Modellgetriebene Entwicklung von Geschäftsanwendungen – Teil 1: Theoretische Grundlagen, Persistenz

Adieu, teure Softwareentwicklung!

NEUE SERIE

Unternehmen, die selbst Geschäftsanwendungen entwickeln – Stichwort „Eigenentwicklung" – sind mit zahlreichen Problembereichen konfrontiert. Und viele Unternehmen setzen nach wie vor auf eigenständig entwickelte Software, um ihre Unternehmensprozesse optimal unterstützen zu können. Wir analysieren die typischen Probleme und schlagen in diesem Artikel neue Konzepte zur Lösung vor. Mit einer domänenspezifischen Sprache (DSL) zeigen wir anhand eines Beispiels, wie Geschäftsanwendungen modellgetrieben implementiert werden können.

von Daniel Stieger, Tom Schindl und Wolfgang Messner

Unternehmen sind ständig darauf bedacht, ihre Leistungserstellung zu optimieren. Software spielt dabei eine zentrale Rolle. Und obwohl es unzählige ERP-(Enterprise-Ressource-Planning-), CRM-(Customer-Relationship-Management-) und PPS-(Production-Planing-System-) Lösungen gibt, nimmt auch die Zahl von Eigenentwicklungen nicht ab. Viele Unternehmen benötigen kein Standardprodukt. Sie brauchen ihre eigene, auf sie zugeschnittene Software. Nur so können sie effizient und wirtschaftlich Unternehmensprozesse unterstützen.

Die meisten großen Unternehmen differenzieren daher: Wenn möglich, wird bei der Automatisierung von Standardprozessen auf Standardsoftware zurückgegriffen. Bei Kernprozessen, die das Unternehmen am Markt differenzieren, wird aber nach wie vor Software selbst entwickelt.

Software zu entwickeln ist komplex und teuer. Insbesondere bei umfangreichen Anwendungen ist Softwareentwicklung nicht einfach mit „programmieren" gleichzusetzen. Für erfolgreiche Eigenentwicklungen ist ein weites Themenfeld zu berücksichtigen: von der Anforderungsanalyse, Softwarearchitektur und Koordination von Entwicklerteams über Releasemanagement, systematische und automatisierte Tests bis hin zum Verfassen zielgruppengerechter Dokumentation. Gerade diese Aktivitäten der Softwareentwicklung gehören meist nicht zu den Kernkompetenzen von Unternehmen. Uns wird häufig über die in Tabelle 1 zusammengefassten Problembereiche berichtet.

Vor dem Hintergrund dieser Probleme ist es nicht verwunderlich, dass für Softwareentwicklungsprojekte häufig mehr Zeit aufgewandt wird als veranschlagt. Dies führt nicht nur zu höheren Entwicklungskosten, sondern mitunter werden auch die Anforderungen nicht richtig erfüllt. Es muss nachgebessert werden, Kosten und Projektdauer steigen. Auch der bei Softwareprodukten geforderte hohe Grad an Investitionssicherheit lässt sich nicht realisieren, wenn langfristig die Potenziale von neuen Technologien nicht erschlossen werden können. Schwierige Wartung und unzureichende Dokumentation können dazu führen, dass Eigenentwicklungen zwar noch funktionieren, aber ungern weiterentwickelt werden.

Doch mit welchen Konzepten lassen sich die Kosten von Eigenentwicklungen senken und dabei der Grad der Anforderungserfüllung und die Investitionssicherheit erhöhen? Wie lässt sich Software für Geschäftsanwendungen deutlich schneller und besser entwickeln als bisher? Wie lässt sich die Produktivität bei der Entwicklung von Geschäftsanwendungen steigern, in einem Ausmaß, das auch einen deutlichen Unterschied erkennen lässt? Wir schlagen alte und bekannte Ideen vor, um diese Probleme anzugehen. Im Wesentlichen zählen dazu Abstraktion und Strukturierung, die nun durch moderne

Artikelserie

Teil 1: Theoretische Grundlagen, Persistenz
Teil 2: Geschäftslogik, Präsentation, Entwicklungsumgebung

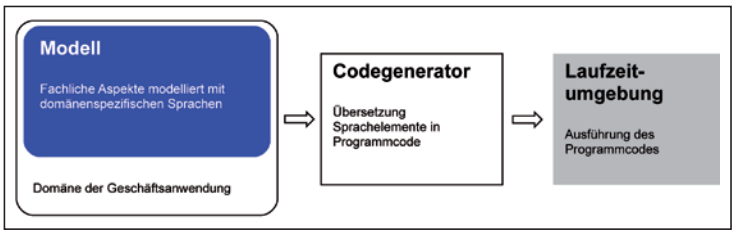


Abb. 1: Modell und Codegenerierung

Entwicklungswerkzeuge zur Softwaremodellierung gezielt eingesetzt werden können.

