden Unterkapitel beschreiben die Elemente zur Datenmodellierung, zur Beschreibung der Benutzeroberfläche und zur Spezifikation der Kontrolllogik im Detail.

4.1 Entwurfsüberlegungen

Die Sprache MD^2-DSL wurde ausgehend von funktionalen Anforderungen aus der App-Entwicklung entworfen. Der Funktionsumfang der Sprache ergab sich demnach aus typischen Anforderungen an datengetriebene Business Apps. Dieses Prinzip steht im Gegensatz zu einem *Bottom-Up*-Vorgehen, bei dem ausgehend von den Funktionen, die mobile Plattformen typischerweise zur Verfügung stellen, die Sprachelemente festgelegt würden. Letzterer Gestaltungsansatz zielte auf eine Überdeckung der API mobiler Plattformen. Es bestünde die Gefahr, an den Anforderungen der Sprachanwender vorbei eine Sprache mit niedrigem Abstraktionsniveau zu entwerfen, die den kleinsten gemeinsamen Nenner mobiler Plattformen unterstützt. Unser *Top-Down*-Ansatz hat hingegen den Anspruch, dass

jedes Sprachelement eine spezifische Anforderung der letztendlichen Apps erfüllt. Entwickler sollen mit MD^2-DSL den Problemraum anstelle des Lösungsraums modellieren. Code-Generatoren sind dann für die Transformation in den Lösungsraum verantwortlich.

Diesem Ansatz folgend war unser Ausgangspunkt deshalb eine Erhebung typischer Anforderungen auf Basis von Fachkonzepten und Interviews mit Verantwortlichen. Ergebnis war die folgende Liste: die Sprache soll es App-Entwicklern ermöglichen,

1. Datentypen zu definieren,
2. sowie lokal und serverseitig Datensätze dieser Typen anzulegen, zu lesen, zu aktualisieren und zu löschen, d.h. die typischen CRUD-Operationen auf Instanzen der Typen auszuführen;
3. die Benutzeroberfläche mit verschiedenen Layouts und typischen Steuerelementen zu beschreiben, besonders wichtig sind dabei Registerkarten (Tabs);
4. die Benutzernavigation zwischen den Ansichten zu steuern;
5. Datenbindung und Eingabevalidation zu definieren;
6. auf Benutzerereignisse und Statusänderungen zu reagieren; und
7. gerätespezifische Funktionen wie GPS oder Kamera zu nutzen.

Die Sprachelemente, welche die funktionalen Anforderungen implementieren, und deren Syntaxen werden in den folgenden Unterkapiteln beschrieben. Daneben prägten einige nicht-funktionale Anforderungen die Struktur und Syntax der Sprache. Gefordert waren eine klare Trennung unterschiedlicher Aspekte, Modularisierung, angemessenes Abstraktionsniveau und Verständlichkeit. Trennung und Modularisierung wurden erreicht, indem die Sprache einer MVC-Architektur folgt. Datenmodell (Model), Benutzerschnittstelle (View) und Kontrolllogik (Controller) sind in separaten Dateien zu formulieren, die zusammen das vollständige Modell ergeben. Beziehungen zwischen diesen Teilmodellen folgen dabei dem MVC-Entwurfsmuster. Die Struktur der Sprache ist so flexibel, dass auch diese Teilmodelle modular aufgebaut sind und über mehrere Dateien verteilt werden können. Zum Beispiel könnte jede Registerkarte einer App in einer eigenen Datei definiert werden. Ein modularer Aufbau der App-Beschreibung wird ferner durch Möglichkeiten zur Wiederverwendung, z.B. von Teilen der Benutzeroberfläche, unterstützt.

MD^2-DSL ist vorwiegend deklarativ. Entwickler beschreiben, was die App erreichen soll, aber nicht (mit Ausnahme einiger Aktionen) wie dies algorithmisch ablaufen soll. Deutlich erkennbar ist die deklarative Natur z.B. bei den Sprachelementen für die Benutzeroberfläche. Der Entwickler beschreibt Aussehen und Zusammensetzung der Oberfläche, aber keine Abfolge imperativer Anweisungen zum schrittweisen Aufbau derselben. Der deklarative Sprachstil sorgt in Verbindung mit dem Konvention-über-Konfiguration-Prinzip für eine verständliche und prägnante Sprache mit einem gehobenen Abstraktionsniveau.

MD^2-DSL ist in Xtext [Xte12b] implementiert, einem Framework zur Erstellung textueller Sprachen. Aus einer Sprachbeschreibung in einer attributierten Grammatik mit EBNF-ähnlicher Syntax generiert Xtext einen Parser, die abstrakte Syntax als Klassenmodell und einen Editor für die Eclipse-Umgebung.

```
1 MD2Model ::= PackageDefinition MD2ModelLayer?
2 PackageDefinition ::= 'package' QualifiedName
3 MD2ModelLayer ::= Model | View | Controller
4 QualifiedName ::= ID ('.' ID)*
```

Abbildung 1: Auszug aus der EBNF-Darstellung von MD^2-DSL: Auflösung des Startsymbols

```
1 MD2Model:
2   package=PackageDefinition
3   modelLayer=MD2ModelLayer? ;
4 MD2ModelLayer: Model | View | Controller ;
5 PackageDefinition:
6   'package' pkgName=QualifiedName ;
7 QualifiedName: ID ('.' ID)* ;
```

Abbildung 2: Pendant zu Abbildung 1 in Xtext

Ausschnitte aus der Syntax von MD^2-DSL sind im Folgenden im W3C-Stil der Erweiterten Backus-Naur-Form [W3C04] dargestellt. Alle Nichtterminal-Symbole beginnen mit einem Großbuchstaben. Als vordefinierte Symbole werden `ID`, `INT` und `STRING` verwendet. Diese symbolisieren die Menge aller gültigen Bezeichner, nicht-negativen Zahlen beziehungsweise Zeichenketten.

