Teil 2: Geschäftslogikschicht und Benutzerschnittstelle

Adieu, teure Softwareentwicklung!

Nachdem wir im ersten Teil Konzepte der modellgetriebenen Entwicklung betrachtet haben, geht es diesmal um Geschäftslogik und Benutzerschnittstelle. Für ihre Strukturierung haben wir die Begriffe Kommando und Prozess gewählt. Jede Logik muss in einem Kommando abgebildet werden. Mehrere Kommandos werden einem Prozess zugeordnet. Die Logik selbst wird mit ausdrucksstarken Sprachelementen für Mengenoperationen formuliert, die auf Entitätsgraphen angewendet werden.

von Daniel Stieger, Tom Schindl und Wolfgang Messner

Kommandos sind Funktionseinheiten, die Geschäftslogik abbilden und zusätzlich eine Schnittstelle zur Benutzeroberfläche aufweisen. Kommandos arbeiten nicht nur gegebene Logik ab, sie können auch Benutzerinteraktionen aufnehmen. Kommandos lassen sich immer durch *Undo* rückgängig machen. Das Hinzufügen einer Rechnungsposition zu einer Rechnung kann als Beispiel für ein Kommando dienen. Dabei bezieht sich das „Hinzufügen" nicht ausschließlich auf die Position zum Graphen. Der Benutzer soll erst die Rechnungsposition bearbeiten, bevor sie der Rechnung hinzugefügt wird. Das entsprechende Kommando ist in **Abbildung 1** veranschaulicht.

Die Logical View rechts in der Abbildung entspricht einer Projektansicht – mehrere Kommandos wurden bereits modelliert. Das Kommando *Rechnungsposition hinzufügen* ist einfach gehalten. Im Überblick lässt sich Folgendes festhalten (beginnend in **Abbildung 1** links oben):

- Das Kommando ist dem Prozess *Rechnungsverwaltung* zugeordnet. Eine aktuelle Instanz der Entität Rechnung wird dem Kommando als *rechnung* übergeben.
- Keine weiteren Parameter werden übergeben.
- Eine lokale Variable *rechPos* ist verfügbar.
- Der Kommandotyp ist auf *GRAPH_ADD* gestellt, da er der Rechnung eine Position anhängt. Das Kommando steht immer zu Verfügung (keine *Enabled-If*-Bedingung), das Kommando kann ohne Rückfrage abgebrochen werden (*question on external abort*), der Name des Kommandos in der Benutzeroberfläche lautet *Rechnungsposition hinzufügen* (wie das Kommando selbst; kein *title addon* gesetzt) und als Icon für das Kommando ist *ICON_NEW* zu verwenden.
- Keine Logik ist bei der Initialisierung des Kommandos (*command init*) auszuführen.
- Als Seite für das Kommando ist *PageDefault* zu verwenden, d. h. dieses Kommando verfügt – im Gegensatz zu einem Wizzard – nur über eine Seite.

Kommandos können ein oder mehrere Seiten aufweisen. Mit ihnen werden mögliche Benutzerinteraktionen abgebildet. In einer Seite wird allerdings nicht das Aussehen der Benutzeroberfläche spezifiziert, es werden lediglich die Daten zur späteren Anzeige in der Oberfläche vorbereitet. Zusätzlich wird in der Seite angegeben, wie die Benutzeroberfläche beendet wird bzw. wie das Kommando beendet werden kann. In der Seite *PageDefault* der **Abbildung 1** ist Folgendes modelliert:

- Die Seite benötigt eine Benutzeroberfläche, z. B. ein Formular, das *RechnungsPositionen* erwartet (*form bound to*).
- Beim Laden dieser Seite wird eine Rechnungsposition erzeugt und an die Benutzeroberfläche übergeben (*page init*).
- Es wird ein Scope – eine Auswahl – berechnet und eingestellt. Dadurch wird festgelegt, welche Artikel dem Benutzer in der Oberfläche als Artikelauswahl zu Verfügung stehen (**Abb. 2**). Die Artikelentitäten werden aus einem Repository geladen.
- Die Seite kann mit '*Ok*' beendet werden (*conclusion*). Das Formular wird dann zum Speichern aller Benutzereingaben in die Rechnungspositionentität angewiesen (*request 'save data' from page form*). Die Rechnungsposition wird der Rechnung angehängt