Unter Abstraktion verstehen wir das Weglassen von unwesentlichen Merkmalen. Wesentliches einer Geschäftsanwendung soll mit ausdrucksstarken Sprachelementen formuliert werden können. Objektorientierte Programmiersprachen wurden geschaffen, um verschiedenste Probleme zu lösen – nicht nur Geschäftsanwendungen. Begriffe wie Klasse, Methode oder statische Methode sind viel zu allgemein und werden nur von Programmierern verstanden. Wir schlagen eine domänenspezifische Sprache (DSL) für Geschäftsanwendungen mit Sprachelementen wie Formular, Kommando oder Prozess vor. Diese Sprachelemente repräsentieren Konzepte, die sich bei dieser spezifischen Art von Anwendungen ständig wiederholen. Diese Sprachelemente lassen sich etablieren, sodass sie bei den Beteiligten mit klaren Vorstellungen verbunden sind. Eine gemeinsame Basis zur Kommunikation wird geschaffen.

Als zweite wesentliche Idee greifen wir die Strukturierung auf. Jedes größere System wird in kleine, überschaubare und beherrschbare Modelle gegliedert, die klar definierte Anforderungen abbilden. Die Modelle werden aus den spezifizierten Sprachelementen gebildet. Auf diese Weise kann ein Modell „Artikelverwaltung“ erstellt werden, das von beliebigen anderen Modellen wiederverwendet werden kann (Modellimport). Es lassen sich Modelle parallel entwickeln, einzeln testen und dadurch langfristig besser warten und erweitern. Für Anwendungsentwickler ist es einfach, Modelle von anderen zu verstehen, die mit der gemeinsamen Sprache verfasst wurden.

Zu einem späteren Zeitpunkt wird aus den Sprachelementen durch Codegeneratoren automatisch ausführbarer Programmcode generiert, d. h. Codegeneratoren übersetzen die spezielle Sprache in klassischen Code. Technische Details einer Implementierung werden in Codegeneratoren verlagert. Wir unterscheiden daher zwischen fachlichen Aspekten der Software, die mit Sprachelementen erfasst werden, und den technischen Aspekten der Anwendungen. Die technischen Aspekte wie Systemarchitektur, Bibliotheken und Plattformen/Laufzeitumgebungen werden in Codegeneratoren abgedeckt, die noch dazu für die notwendige Standardisierung sorgen. Dadurch strukturieren wir die Umgebung, in der ein Anwendungsentwickler arbeitet. Statt sich mit technischen Details oder der technischen Architektur zu beschäftigen, greift der Anwendungsentwickler auf standardisierte Sprachelemente für Geschäftsanwendungen zurück.

Abbildung 1 verdeutlicht die Generierung von Programmcode aus einem Modell. Der Code wird wiederum in einer Laufzeitumgebung ausgeführt. Die

Problembereich	Beschreibung
Verständnis	Die Zusammenarbeit von Auftraggebern und Entwicklern gestaltet sich bis dato problematisch. Man redet aneinander vorbei. Wenn eine einheitliche Sprache fehlt, werden für Konzepte verschiedene Begriffe verwendet. Ein gemeinsames Verständnis für das Aufgabenfeld kann nur schwer geschaffen werden.
Anforderungen	Häufig fehlt auch dem Auftraggeber ein klares Bild seiner Anforderungen. Letztere müssen daher im Entwicklungsprojekt erst erarbeitet werden; der Auftraggeber muss sich erst nochmals intensiv mit seinen Anforderungen befassen. Eine Systematik dafür fehlt aber.
Systemdesign	Applikationsarchitektur ist unpassend oder verwässert über die Zeit. Wenige Entwickler verfügen über das Wissen und die Erfahrung, effektive Leitlinien vorzugeben. Die Applikationsarchitektur unterscheidet sich zwischen den Entwicklern und ist von Projekt zu Projekt verschieden. Konsistenz fehlt.
Silo-Denken	Code von anderen Entwicklern zu verstehen, gestaltet sich schwierig. Es fehlt an klaren Richtlinien. Ein Silo-Denken greift Platz, bei dem nicht ein Team, sondern jeweils ein Entwickler ein Projekt erarbeitet. Nur dieser Entwickler kennt seine Geschäftsanwendung im Detail; nur er weiß, in welcher Klasse was implementiert wurde.
Testen	Software geordnet zu testen, erhöht die Qualität entscheidend, wird in der Praxis dennoch häufig vernachlässigt – sie wird dann einfach durch den Anwender getestet. Richtigerweise sollte es bereits bei der Formulierung der Anforderungen möglich sein, mit minimalem Aufwand Tests zu schreiben, um die Anforderungen zu prüfen (Anforderungstests). Aber auch einzelne Komponenten der Software müssen mit geringem Aufwand testbar sein (Unit Tests).
Wartbarkeit	Meist ändern sich die Anforderungen an Geschäftsanwendungen über die Zeit hinweg. Softwareentwickler sind noch viele Jahre in der Verantwortung, wenn es darum geht, neuen und sich ändernden Anforderungen nachzukommen. Statt „Meet the ongoing Business-Requirements“ hört man bei vielen Eigenentwicklungen dann aber „Bitte nicht mehr angreifen“.
Dokumentation	Die Dokumentation ist unzureichend oder fehlt völlig. Eigentlich ist die Dokumentation – so wie das Testen – in den Entwicklungsprozess zu integrieren. Dazu gehören auch konsistente Bezeichnungen, Tooltipps und Hilfen.
Unabhängigkeit	IT-Infrastrukturtechnologien ändern sich ständig. Allein bei Benutzeroberflächen ist eine kaum überschaubare Anzahl festzustellen. Neben Bibliotheken für Weboberflächen, Mobile und Tablet werden auch im Desktopbereich ständig Innovationen hervorgebracht. Diese Potenziale zu erschließen, gestaltete sich bisher schwierig, da die Implementierungen meist an eine konkrete Technologie gebunden sind.