Abbildung 1 enthält die Regeln zur Auflösung des Startsymbols `MD2Model`, die somit die grundlegende Struktur von MD^2-DSL-Modellen beschreiben. Deutlich wird die generelle Aufteilung in Model, View und Controller (Zeile 3). Jedes Teilmodell beschreibt einen dieser Bereiche. Die entsprechenden Nichtterminal-Symbole werden in den folgenden Unterkapiteln weiter aufgelöst. Das Nichtterminal-Symbol **QualifiedName** wird an den Stellen der Grammatik verwendet, an denen Referenzen zwischen Elementen spezifiziert werden. Xtext ermöglicht für solche Querverweise zusätzlich, auf Ebene der abstrakten Syntax zu beschränken, welcher Typ von Elementen referenziert werden kann. Des Weiteren kann der Gültigkeitsbereich von Elementen feingranular festgelegt werden. Zur Auflösung der Querverweise sieht Xtext nach dem Parsen eine Phase des Linkings vor.

Abbildung 2 enthält die zu Abbildung 1 korrespondierende Implementierung in Xtext, die zusätzlich zur Definition der konkreten Syntax über Zuweisungen (z.B. in Zeile 2) die abstrakte Syntax festlegt. Die vollständige Xtext-Grammatik von MD^2-DSL ist aus Platzgründen unter http://www.wi.uni-muenster.de/pi/forschung/md2/MD2.xtext verfügbar.

Die folgende Darstellung von MD^2-DSL erläutert die Sprache in Ergänzung zu ihrer Grammatik am Beispiel einer vereinfachten Bestell-App, die mit MD^2 implementiert wurde. In ihrer ersten Ansicht kann der Benutzer den Namen eines Produkts eingeben und nach diesem suchen. Das von einem Server übermittelte (zu Illustrationszwecken eindeutige) Ergebnis stellt eine zweite Ansicht dar, in der der Benutzer eine Bestellung aufgeben kann. Zur Vereinfachung genügt dazu die Angabe seiner E-Mail-Adresse. Die Abbildungen 3, 5 und 7 zeigen den nahezu vollständigen MD^2-DSL-Quelltext der App, aufgeteilt nach den Sprachkomponenten Model, View und Controller. Abbildung 9 stellt Screenshots der iOS- und Android-Apps gegenüber, die von MD^2 aus dem Quelltext generiert worden sind.

```
1  package de.md2.bestellung.models
2  entity PRODUKT {
3    name : string
4    preis : integer
5    beschreibung : string(optional)
6  }
7  entity BESTELLUNG {
8    produkt : PRODUKT
9    email : string
10 }
```

Abbildung 3: Datenmodell einer Beispiel-App in MD^2-DSL

```
1 Model ::= ModelElement*
2 ModelElement ::= Entity | Enum
3 Entity ::= 'entity' ID '{' Property* '}'
4 Property ::= ID ':' PropertyType ('(' TypeParam (',' TypeParam)* ')')?
5 PropertyType ::= QualifiedName | 'integer' | 'string' | 'date' | ...
```

Abbildung 4: EBNF-Darstellung der Grammatik von MD^2-DSL: Auszug aus dem Model-Teil

4.2 Datenmodellierung in MD^2-DSL

MD^2-DSLs Komponente zur Datenmodellierung stellt die üblichen Elemente zur Beschreibung von Typen bereit. Sogenannte Entities werden mit dem entsprechenden Schlüsselwort und einem Bezeichner eingeleitet (Abb. 4, Zeile 3). Eingeschlossen in geschweifte Klammern folgt eine Liste von Eigenschaften, bestehend aus Bezeichner und Typangabe. Neben vordefinierten Standardtypen, u. A. für Zeichenketten, ganze Zahlen und Datumsangaben, können Eigenschaften auch auf andere Entities über deren qualifizierten Bezeichner verweisen (Zeile 5). An dieser Stelle sind alle im selben Paket spezifizierten Entities allein über ihren Bezeichner referenzierbar. Die Xtext-Implementierung definiert, dass hier tatsächlich nur Bezeichner von Entities gültig sind. Falls nötig, können Typparameter eine Eigenschaft näher spezifizieren, z.B. als optional oder mehrwertig, aber auch durch komplexere Einschränkungen des zulässigen Wertebereichs.

Zusätzlich zu Entities unterstützt MD^2-DSL Enumerationen. Komplexere Typbeziehungen wie Vererbung werden absichtlich nicht unterstützt, um die Komplexität der Sprache zu reduzieren. Da sich dieser Teil der Sprache vorwiegend bekannter Konzepte bedient, ermöglicht er App-Entwicklern einen einfachen Einstieg.