Artikelserie

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

```
command 'Rechnungsposition hinzufügen' in process Rechnungsverwaltung with Rechnung rechnung

  parameter:
    <no params>

  local variables:
    RechnungsPosition rechPos

  documentation and help text:
  // 'Bedingung: ',
  'Status der Rechnung muss angelegt sein' //


  command type: GRAPH_ADD
  enabled if: <cond>
  question on external abort: <msg>
  title addon: <msg>
  command icon: HafinaDefaults.ICON_NEW

  command init:
    <func>


  command pages:
     page 'PageDefault'
      form bound to list< RechnungsPosition > as  form

      page init:
        pageLoadFunc()->Object {
          rechPos = new RechnungsPosition();
          rechPos;
        }

      set scopes for page:
        pageSetScopesFunc()->void {
          rechPos.artikel#Meta.setScope(call StammdatenRepo.findAktiveArtikelZuLieferant(rechnung.lieferant#Key) );
        }


      conclusions:
        conclusion 'Ok'  (enabled if: <cond>)
          request 'save data' from page form: save hotkey: SAVE
          func()->void {
            rechnung.mergeRechPos(rechPos);

            done (run FINAL_OK_CONCLUSION)
          }
```

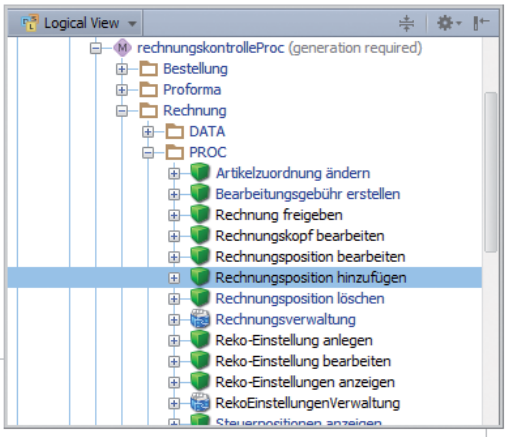

Abb. 1: Kommando, um eine Rechnungsposition hinzuzufügen

(über die Methode *mergeRechPos()*); anschließend wird das Kommando beendet (*done*).

Selbstverständlich kann der Benutzer jedes Kommando abbrechen, das muss nicht explizit modelliert werden. In **Abbildung 2** ist die Benutzeroberfläche während der Ausführung des Kommandos zu sehen. Das Kommando wurde gestartet, eine Rechnung und ein Formular zum Bearbeiten der Position wurden beim Starten als Parameter übergeben.

Neben den Kommandos kann ein Prozess als Zustandsmaschine modelliert werden. In **Abbildung 3** ist die Rechnungsverwaltung dargestellt. Die Rechnung kann drei Zustände annehmen: *Angelegt*, *Erfasst* und *Freigegeben*. Je nach Zustand sind für den Anwender unterschiedliche Kommandos freigegeben. Zusätzlich ist im Zustand *Angelegt* auch vermerkt, dass automatisch auf den Zustand *Erfasst* gewechselt wird, wenn die Bedingung *Rechnung Ok* erfüllt ist. Die Bedingung selbst wurde weiter unten im Detail modelliert, d. h. Bedingungen können während der Modellierung bereits angelegt und textuell beschrieben werden, ohne dass die Logik im Detail spezifiziert wird.

Neben Prozessen und Kommandos lässt sich Geschäftslogik vor allem durch Mengenoperationen adäquat formulieren. Häufig ist eine Zahl von verschiedensten Entitäten (z. B. elektronischen Dokumenten wie etwa Belege) zu sortieren, umzuordnen oder zusammenzufassen. Statt *For*-Schleifen bieten Mengenoperationen für derartige Aufgaben übersichtliche Modellierungsmöglichkeiten (**Abb. 4**).

Wie bei den zuvor vorgestellten Datenbankabfragen werden die Operationen in moderner Closure-Syntax formuliert. *Where* schränkt über eine Bedingung eine Menge ein, *Select* bildet eine neue Menge aus dem spezifizierten Datenfeld, *ReduceLeft* reduziert eine Menge, in diesem Beispiel durch Summation.