Tabelle 1: Problembereiche bei Eigenentwicklungen

Eigenentwicklungen eines Unternehmens fokussieren sich nur noch auf die fachlich richtige Erstellung der notwendigen Modelle.

Wir erreichen zum einen eine klare Trennung von Zuständigkeiten (*separation of concerns*) in fachliche und technische Belange, zum anderen lassen sich die Codegeneratoren aber auch austauschen. Das schafft zusätzliche Investitionssicherheit. Das in Modelle gegossene fachliche Wissen ist Kern der Anwendungsentwickler im Unternehmen. Die Modelle lassen sich durch Codegeneratoren auf verschiedenen Plattformen migrieren. Die Erstellung der „technischen" Codegeneratoren ist Aufgabe von Softwarespezialisten.

Wir haben uns ausführlich Gedanken über passende Sprachelemente gemacht, mit denen Geschäftsanwendungen modelliert werden können. Mithilfe unserer domänenspezifischen Sprachen lassen sich Anwendungen in einem Modell formulieren. Beteiligte konzentrieren sich auf die fachlichen Aspekte, nicht auf technische Implementierungsdetails. Die sind durch eine automatische Transformation von Modellen in lauffähige Anwendungen standardisiert. Die Konzentration auf Wesentliches beschleunigt die Entwicklung und erhöht die Qualität.

Von der Theorie zur Praxis

Zugegeben: Die diskutierten Ideen klingen plausibel, aber theoretisch. Wie lassen sie sich nun konkret umsetzen? Im Folgenden zeigen wir, wie Abstraktion und Strukturierung bei Geschäftsanwendungen aussehen kann. Wir zeigen ein Beispiel, bei dem wir eine Software zur Rechnungskontrolle mit einem Kunden umgesetzt haben. Die Software wurde dabei vollständig modelliert. Alle Anforderungen wurden durch die Sprachelemente der von uns entwickelten DSL erfasst.

Bei **Abbildung 2** und **3** handelt es sich um Screenshots der Rechnungskontrollsoftware. Die Software verwaltet im Wesentlichen so genannte Rechnungskontrollakte. Diese Akte können mehrere Rechnungen und Lieferscheine enthalten, die – um einen Akt abschließen zu können – sich in ihrer Summe auf 0 auflösen müssen. Der Wert aller Rechnungen muss dem Wert aller Lieferscheine im Akt entsprechen.

In Bezug auf die Investitionssicherheit bestanden beim Kunden klare Vorstellungen. Die Software sollte vorerst als einfacher Fat-Client ausgeliefert werden, um auf einen Applikationsserver verzichten zu können. Als Programmiersprache war Java gesetzt, allerdings kein Java EE, keine komplexen Bibliotheken zur Persistierung (z. B. Hibernate), keine BPM- oder SOA-Werkzeuge. Grundsätzlich sollte Java 7 mit wenigen zusätzlichen Technologien zum Einsatz kommen, um notwendiges technisches Wissen für Erweiterung und Wartung seitens des Kunden in Grenzen zu halten. Dennoch wurde eine offene Architektur gefordert, um gegebenenfalls zu

Anzeige

Jetzt abonnieren und 3-fach profitieren!