4.3 Beschreibung der Benutzerschnittstelle mit MD^2-DSL

Die Sprachkomponente für die Benutzerschnittstelle stellt zwei Arten von Anzeigeelementen zur Verfügung: Inhaltselemente und Container (Abb. 6, Zeile 3). Inhaltselemente wie Label, Buttons oder Texteingabefelder werden in Containern gruppiert. Verschiedene Containerarten (*Panes*) repräsentieren unterschiedliche Layouts. Sie können wiederum in anderen Containern geschachtelt werden. Eine besondere Art von Container sind *TabbedPanes* für Benutzeroberflächen mit App-typischer Registernavigation. Die direkten Kind-Container eines TabbedPane bilden dabei die Registerkarten.

```
1  package de.md2.bestellung.views
2  TabbedPane AppFenster {
3    SuchenTab -> Suchen
4    BestellenTab -> Bestellen(tabTitle "Bestellung")
5    InfoTab(tabTitle "Info")
6  }
7  FlowLayoutPane SuchenTab(vertical) {
8    Label sucheLbl { text "Suche nach Produkt" style Gross }
9    TextInput suchFeld { label "Produktname"
10     tooltip "Geben Sie den Namen des Produkts ein, ..."
11   }
12   Button sucheBtn ("Suchen")
13 }
14 FlowLayoutPane BestellenTab(vertical) {
15   Label bestellenLbl { text "Bestellung aufgeben" style Gross }
16   Label info("Bitte geben Sie Ihre E-Mail-Adresse an, ...")
17   AutoGenerator bestellung { contentProvider bestellungProvider }
18   Button bestellenBtn ("Bestellen")
19 }
20 FlowLayoutPane InfoTab { ... }
21 style Gross { fontSize 20 textStyle bold }
```

Abbildung 5: Modell der Benutzerschnittstelle einer Beispiel-App in MD^2-DSL

```
1  View ::= ViewElement*
2  ViewElement ::= ViewGUIElement | Style
3  ViewGUIElement ::= ContentElement | ContainerElement
4  ContentElement ::= Label | TextInput | Button | ...
5  Label ::= 'Label' ID ( '(' STRING ')' | '{' 'text' STRING ('style' ID )? '}' )
6  ContainerElement ::= FlowLayoutPane | GridLayoutPane | TabbedPane | ...
7  FlowLayoutPane ::= 'FlowLayoutPane' ID ('(' FlowParam (',' FlowParam )* ')')?
8                     '{' PaneContent* '}'
9  PaneContent ::= ViewGUIElement | (QualifiedName ('->' ID)?)
```

Abbildung 6: EBNF-Darstellung der Grammatik von MD^2-DSL: Auszug aus dem View-Teil

Um eine modulare Beschreibung der Oberfläche zu ermöglichen, bietet MD^2-DSL die Option, Anzeigeelemente nicht nur direkt zu schachteln, sondern auch anderweitig definierte Elemente zu referenzieren (Zeile 9). Dieses Feature kann wie im Beispiel (Abb. 5) genutzt werden, um einzelne Registerkarten oder Teile der Oberfläche separat zu definieren und die Übersichtlichkeit zu erhöhen. Es ermöglicht ebenso Wiederverwendung. Über ihren qualifizierten Namen referenzierten Anzeigeelementen kann ein neuer Bezeichner zugewiesen werden, mit dem die spezifische Instanz im Controller referenziert werden kann.

Auch die UI-Komponente von MD^2-DSL nutzt allgemein bekannte Konzepte und setzt damit Anforderung 3 auf einfach erlernbare, aber mächtige Weise um. Ein besonderes Element sind sogenannte AutoGenerator-Elemente (Abb. 5, Zeile 17). Diese stehen für eine Standarddarstellung eines Entity-Typs (im Beispiel: BESTELLUNG) mit entsprechenden Labeln und Eingabefeldern. Sie implizieren zudem Datenbindungen zwischen den Inhaltselementen und Objekten eines Datenlieferanten (*ContentProvider*, s.u.) und entledigen den Entwickler, falls gewünscht, manuellen Aufwands. Auch besondere und individuelle Anforderungen an das Design von Apps unterstützt MD^2-DSL, z.B. indem der Stil von Inhaltselementen spezifiziert oder Grafiken eingebunden werden können. Die aus dem GUI-Modell generierte Oberfläche der Beispielapp stellt Abbildung 9 dar.

```
1  package de.md2.bestellung.controllers
2  main {
3    appName "Bestellapp" appVersion "ATPS 2013" modelVersion "1.0"
4    startView AppFenster.Suchen
5    onInitialized init
6  }
7  action CombinedAction init {
8    actions ereignisseRegistrieren validatorenBinden
9  }
10 action CustomAction ereignisseRegistrieren {
11   bind action produktLaden on SuchenTab.sucheBtn.onTouch
12   bind action bestellungAufgeben on BestellenTab.bestellenBtn.onTouch
13 }
14 action CustomAction validatorenBinden {
15   bind validator RegExValidator(regEx "...")
16     on BestellenTab.bestellung[Bestellung.email]
17 }
18 action CustomAction produktLaden {
19   call DataAction(load suchergebnis)
20   call NewObjectAction(bestellungProvider)
21   call AssignObjectAction(use suchergebnis for bestellungProvider.produkt)
22   call GotoViewAction(AppFenster.Bestellen)
23 }
24 action CustomAction bestellungAufgeben {
25   call DataAction(save bestellungProvider)
26 }
27 contentProvider Produkt suchergebnis { providerType backend
28   filter first where Produkt.name equals AppFenster.Suchen->SuchenTab.suchFeld
29 }
30 contentProvider Bestellung bestellungProvider {providerType backend }
31 remoteConnection backend { uri "http://.../de.md2.bestellung.backend/service" }
```