Mengenoperationen können mit bekannten Java-Anweisungen (*for*, *while*, *if*, Variablendeklaration etc.) auch kombiniert werden. Die Funktionen in Kommandos laut **Abbildung 1** (z. B. *command init*, *page init*, *conclusion*) bieten dafür entsprechend Platz. Betont werden

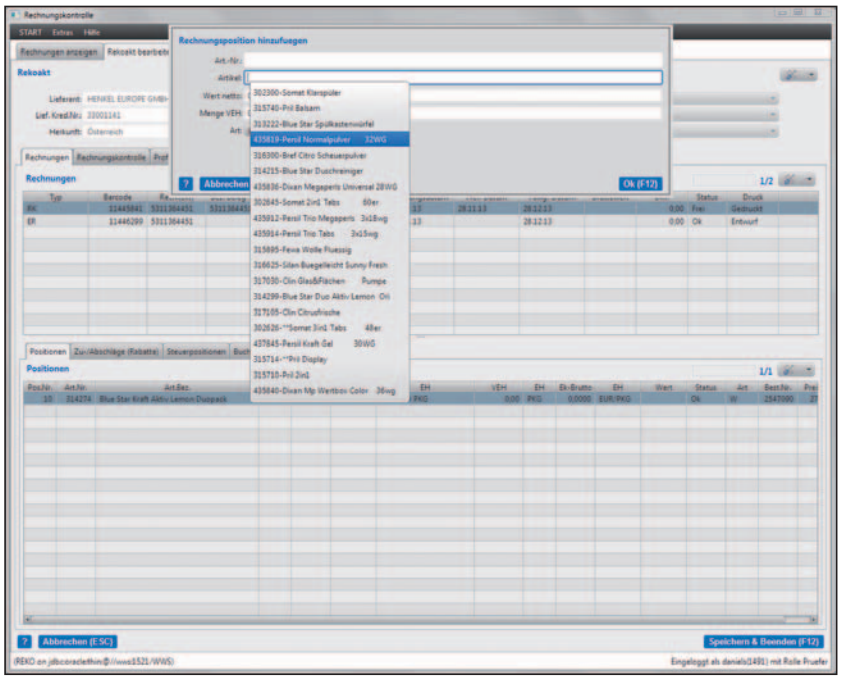

Abb. 2: Hinzufügen einer Rechnungsposition

```
process 'Rechnungsverwaltung'  using Rechnung rechnung process-status-field is status

  <docu>


  creators and state-independent commands:  // and open a session?
    on trigger [<cndt>] Steuerpositionen anzeigen -> < >
    on trigger [<cndt>]+access Steuerpositionen exportieren -> < >


  states:
    state Angeletgt: [<cndt>]
      <docu>
      on entry: <exp>  // can be called multiple times!
      on trigger [<cndt>] Rechnungskopf bearbeiten -> < >
      on trigger [<cndt>] Rechnungsposition bearbeiten -> < >
      on trigger [<cndt>] SummenPosition bearbeiten -> < >
      on trigger [<cndt>] ZuAb-Position bearbeiten -> < >
      on trigger [<cndt>] ZuAb-Position erstellen -> < >
      on trigger [<cndt>] Verpackungsgröße ändern -> < >
      auto [Rechnung Ok] -> Erfasst
      on exit: <exp>


    state Erfasst: [<cndt>]
      <docu>
      on entry: rechnung.abschliessen()  // can be called multiple times!
      on trigger [<cndt>] Rechnung freigeben -> < >
      on trigger [<cndt>] Rechnungskopf bearbeiten -> < >
      on trigger [<cndt>] SummenPosition bearbeiten -> < >
      on trigger [Typ ist Berichtigung] Rechnungsposition löschen -> < >
      on trigger [Typ ist Berichtigung] ZuAb-Position bearbeiten -> < >
      on trigger [Typ ist Berichtigung] Bearbeitungsgebühr erstellen -> < >
      on trigger [<cndt>] Artikelzuordnung ändern -> < >
      on trigger [<cndt>] Verpackungsgröße ändern -> < >
      auto [!(Rechnung Ok)] -> Angeletgt
      auto [Steuerpositionen freigegeben] -> Freigegeben
      on exit: <exp>


    state Freigegeben: [<cndt>]
      <docu>
      on entry: rechnung.freigabeDatum = new_DateTimeFromServer()  // can be called multiple times!
      on trigger [<cndt>]+access Artikelzuordnung ändern -> < >
      on trigger [<cndt>]+access Rechnungskopf bearbeiten -> < >
      on exit: <exp>


  conditions:
    condition 'Rechnung Ok'
      "Rechnungskopf ist richtig ausgefüllt /die Summe der Positionen entspricht den Summenzeilen"
      rechnung.headerOk() && rechnung.summenStatus == SummenStatus.Ok &&
          rechnung.typ != RechnungsTyp.GutschriftAnforderung && rechnung.artikelOk()
      "Gutschriftsanforderungen können nie in den Zustand erfasst wechseln: Löschen oder in GS umwandeln"
```