Print | Apps | PDF

Ihre Vorteile auf einen Blick:

1. Frei-Haus-Lieferung des Print-Magazins!
2. Zugriff auf die Magazine mit der Mobile-Technology-App für iOS und Android
3. Offline-PDF-Export mit der Intellibook-ID

www.mobiletechmag.de

S&S Media Group

Abb. 2: Oberfläche der Rechnungskontrollsoftware

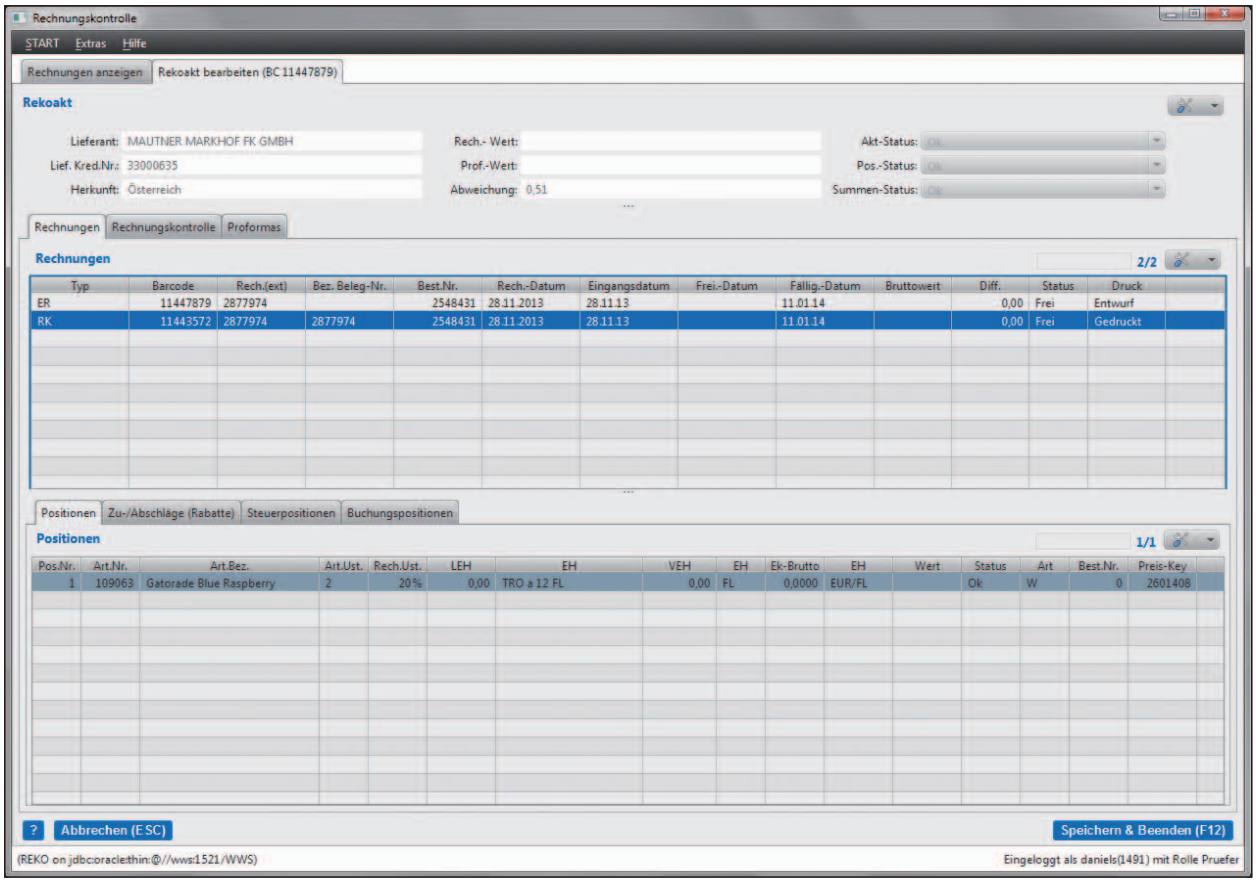
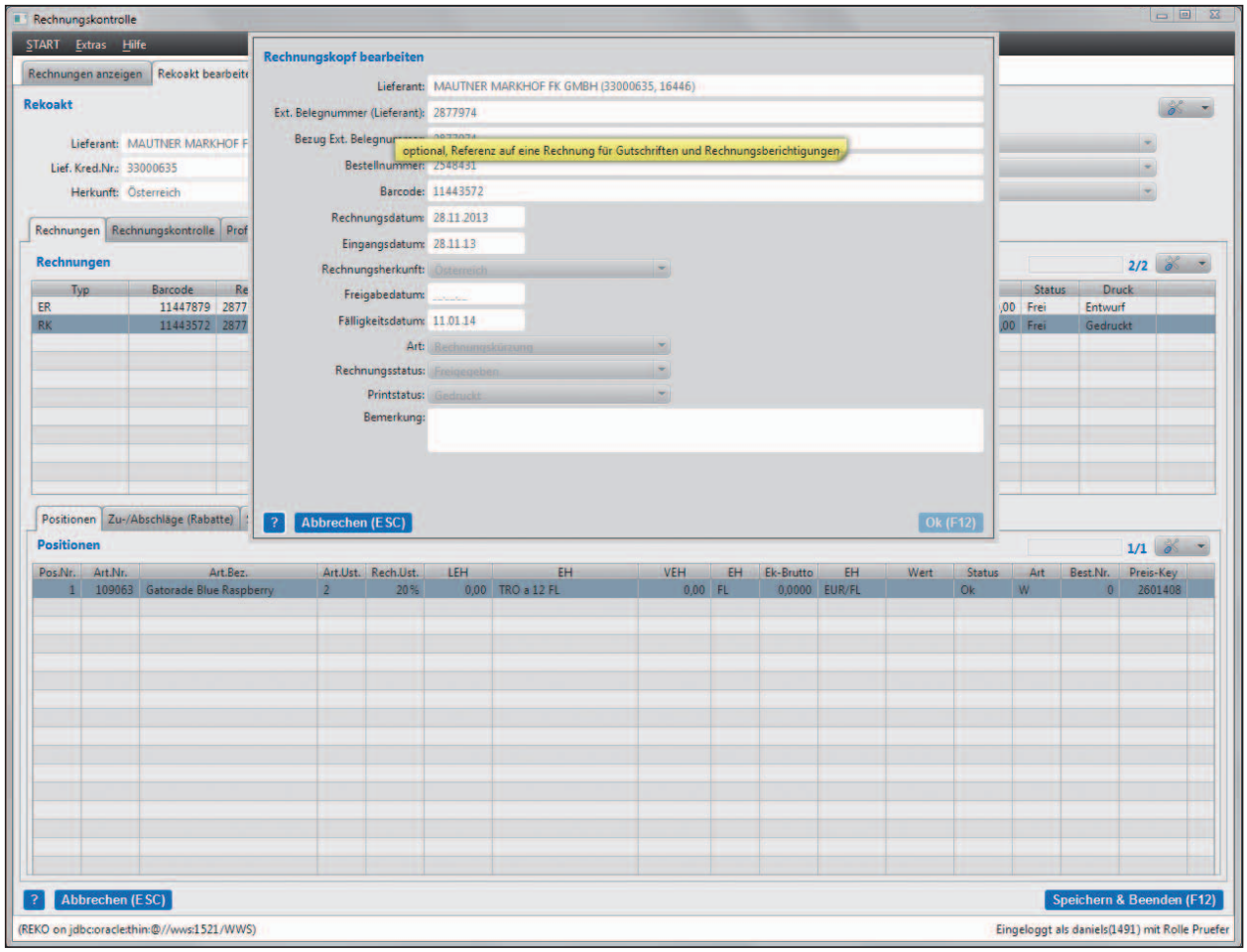