Abbildung 7: Modell der Kontrolllogik einer Beispiel-App in MD^2-DSL

4.4 Beschreibung der Kontrolllogik und des Verhaltens mit MD^2-DSL

Während Model und View die statischen Teile einer App umfassen, steuert der Controller einer MD^2-App deren Verhalten. Die entsprechende MD^2-DSL-Komponente bringt dazu Model und View zusammen. Ein `Main`-Block definiert übergreifende Informationen und spezifiziert den Anfangszustand der GUI sowie beim Start auszuführende Aktionen.

Aktionen sind das zentrale Element in MD^2-DSL zur Definition dynamischer Aspekte. Die mächtigen individuellen Aktionen (*CustomAction*) erlauben es, eine sequentiell auszuführende Liste von Aktionsfragmenten zu definieren. Eine wichtige Art derartiger Fragmente registriert wiederum Aktionen als Behandler für bestimmte Ereignisse (Abb. 8, Zeile 6), wodurch interaktive Benutzerschnittstellen realisiert werden können (Anforderung 6). Neben Interaktionen des Benutzers mit der GUI kann auch auf Veränderungen des globalen Zustandsraums reagiert werden, z.B. auf einen Abbruch der Netzwerkverbindung. Das Sprachkonzept der Bedingungsereignisse erlaubt die Spezifikation komplexer Bedingungen unter Rückgriff auf den Zustand von Model und View, bei deren Erfüllung die für das Ereignis registrierten Aktionen ausgeführt werden. Die hierfür notwendige Sprache für boolesche Ausdrücke wird auch an anderen Stellen in MD^2-DSL verwendet, z.B. für Selektionsausdrücke in Datenabfragen. MD^{2}s dynamische Komponente ist vorwiegend ereignisbasiert, da Aktionen – mit Ausnahme der Startaktion – nur als Reaktion auf Ereignisse ausgeführt werden, an die sie zuvor, zumeist in der Startaktion, gebunden wurden.

Die Aktionen selbst werden als Sequenz definiert und haben einen imperativen Charakter, sind aber auf einem hohen Abstraktionsniveau in das deklarative Umfeld eingebettet. Die Sprache unterstützt keine anderen Kontrollstrukturen.

Die Angabe des Anzeigeelements, an das ein Ereignis gebunden werden soll, erfolgt wie alle Verweise auf ein Anzeigeelement, das im View-Modell definiert wurde, in MD^2-DSL über das Konzept der Anzeigeelement-Referenz (*ViewGUIElementRef*, Abb. 8, Zeile 8 f.). Im einfachsten Fall handelt es sich um den qualifizierten Bezeichner eines Anzeigeelements. Bei wiederverwendeten, referenzierten Anzeigeelementen besteht die Referenz aus zwei Teilen: dem qualifizierten Bezeichner zu der Stelle, an der die Einbindung definiert ist, und, abgetrennt durch einen Pfeil, dem Bezeichner innerhalb des eingebundenen Elements. Auf diese Weise kann ein spezifisches Vorkommen referenziert werden (Abb. 7, Zeile 28). Es ist aber genauso möglich, alle Vorkommen eines Anzeigeelements zu referenzieren, indem nach der ersten Methode der qualifizierte Bezeichner der allgemeinen Definition genutzt wird. Wenn die auf eine dieser Arten definierte Referenz auf einen AutoGenerator verweist, kann zusätzlich in eckigen Klammern eine Eigenschaft des Entity-Typs angegeben werden, für den der Generator definiert wurde (Zeile 16). Die Anzeigeelement-Referenz bezieht sich dann auf das dieser Eigenschaft entsprechende implizit generierte Feld.

Neben dem erwähnten Fragment zur Ereignisbindung stellt MD^2-DSL eine Vielzahl weiterer Aktionsfragmente bereit. Sie ermöglichen beispielsweise, Inhaltselemente mit Validatoren und Datenbindungen zu versehen (siehe unten), oder gerätespezifische Aktionen (GPS-Lokation, Anforderung 7) auszuführen. Andere Fragmente steuern die Navigation innerhalb der grafischen Oberfläche der App und können zwischen Ansichten wechseln. Die CRUD-Operationen werden ebenfalls durch mehrere Aktionsfragmente unterstützt.