Abb. 3: Prozess als Zustandsmaschine modelliert

muss allerdings, dass es sich dabei dann um Java-Code handelt, der den Graphen manipuliert. Der Code steht in keiner technischen Abhängigkeit und interagiert auch nicht mit der zugrunde liegenden Java-Infrastruktur.

Zusammenfassend: Eine Geschäftsanwendung wird durch eine Zahl von Kommandos modelliert. Kommandos können Geschäftslogik enthalten und bilden Benutzerinteraktionen ab. Prozesse legen den Ablauf fest und können anhand von Zuständen Kommandos sperren und freigeben. Als wesentliches Sprachelement fehlen nun noch Formulare, um das Aussehen der Benutzerschnittstelle zu modellieren.

Die Benutzerschnittstelle

Benutzerschnittstellen für Geschäftsanwendungen lassen sich besonders effektiv modellieren, da sie sich einfach und effizient logisch abstrahieren lassen. Im Vordergrund stehen Konzepte wie Tabelle, Eingabeformular und verschiedene Layoutvarianten. In **Abbildung 5** ist ein Tabellenformular dargestellt, das eine Liste von Rechnungen anzeigt. Bei den verschiedenen Spalten der Tabelle lässt sich das anzuzeigende Datenfeld, eine Spaltenüberschrift und das Format einstellen. Eine Tabelle stellt jeweils eine Liste von Entitäten dar.

Über der Spaltendefinition wurde ein *on trigger* modelliert, das letztlich eine Schaltfläche in der Oberfläche darstellt, über die ein Kommando gestartet wird. Im obigen Fall wird bei Doppelklick auf eine Tabellenzeile oder durch Drücken der Entertaste (*hk: ENTER*) das Kommando *Rechnungskopf bearbeiten* gestartet. Dabei wird die aktuell selektierte Rechnung als Parameter dem Kommando übergeben. Zusätzlich wird festgelegt, dass für die Seite *Rechnung_Editor* – offensichtlich die einzige Seite des Kommandos – das Formular *RechWizzKopfFC* verwendet werden soll. Mit dem *on trigger*-Sprachelement werden also verschiedene Kommandos über Schaltflächen in die Benutzeroberfläche modelliert. Ob die Schaltfläche aktiviert ist – oder nicht – entscheidet dann der Prozess. Der Vollständigkeit halber ist nachfolgend das angesprochene Eingabeformular „RechWizzKopfFC“ abgebildet.

Das Eingabeformular benötigt nicht eine Liste von Rechnungen, sondern lediglich eine Rechnungsentität. Aufgelistet sind zahlreiche Eingabefelder für die Datenfelder einer Rechnung. Neben einer Kurzbezeichnung können auch Mininim-/Maximum-Werte eingestellt werden. Zusätzlich wird bei jedem Eingabefeld die Dokumentation angezeigt, wie laut Rechnungsentität festgelegt. Schaltflächen zum Beenden des Editiervorgangs werden nicht im Formular modelliert, sondern auf den Seiten der Kommandos (**Abb. 1**).