Abb. 3: Oberfläche der Rechnungskontrollsoftware mit Formular



Entity Rechnung extends <no super>
// Abbildung einer Rechnung im Rahmen des WarenWirtschaftssystems
WOME, initial draft 06.12.2013

Type	Name	Options	Documentation	UseCase 1
int	id	KEY	Primary Key / interne Rechnungsnummer	1
int	intBelegNummer	<opts>	interne, eindeutige Belegnummer der Rechnung (Barcode)	3889177
RekoAkt	rekoAkt	<opts>	Referenz auf Rechnungsakt, zu der Rechnung gehoert	null
Lieferant	lieferant	<opts>	<doc>	4772
string	extBelegNummer	<opts>	Rechnungsnummer des Lieferanten	331533
int	bestellNummer	<opts>	interne Bestellnummer zur Rechnung (nur bei Einzelrechnungen)	3889177
DateTime	rechnungsDatum	<opts>	Rechnungsdatum des Lieferanten	01.12.2012
DateTime	eingangsDatum	<opts>	Valutadatum (Datum, ab dem das Zahlungsziel beginnt) bei Papierrechnungen entspricht das Eingangsdatum dem Valutadatum	05.12.2012
DateTime	faelligKeitsDatum	<opts>	Datum, an dem die Rechnung zur Zahlung fällig ist	31.12.2012
DateTime	freigabeDatum	<opts>	Datum, an dem die Rechnung/Rekoakt in der Reko freigegeben wurde	28.12.2012
PrintStatus	printStatus	<opts>	Beim ersten Ausdruck der freigegebenen Rechung => Gedruckt. Danach Nachdruck	not printed
Herkunft	herkunft	<opts>	Die Herkunft bestimmt die möglichen Steuerklassen und somit die Buchungskonten	06.11.1900
RechnungsTyp	typ	<opts>	Hauptrechnung, Gutschrift, Zusatzrechnung, Rechnungsberichtigung	Hauptrechnung
RekoEinstellung	eigenschaft	<opts>	<doc>	null
string	bemerkung	LENGTH(0 - 1000)	interne Notiz zur Rechnung	UseCase 1
list<RechPos>	rechPositionen	<opts>	Alle Positionen, die auf dieser Rechnung gelistet sind	
list<SteuerPos>	steuerPositionen	<opts>	Set -> Eindeutig ueber Buchungskonto	
list<ZuAbPos>	zuAbPositionen	<opts>	Set -> Eindeutig ueber ZuAb.Art	
RechnungsStatus	status	<opts>	Status der Rechnung (angelegt | erfasst | freigegeben) angelegt -> erfasst: Der Rechnungskopf ist richtig ausgefüllt und die Summe der Positionen entspricht den Summenzeilen	angelegt
Status	summenStatus	<opts>	Summe ok oder nicht OK	nicht ok
BigDecimal	nettoWertSoll	<opts>	<doc>	
BigDecimal	steuerWertSoll	<opts>	<doc>	
BigDecimal	nettoWarenWert	<opts>	<doc>	
BigDecimal	nettoZuAbWert	<opts>	<doc>	
BigDecimal	mengeVEH	<opts>	Menge für die Verrechnungseinheit	