Als Sprache für die Beschreibung datenzentrierter Business Apps bietet MD^2-DSL Elemente zur Verknüpfung der Apps mit lokalen und entfernten Datenquellen. Beide werden über Datenlieferanten (*ContentProvider*) angebunden. Ein Lieferant verweist auf einen Datenspeicher, bei dem es sich entweder um eine Datenbank auf dem Gerät oder um einen Server handelt, der über eine vorgegebene API Zugriff bietet. Jeder Datenlieferant verweist auf einen spezifischen Ausschnitt der Datenquelle, der über den Typ der Daten und einen Filter definiert wird. Im Beispiel verweist der Lieferant `suchergebnis` (Zeile 27 ff.) auf das Produkt, dessen Namen der Eingabe im Suchfeld entspricht. Der Typ des Datenlieferanten wird so angegeben, wie er auch für Eigenschaften in der Datenkomponente von MD^2-DSL definiert ist, d.h. er verweist entweder auf einen vordefinierten Datentyp oder einen im Datenmodell deklarierten Entitytyp; zudem kann er mehrwertig sein. Ein Filter entspricht einer Abfrage, die Objekte selektiert, die der angegebenen Bedingung entsprechend. Im einfachsten Fall entspricht die Bedingung einem Vergleich einer Eigenschaft des Entitytyps, angegeben als qualifizierter Bezeichner, mit einem Anzeigeelement, auf das über die beschriebene Anzeigeelement-Referenz verwiesen wird (Abb. 8, Zeile 13).

Verschiedene Aktionsfragmente von MD^2-DSL operieren auf Datenlieferanten und setzen zusammen mit diesen Anforderung 2 um. Eine *DataAction* spezifiziert, dass auf dem durch den Datenlieferanten bezeichneten Ausschnitt eine der CRUD-Operationen Speichern, Laden oder Löschen ausgeführt wird. Eine neue Initialisierung ist mit einer *NewObjectAction* möglich, während Assoziationen zwischen den Entities zweier nicht-mehrwertiger

```
1  Controller ::= ControllerElement*
2  ControllerElement ::= Main | Action | ContentProvider | Workflow | ...
3  Action ::= 'action' (CustomAction | CombinedAction)
4  CustomAction ::= 'CustomAction' ID '{' CodeFragment* '}'
5  CodeFragment ::= EventBinding | ValidatorBinding | ActionCall | Mapping | ...
6  EventBinding ::= 'bind action' QualifiedName* 'on' (GUIEvent | GlobalEvent)
7  GUIEvent ::= ViewGUIElementRef '.' EventType
8  ViewGUIElementRef ::= QualifiedName ('->' QualifiedName)*
9                        ('[' QualifiedName ']')?
10 ContentProvider ::= 'contentProvider' PropertyType '[]'? ID
11                     '{' ProviderType Filter? '}'
12 Filter ::= 'filter' FilterType ('where' WhereClause)?
13 WhereClause /*stark verkürzt*/ ::= QualifiedName 'equals' ViewGUIElementRef
```

Abbildung 8: EBNF-Darstellung der Grammatik von MD^2-DSL: Auszug aus dem Controller-Teil

Datenlieferanten über *AssignObjectAction* gepflegt werden. Dabei wird die aktuelle Entity eines Lieferanten einer Eigenschaft der Entity eines zweiten Lieferanten zugewiesen. Die Bestell-App nutzt das geladene Suchergebnis als Produkt für eine neu initialisierte Bestellung (Abb. 7, Zeilen 19–21).

Datenbindungen zwischen Datenlieferanten und Inhaltselementen sind als Aktionsfragmente realisiert und können als solche dynamisch zugewiesen werden. Eine Datenbindung verknüpft ein Anzeigeelement mit einer Eigenschaft eines Datenobjekts, das von einem Datenlieferanten verwaltet wird. Sie stellt dauerhaft die Konsistenz zwischen Daten und GUI in beide Richtungen sicher (Anforderung 5). Im Fall eines Eingabefelds impliziert eine Datenbindung zudem einen Validator, der vom Datentyp und zusätzlich spezifizierten Einschränkungen abgeleitet wird. Das Überschreiben und Ergänzen impliziter Validatoren ist über entsprechende Aktionsfragmente möglich. Eingaben können z.B. darauf überprüft werden, ob sie vorhanden sind, ob sie konform zu einem Datentyp sind oder ob sie einem regulären Ausdruck entsprechen. Bei AutoGeneratoren werden sowohl Datenbindungen als auch Validatoren zunächst automatisch abgeleitet, was in der Beispielapp genügt.

Ein fortgeschrittenes Sprachelement von MD^2-DSL ist die Möglichkeit, Navigationspfade durch die App im Sinne von *Workflows* zu spezifizieren. Sie sind insbesondere in Registerkarten-basierten Benutzeroberflächen wichtig, um den Nutzer einer App angesichts vergleichsweise vielfältiger Optionen bei der Navigation zu unterstützen. Aus Platzgründen kann das Konzept hier nur angerissen werden und wird nicht im Beispiel genutzt. Ein Workflow besteht aus mehreren Schritten. Jedem Workflow-Schritt ist ein Container der GUI zugewiesen, der angezeigt wird, falls der Schritt aktiv ist. Für einen Schritt definierte Bedingungen – z.B. vollständig ausgefüllte Eingabefelder – beeinflussen, wann ein anderer Schritt aktiv werden kann. Verschiedene Aktionsfragmente ermöglichen es in Verbindung mit Workflows, die Navigationspfade einer App zu weiter spezifizieren.