Der in der **Abbildung 6** verwendete *FormContainer* (links oben) dient nur der Layoutgestaltung, die über

cols/rows eingestellt werden kann. In diesem Beispiel bedeutet "*1*", dass für dieses Formular die maximale Breite, mit "*-1*" die minimal notwendige Höhe, verwendet werden soll.

```
BigDecimal offeneSumme = rechnung.where({~it => it.status == RechnungsStatus.Erfasst; }).
    select({~it => it.nettoWarenWert; }).reduceLeft({~a,~b => a + b; });
```

Abb. 4: Berechnung über Mengenoperationen

Ausgehend von den theoretischen Ideen zur Strukturierung und Abstraktion haben wir in diesem Abschnitt anhand verschiedener Screenshots aus unserer Entwicklungsumgebung konkrete Konzepte/Sprachelemente zur Modellierung von Geschäftsanwendungen vorgestellt. Abgedeckt haben wir dabei die Persistenzschicht, die Geschäftslogikschicht und die Präsentationsschicht. Die Verwaltung von Modellen haben wir beiläufig gestreift. In **Abbildung 1** ist in der Logical View das Modell *rechnungskontrollProc* zu sehen, das die Kommandos zur Bearbeitung von Rechnungen enthält, d.h. das Modell deckt im vorliegenden Fall den kompletten Prozess Rechnungsverwaltung ab. Sowohl Kommandos als auch Datenstrukturen und Formulare können von anderen Modellen importiert und wiederverwendet werden.

Die Entwicklungsumgebung und Codegeneratoren

Als Entwicklungsumgebung für die Modellierung verwenden wir das Meta-Programming-System (MPS) von JetBrains. Die Entwicklungsumgebung bietet umfangreiche Projekt- und Modellunterstützung. Erstellte Modelle können mit Git voll integriert verwaltet werden. JetBrains MPS bietet für die in diesem Artikel vorgestellten Sprachelemente umfangreiche IDE-Unterstützung mit Code Completion.

Die Sprachelemente werden mit Codegeneratoren – ebenfalls in MPS implementiert – in herkömmlichen Java-Code übersetzt. So werden beispielsweise Kommandos, Prozesse, Formulare oder Entitäten direkt in einzelne Java-Klassen übersetzt. Generierte Java-Klassen aus Formularen instanziieren Elemente der JavaFX-Grafikbibliothek. Mappings und Abfragen in Repositories werden in Jdbc Queries, ausgeführt mit der Bibliothek Spring, transformiert. Der generierte Code kann einfach gelesen werden, im Notfall ist auch ein Debugging möglich. Für den Build- und Deployment-Prozess können alle bekannten Java-Tools weiterhin eingesetzt werden.

MPS stellt unter den DSL-Entwicklungsumgebungen eine Besonderheit zur Verfügung: Neue Sprachen können bestehende erweitern. In diesem Sinne kann eine neue Sprache alle Sprachelemente einer anderen erben. Sie kann bereits existierende Elemente übernehmen, zusätzlich aber auch anpassen und ergänzen. Dadurch wird es möglich, sehr effizient (kunden-)spezifische Anpassungen vorzunehmen. Eine DSL-Variante kann

```
TableForm RechungenListeTF (bound type: list<Rechnung> loaded with selected: <class>.<property>)
  label: "Rechnungen"
  select first: true
  <advanced selections>

  on trigger run Rechnungsverwaltung.Rechnungskopf bearbeiten(getSelected(Rechnung.class))
    view for page: Standard = RechWizzKopfFC
  hk: ENTER


  typ heading: "Typ" format: - width: 100 ;
  intBelegNummer heading: "Barcode" format: 0 width: 90 ;
  extBelegNummer heading: "ext. Beleg-Nr." format: - width: 90 ;
  refExtBelegNummer heading: "Bez. Beleg-Nr." format: - width: 90 ;
  bestellNummer heading: "Bestell-Nr." format: 0 width: 90 ;
  rechnungsDatum.<?> heading: "Rech.-Datum" format: dd.MM.yy width: 100 ;
  eingangsDatum.<?> heading: "Eingangsdatum" format: dd.MM.yy width: 100 ;
  freigabeDatum.<?> heading: "Frei.-Datum" format: dd.MM.yy width: 100 ;
  faelligKeitsDatum.<?> heading: "Fällig.-Datum" format: dd.MM.yy width: 100 ;
  bruttoWertSoll.<?> heading: "Brutto soll" format: #,##0.00 width: 79 ;
  diffNettoWert.<?> heading: "Diff." format: #,##0.00 width: 79 ;
  status heading: "Status" format: - width: 60 ;
  printStatus heading: "Druck" format: - width: 79 ;
  id heading: "Id" format: 0 width: 2 ;
```