Abb. 4: Modellierung einer Entität Rechnung mit Testdaten

einem späteren Zeitpunkt auf andere Plattformen migrieren zu können (bspw. Weboberflächen mit Applikationsserver). Langfristig wird das auch angestrebt.

Die DSL selbst wurde anhand der Rechnungskontrolle und ähnlich gelagerten Projekten konzipiert, mit dem Ziel, zukünftig allgemein Geschäftsanwendungen modellieren zu können. Diese Vorgehensweise hat uns ermöglicht, direkt beim Kunden zu lernen und alle Anforderungen dieser Domäne unmittelbar selbst zu erleben. Im Vergleich zu anderen Programmiersprachen auf hoher Abstraktionsebene (bspw. 4-GL-Sprachen) sind die von uns konzipierten Sprachelemente noch spezifischer auf die Domäne der Geschäftsanwendung zugeschnitten. Wir bezeichnen diese Form von Geschäftsanwendungen als *dokumentenzentrierte Anwendung*.

Als erste Strukturierung haben wir uns an die klassische Drei-Schichten-Architektur angelehnt. Dabei unterscheidet man (1) die Persistenzschicht, die verwendete Datenstrukturen festlegt und diese Datenstrukturen auf Datenbanken speichern und laden kann, (2) die Geschäftslogikschicht, in der die Verarbeitungsmechanismen und die Logik der Anwendung abgebildet werden und (3) die Präsentationsschicht, die Benutzerschnittstellen abbildet. Die Drei-Schichten-Architektur ist weit verbreitet, aber als Strukturierungselement sehr grob. Wir haben daher jede der drei Schichten genau untersucht, um spezifische Sprachelemente je Schicht zu bestimmen. Nachfolgend und im zweiten Teil dieser Artikelserie gehen wir auf jede der Schichten einzeln ein.

Die Persistenzschicht

Da Geschäftsanwendungen häufig Daten von Geschäftsprozessen erfassen, verändern und prüfen, stehen Daten und Datenstrukturen bei der Entwicklung häufig im Mittelpunkt. Je nach Aufgabenstellung sind die Datenstrukturen entsprechend umfangreich und komplex – wir möchten darüber nicht hinwegtäuschen. Einerseits ist nicht von Beginn an klar, welche Daten erfasst und bearbeitet werden müssen. Andererseits ist es auch nicht einfach, eine geeignete Struktur zu definieren, in der die Daten dann auch effizient abgelegt und in der Software einfach manipuliert werden können. Die Entwicklung geeigneter Datenstrukturen bei Geschäftsanwendungen ist eine komplexe Aufgabe und lässt sich unserer Ansicht nach auch nicht durch Modellierung vorwegnehmen. Allerdings lässt sich die Entwicklung von Datenstrukturen vereinfachen, wenn man sich dabei auf das Wesentliche konzentrieren kann.

Die Modellierung von Datenstrukturen haben wir mit dem Sprachelement „Entität“ vereinheitlicht. Neben einem eindeutigen Namen besitzen Entitäten verschiedene Datenfelder, die einen Typ, einen Namen, Optionen und Dokumentation enthalten. **Abbildung 4** zeigt als Beispiel eine Rechnung in Tabellendarstellung – ein Screenshot aus der Entwicklungsumgebung. In dieser Tabelle können Beteiligte die Datenstruktur bearbeiten. Wie zu erkennen, können DSLs auch spezielle Notationen (in diesem Fall eine Tabelle) verwenden. Das erhöht die Lesbarkeit und ist so besser für eine Diskussion geeignet. Gerade Beteiligte aus Fachabteilungen, die mit