5 Generierung von iOS- und Android-Apps

Die Spezifikation von MD^2-DSL, der DSL von MD^2, entspricht auch dem Funktionsumfang des Frameworks insgesamt, da die Menge gültiger Modelle in MD^2-DSL die Menge möglicher Eingaben des Code-Generators darstellt. Das vorstehende Kapitel beschreibt

neben den zur Verfügung stehenden Sprachelementen auch deren Semantik, soweit es der beschränkte Platz erlaubt. Dieser Abschnitt geht kurz auf die Implementierung der drei Code-Generatoren ein (Details in [HMK13]). Wie eingangs beschrieben, werden vollständige und lauffähige Android- und iOS-Apps sowie eine JavaEE-Anwendung generiert, welche die Serverschnittstelle von Datenlieferanten implementiert.

Jeder Code-Generator ist in Xtend [Xte12a] implementiert, einer Java-ähnlichen Programmiersprache mit für Codegeneratoren hilfreicher Zusatzfunktionalität wie Template-Ausdrücken. Wie Abbildungen 10 und 11 am Beispiel der Generierung von zu Entities korrespondierendem Quelltext demonstrieren, sind die Generatoren hinreichend unterschiedlich, um voneinander unabhängig implementiert zu sein. Abfragen auf der abstrakten Syntax des Eingabemodells werden aber von allen Generatoren genutzt. Zudem reichert ein vorverarbeitender Schritt das Modell mit zusätzlichen Informationen an und modifiziert es, um die Codegenerierung zu vereinfachen. So werden beispielsweise Auto-Generatoren durch explizite Anzeigeelemente, Datenbindungen und Validatoren ersetzt.

Hauptaufgabe der Generatoren ist es, die deklarativen Sprachkonzepte auf den jeweiligen Zielplattformen zu explizieren. Eine Zielplattform ist nicht nur durch die Programmiersprache bestimmt, in der generierter Code verfasst ist, sondern auch durch die zur Verfügung stehenden Bibliotheken und statischen, von MD^2 mitgelieferten Inhalt. Die Generatoren folgen den Richtlinien der jeweiligen mobilen Plattform und erzeugen z.B. XML-Dokumente für das Layout von Android-Apps. Die Generatoren nutzen jeweils plattformspezifische Konzepte, um die abstrakten Sprachkonzepte umzusetzen. Wo möglich, greifen sie auf typische Elemente der Plattform zurück, um ein natives Aussehen zu erreichen. Fehlt auf einer Plattform ein Pendant eines bestimmten Konzepts, setzt der Generator dieses aus bestehenden Elementen zusammen, z.B. wird das Grid-basierte Layout auf Android für den App-Nutzer transparent durch ein TableLayout emuliert.

6 Verwandte Arbeiten und Diskussion

MD^2 lässt sich von anderen Ansätzen zur plattformübergreifenden App-Entwicklung abgrenzen. Hierzu gehören: mobile Web-Apps, hybride Apps, Laufzeitumgebungen und generative Ansätze. *Webapps* verwenden HTML, CSS und JavaScript und lassen sich in einem Browser aufrufen. Sie können auf gerätespezifische Features nicht bzw. nur eingeschränkt im Rahmen von HTML 5 zugreifen. Weiterhin vermitteln sie das Bediengefühl einer Webseite und erreichen kein plattformspezifisches Look & Feel. Letzteres ist aber sehr wichtig, wie wir von unseren Praxispartnern lernen konnten. *Hybride Apps* wie Apache Cordova [Apa12] (ehemals PhoneGap) verpacken eine native Komponente in eine Webseite und können so auch gerätespezifische Features verfügbar machen. Aber auch diese erreichen kein plattformspezifisches Look & Feel. MD^2 erreicht dieses plattformspezifische Look & Feel trivialerweise, da jeweils plattformspezifischer Code generiert wird.

Ansätze wie Appcelerator Titanium [App12a], die eine separate Laufzeitumgebung verwenden und hiermit plattformübergreifenden Skript-Code interpretieren, können im Prinzip ein plattformspezifisches Look & Feel erreichen. Allerdings führt die zusätzliche In-