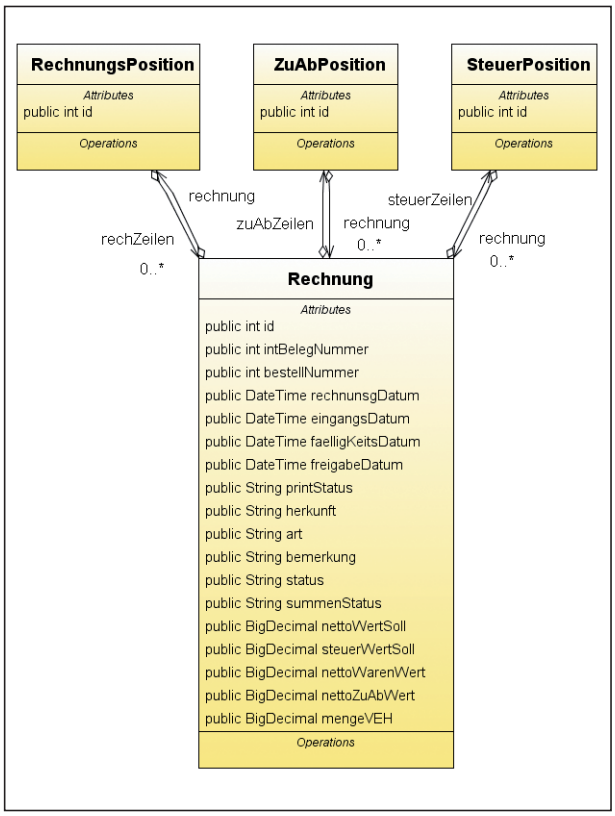


Abb. 5: Darstellung der Rechnung als UML-Klassendiagramm; vereinfacht

```
CHECKOUT Rechnung checkoutRechnung(int id) {
  Rechnung rech = RechnungMapping <join> get(id) Checkout;

  rech.rechnungPositionen = RechnungsPositionMapping <join> where({~rechpos,  => rechpos.rechnung.id == rech.id  })
      Checkout;

  rech.steuerPositionen = SteuerPositionMapping <join> where({~steuerpos,  => steuerpos.rechnung.id == rech.id  })
      Checkout;

  rech.zuAbPositionen = ZuAbPositionMapping <join> where({~zuabpos,  => zuabpos.rechnung.id == rech.id  }) Checkout;

  rech;
}
```

Abb. 6: Aufbau eines Rechnungsgraphen mit Datenbankabfragen

```
map Rechnung as RechnungenMapping
on table "REKO_RECHNUNG" options OPTIMISTIC_LOCK
```

Property	Table fieldname	Options
int id	"KEY_RECHNUNG"	KEY, AUTOID("SREKO_RECHNUNG"), NOTNULL
int intBelegNummer	"NUM_BELEG"	<opts>
ref RekoAkt rekoAkt: int id	"REF_REKOAKT"	<opts>
ref Lieferant lieferant: int id	"REF_LIEFERANT"	<opts>
string refExtBelegNummer	"REF_EXT_BELEG"	<opts>
string extBelegNummer	"NUM_EXT_BELEG"	<opts>
int bestellNummer	"REF_BESTELLUNG"	<opts>
DateTime rechnungsDatum	"DAT_RECHNUNG"	<opts>
DateTime eingangsDatum	"DAT_VALUTA"	<opts>
DateTime faelligKeitsDatum	"DAT_FAELLIG"	<opts>
DateTime freigabeDatum	"DAT_FREIGABE"	<opts>
status printStatus	"COD_PRINT_STATUS"	<opts>
status herkunft	"COD_RECHNUNGSHERKUNFT"	<opts>
status typ	"COD_RECHNUNGSART"	<opts>
ref RekoEinstellung eigenschaft: int id	"REF_RECHEIGENSCHAFT"	<opts>
string bemerkung	"TXT_BEMERKUNG"	<opts>
status status	"COD_RECHNUNG_STATUS"	<opts>
status summenStatus	"COD_SUMMEN_STATUS"	<opts>
BigDecimal nettoWertSoll	"BTR_SOLL_NTO"	<opts>
BigDecimal steuerWertSoll	"BTR_SOLL_STEUER"	<opts>
BigDecimal nettoWarenWert	"BTR_WARE_NTO"	<opts>
BigDecimal nettoZuAbWert	"BTR_ZUAB_NTO"	<opts>
BigDecimal mengeVEH	"MGE_VEH"	<opts>

Abb. 7: Mapping zwischen Tabelle und Entität

Programmiercode nicht vertraut sind, finden eine derartige Darstellung besonders ansprechend.

Neben Typ, Name, Optionen und Dokumentation enthält die Tabelle auch (Test-)Daten für konkrete Anwendungsfälle. Datenstrukturen und erste Daten für einen Anwendungsfall können gleichzeitig aufgebaut werden. Von Beginn an wird die Datenmodellierung durch einen konkreten Anwendungsfall geleitet. Zu einem späteren Zeitpunkt werden diese Daten auch gleich als Testdaten herangezogen, um die Software anhand der konkreten Anwendungsfälle zu testen.

Die in einer Entität enthaltene Dokumentation wird automatisch in der Benutzeroberfläche angezeigt, wann immer eine Rechnungsentität angezeigt wird (z. B. in einem Formular über Tooltipps). Neben der Dokumentation werden auch so genannte Metainformationen zu den Datenfeldern erfasst, bspw. der Wertebereich von Zahlen oder die minimale und maximale Länge von Zeichenketten. Die Metainformationen werden bei der Validierung von Formulardaten, aber auch zur Generierung eines DB-Schemas verwendet.

Jede Geschäftsanwendung benötigt umfangreiche Datenstrukturen. Diese Datenstrukturen können einfach durch Entitäten abgebildet werden, die untereinander durch Referenzen und durch Listen verbunden werden. So enthält die oben abgebildete Rechnung bspw. drei Listen von weiteren Entitäten; in diesem Fall mehrere *RechnungsPositionen*, mehrere *SteuerPositionen* und mehrere *ZuAbPositionen* für Rabatte. Das heißt, aus einer Rechnungsentität wird ein ganzer Instanzengraph. Jede einzelne Entität wird in einer Tabelle modelliert; sie können aber auch gemeinsam als UML-Klassendiagramme exportiert und angezeigt werden (**Abb. 5**). Die grafische Darstellung ist besonders zur Dokumentation geeignet.

Im Zusammenhang mit der Persistenzschicht ergibt sich damit unmittelbar eine weitere Aufgabenstellung: Wie werden Daten aus Datenbanktabellen geladen, in Entitäten abgepackt und anschließend untereinander verbunden, d. h. in einen Graphen überführt? Diese Aufgabe übernimmt ein so genanntes Repository. Dieses verfügt über einen Datenbankzugriff, kann Abfragen durchführen und auch den gewünschten Graphen zusammensetzen [1]. In **Abbildung 6** wird gezeigt, wie in einem Repository (genauer: in einer Repository-Funktion) eine Rechnung über ihren Schlüssel *id* aus der Datenbank geladen wird.

Erst erfolgt eine Datenbankabfrage, bei der über *RechnungenMapping* die Rechnung mit dem Schlüssel *id* geladen wird. Dann werden Rechnungspositionen, Steuerpositionen und *ZuAbPositionen* der Rechnung hinzugefügt, wodurch ein Rechnungsgraph entsteht. Ein Mapping (z. B. das *RechnungenMapping* in **Abbildung 6**) gibt dabei an, von welcher Datenbanktabelle die Entität geladen wird. Das *where*-Element entspricht einem SQL-*Where*-Statement, formuliert mit moderner Closure-Syntax und mit voller IDE-Unterstützung. Checkout am Ende der Anweisung lädt die Entitäten mit Schreibrechten – man könnte auch *read-only* laden.

In **Abbildung 7** ist das *RechnungenMapping* abgebildet. Rechnungsentitäten werden auf der Tabelle *REKO_RECHNUNG* persistiert, das Datenfeld *id* ist dabei Datenbankschlüssel und in der Tabellenspalte *KEY_RECHNUNG* abgelegt.

Mit den vorgestellten Modellelementen wurde die Persistenzschicht sauber strukturiert. Daten werden mit Entitäten modelliert. Neben Dokumentation werden auch Daten für Anwendungsfälle festgehalten, die bereits von Beginn an konzeptionelle Überlegungen unterstützen. Die Daten dienen zugleich Anwendungstests. Mappings bilden die Beziehung zwischen Datenbanktabellen und Entitäten ab. Sehr einfach können in Repositories über Mappings Datenbankabfragen gestellt und die Entitätsinstanzen zu einem Graphen verbunden werden. Die Modellelemente Repository und Mapping sind auch für die Speicherung von Entitäten verantwortlich.

Daniel Stieger ist Partner der modellwerkstatt.org. Er beschäftigt sich seit mehreren Jahren mit Model-driven Software Development (MDSD) und entsprechenden Werkzeugen im Enterprise- und Embedded-Umfeld. An der Universität Innsbruck forscht er im Bereich Innovation und Strategie.

Tom Schindl ist CTO von BestSolution.at, ein auf Eclipse und JavaFX spezialisiertem Softwarehaus in Österreich. Er ist e4-Committer und Projektlead des e(fx)clipse-Projekts.

Wolfgang Messner ist Partner der modellwerkstatt.org. Er beschäftigt sich intensiv mit modellgetriebener Softwareentwicklung im Maschinenbau/Automatisierungstechnik und mit der Modellierung von Unternehmensprozessen und Geschäftsanwendungen.

Link & Literatur

[1] Evans, Eric (2003): „Domain Driven Design“, Addison-Wesley Longman, Amsterdam

[2] Jetbrains 2013: http://www.jetbrains.com/mps (aufgerufen am 10.10.2013)

Anzeige