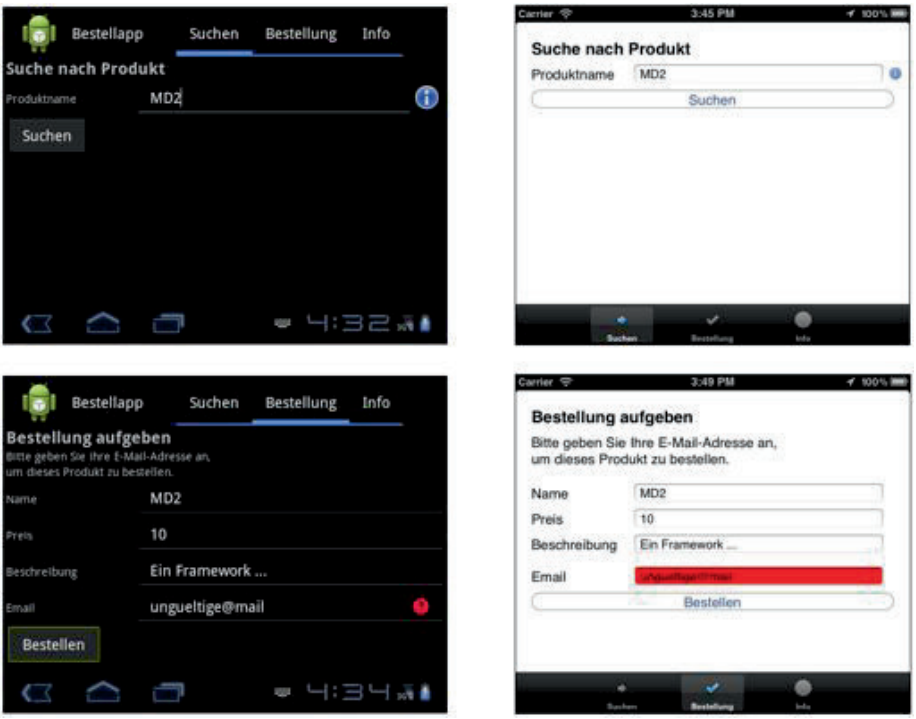

Abbildung 9: Screenshots der Bestell-App für iOS und Android (zu Illustrationszwecken modifiziert)

```
1 public class «entity.name» implements Entity {
2   private int __internalId;
3   «FOR attribute : entity.attributes»
4     private «attributeTypeName(entity, attribute)» «attribute.name»;
5   «ENDFOR»
6 }
```

Abbildung 10: Template zur Generierung von Java-Quelltext für Entities in Android (Auszug)

```
1 @interface «entity.name.toFirstUpper»Entity : DataTransferObject
2
3 «FOR attribute : entity.attributes»
4   @property (retain) «attributeTypeName(entity, attribute)»* «attribute.name»;
5 «ENDFOR»
```

Abbildung 11: Template zur Generierung von Objective-C-Quelltext für Entities in iOS (Auszug)

terpretation zu erheblichen und manchmal prohibitiven Laufzeiteinbußen.

Neben modellgetriebenen Ansätzen gibt es auch weitere generative Ansätze zur App-Entwicklung. Die Cross-Compiler XMLVM [XML12] und J2ObjC [J2O12] beispielsweise transformieren Java- in Objective-C-Code (für iOS), berücksichtigen aber keine gerätespezifische Funktionalität und sind auf iOS begrenzt. Alternative modellgetriebe-

ne Ansätze wie applause [app12b] und AXIOM [JJ12] generieren ebenso wie MD^2 Apps aus einer DSL. Applause ist auf Informationsdarstellung beschränkt. Im Fall von AXIOM wurden die Sprachelemente bottom-up aus den Geräte-Features abgeleitet. Die betreffende DSL arbeitet daher auf einem niedrigen Abstraktionsniveau; zudem ist die Transformation nicht vollständig automatisiert. Wie alle zuvor erwähnten Ansätze profitieren sie daher im Gegensatz zu MD^2 nicht von Besonderheiten der Anwendungsdomäne Business Apps.

MD^2 erzeugt dagegen aus sehr kompakten DSL-Modellen effizienten, nativen Code für jede Zielplattform. Die DSL-Features wurden nicht aus den Gerätemerkmalen sondern aus den Anforderungen der Anwendungsdomäne abgeleitet. Ein Tarifrechner, der in Zusammenarbeit mit einem Praxispartner aus der Versicherungsbranche entwickelt wurde, konnte beispielsweise in 709 DSL-Zeilen formuliert werden, die in 10110 Zeilen Java und 2263 Zeilen XML für Android bzw. 3270 Zeilen Objective-C und 64 Zeilen XML für iOS transformiert wurden. Dies verdeutlicht den Produktivitätsgewinn durch die deutlich höhere Abstraktion. Weiterhin mussten die 709 Zeilen natürlich nur einmal erstellt werden und konnten dann automatisch in Code für jede Plattform transformiert werden.

Unser Ansatz hat weiterhin eine gewisse Ähnlichkeit zu dem Task-orientierten Toolkit iTasks [MPA11], mit dem sich Workflows mit GUI-Elementen erstellen lassen. iTasks verwendet keinen Modell-getriebenen Ansatz sondern monadische Kombinatoren höherer Ordnung ist damit wesentlich komplexer als MD^2-DSL. Außerdem ist es nicht auf mobile Endgeräte zugeschnitten und bietet insofern auch keine Unterstützung gerätespezifischer Funktionen auf verschiedenen Plattformen.

MD^2 ist nicht für jede App geeignet, sondern gezielt auf datengetriebene Business Apps zugeschnitten, bei denen die Benutzerschnittstelle im Wesentlichen aus Textfeldern, Labels, Buttons, Menüs und Listen besteht. MD^2 ist insbesondere ungeeignet für Apps, bei denen im Gerät komplexe Algorithmen ausgeführt werden müssen, wie dies zum Beispiel bei Spielen oder Multimedia-Anwendungen der Fall ist. MD^2 unterstützt nur die gerätespezifischen Features (wie z.B. GPS), die für Business Apps benötigt werden. Hinsichtlich der zur Verfügung gestellten Kontrollstrukturen ist MD^2 bewusst limitiert. Verzweigungen sind nur auf Grundlage der festgestellten Ereignisse möglich. Schleifen lassen sich in eingeschränkter Form durch Rückkehr zu einem zuvor besuchten Bildschirm realisieren. Komplexere Algorithmen werden bei Business Apps typischerweise auf der Server-Seite ausgeführt. Bei allen bisher betrachteten Business Apps (neben einem Tarifrechner für ein Versicherung u.a. eine Bestell-App für einen E-Shop und eine Bibliotheks-App) reichten die verfügbaren Kontrollstrukturen aus. Um die DSL kompakt zu halten, haben wir daher auf Weitere verzichtet. Sollte sich zukünftig ein Bedarf an weiteren Kontrollstrukturen abzeichnen, so können diese gegebenenfalls später ergänzt werden. Die Einbeziehung eines Servers ist typisch, aber nicht zwingend erforderlich. Auch Apps, bei denen Daten lokal auf dem Gerät gespeichert und durch simple Workflows manipuliert werden, sind realisierbar.

7 Zusammenfassung und Ausblick

Wir haben die textuelle domänenspezifische Sprache MD^2-DSL vorgestellt, die sich zur plattformübergreifenden Entwicklung datengetriebener Business Apps für mobile Endgeräte eignet. MD^2-DSL-Dokumente werden unter Verwendung von Xtext und Xtend automatisch in Code für die jeweilige Zielplattform übersetzt. Zur Zeit werden die Plattformen iOS/Objective-C und Android/Java unterstützt. Weitere Plattformen werden später ergänzt. Im Gegensatz zu anderen DSLs zur App-Entwicklung wurde MD^2-DSL nicht bottom-up aus den Features mobiler Endgeräte sondern top-down aus den Anforderungen an Business Apps konzipiert und erreicht damit ein gegenüber anderen Ansätzen deutlich höheres Abstraktionsniveau. Insbesondere unterstützt MD^2-DSL die bekannte Model-View-Controller-Architektur. Die View-Komponente wird deklarativ durch Schachtelung von Containern und Platzierung von Bedienelementen wie Texteingabefeldern und Knöpfen beschrieben. Das Model besteht im Wesentlichen aus einer ebenfalls deklarativen Beschreibung der benötigten Entitäten. Die Controller-Komponente legt schließlich fest, wie die relevanten Ereignisse behandelt werden sollen. Weiterhin können Validatoren zur Überprüfung von Eingaben angegeben werden. Einen erheblichen Teil einer typischen Business App machen simple CRUD-Operationen auf den Entitäten aus. Eine zugehörige Benutzerschnittstelle hierfür kann automatisch auf Basis von MD^2-DSLs Datenlieferanten und Autogeneratoren generiert werden.

Manche anderen Ansätze zur plattformübergreifenden App-Entwicklung verlieren Effizienz durch das Einschieben einer zusätzlichen Skript-Code-Interpretationsschicht. Bei unserem Ansatz ist dies nicht der Fall. Es wird direkt plattformspezifischer Code erzeugt, wie man ihn in etwa auch von Hand schreiben würde. Dieser plattformspezifische Code garantiert insbesondere auch das Look & Feel der jeweiligen Zielplattform. Diese sehr wichtige Eigenschaft wird von den meisten anderen Ansätzen zur plattformübergreifenden App-Entwicklung nicht erreicht. Wie sich an einigen Beispielanwendungen mit einem Praxispartner aus der Versicherungsbranche zeigte, erlaubt MD^2-DSL eine sehr kompakte Formulierung von Business Apps und damit eine hohe Produktivität. Mehraufwand für die Übertragung einer App auf andere Plattformen entfällt.

Literatur

[Apa12] Apache Cordova, 2012. http://incubator.apache.org/cordova/.

[App12a] Appcelerator, 2012. http://www.appcelerator.com/.

[app12b] applause, 2012. https://github.com/applause/.

[Fow10] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley, 2010.

[Gar12] Gartner Press Release, 2012. http://www.gartner.com/it/page.jsp?id=1622614.

[HMK13] Henning Heitkötter, Tim A. Majchrzak und Herbert Kuchen. Cross-Platform Model-Driven Development of Mobile Applications with MD2. In *Proc. of the 2013 ACM Symp. on Applied Computing (SAC)*. ACM, 2013.

[J2O12] J2ObjC, 2012. https://code.google.com/p/j2objc/.

[JJ12] Xiaoping Jia und Christopher Jones. AXIOM: A Model-driven Approach to Cross-platform Application Development. In *Proc. 7th ICSOFT*, 2012.

[MPA11] Steffen Michels, Rinus Plasmeijer und Peter Achten. iTask as a New Paradigm for Building GUI Applications. In *Proceedings of 22nd IFL, LNCS 6647*, Seiten 153–168, 2011.

[SV06] Thomas Stahl und M. Völter. *Model-driven software development*. John Wiley & Sons New York, 2006.

[W3C04] Extensible Markup Language (XML) 1.0. Notation. Bericht, W3C, 2004. http://www.w3.org/TR/2004/REC-xml-20040204/#sec-notation.

[XML12] XMLVM, 2012. http://www.xmlvm.org/.

[Xte12a] Xtend, 2012. http://www.eclipse.org/xtend/.

[Xte12b] Xtext, 2012. http://www.eclipse.org/Xtext/.