Abb. 5: Ein Tabellenformular, das eine Liste von Rechnungen anzeigt

```
FormContainer RechWizzKopfFC (bound type: Rechnung loaded with selected: <class>.<property>)
 cols(|): ["1*"] rows(-) ["-1"]
 label: <no heading>
 <no command trigger>

 DelegateForm RechWizzKopfDF (bound type: Rechnung loaded with selected: Rechnung.<property>)
   col(|): ["1*"]
   label: <no lable>
   ReferenceDelegate refLieferant setLabel("Lieferant"), setProperty(lieferant.<?>) ;
   StringDelegate inpExtBelegNummer setLabel("Ext. Belegnummer (Lieferant)"), setProperty(extBelegNummer) ;
   StringDelegate inpExtBelegNummerBez setLabel("Bezug Ext. Belegnummer"), setProperty(refExtBelegNummer) ;
   IntegerDelegate inpBestellNummer setLabel("Bestellnummer"), setProperty(bestellNummer), setMinimum(0) ;
   IntegerDelegate inpBelegNummer setLabel("Barcode"), setProperty(intBelegNummer), setMinimum(0) ;
   DateTimeDateDelegate inpAusstellDatum setLabel("Rechnungsdatum"), setProperty(rechnungsDatum.<?>) ;
   DateTimeDateDelegate inpEingangsDatum setLabel("Eingangsdatum"), setProperty(eingangsDatum.<?>) ;
   StatusDelegate inpRechnungsHerkunft setLabel("Rechnungsherkunft"), setProperty(herkunft) ;
   DateTimeDateDelegate inpFreigabedatum setLabel("Freigabedatum"), setProperty(freigabeDatum.<?>) ;
   DateTimeDateDelegate inpFaelligkeitsdatum setLabel("Fälligkeitsdatum"), setProperty(faelligKeitsDatum.<?>) ;
   StatusDelegate inpArt setLabel("Art"), setProperty(typ), setSelectionList(typ) ;
   StatusDelegate inpStatus setLabel("Rechnungsstatus"), setProperty(status), setEnabled(false) ;
   StatusDelegate inpPrintStatus setLabel("Printstatus"), setProperty(printStatus), setSelectionList(printStatus) ;
   StringDelegate inpBemerkung setLabel("Bemerkung"), setProperty(bemerkung), setNumOfLines(3) ;

   <no onLoad>
   <no onStore>
   onValidate()->void {
     int barcode = inpBelegNummer.getValue();
     boolean barcodeValid = barcode == 0;
     barcodeValid |= barcode > 10000000 && barcode < 99999999;
     if (!(barcodeValid)) {
       inpBelegNummer.setValidationErrorText("Barcode muss 8 stellig sein");
     }

   }
```

Abb. 6: Eingabeformular „RechWizzKopfFC“ zum Bearbeiten einer Rechnung

damit eine bestimmte Zielgruppe oder bestimmte Anwendungsfelder noch spezifischer abdecken.

Die bisher gesammelten Erfahrungen

Wir haben die Ideen der Abstraktion und Strukturierung aufgegriffen und durch eine domänenspezifische Sprache umgesetzt. Die DSL stellt dabei Sprachelemente zur Verfügung, die für einen bestimmten Aufgabenbereich zugeschnitten sind – in diesem Fall für Geschäftsanwendungen. Domänenspezifische Sprachen erlauben uns eine Trennung der fachlichen Aspekte von Implementierungsdetails, die in Codegeneratoren verschoben werden. Die gesamte Implementierung einer Geschäftsanwendung erfolgt modellgetrieben.

Neben der in diesem Artikel angesprochenen Rechnungskontrollsoftware wurden noch zwei weitere Anwendungen vom Kunden erstellt, die umfangreiche Geschäftsprozesse unterstützen. Die Applikationen wurden gemeinsam von einem Team von Anwendungsentwicklern vom Kunden selbst erstellt. Es lassen sich nach zwei Jahren folgende Erfahrungen berichten:

Verständnis/Silodenken: Mit Sprachelementen wie Kommando, Prozess und Formular wurde eine einheitliche Sprache eingeführt, die auch zur Kommunikation unter Beteiligten verwendet wird. In den Modellen finden sich ausschließlich fachliche Aspekte. Modelle werden dadurch relevanter, klarer gegliedert und so verständlicher. Datenstrukturen können anschaulich von Beteiligten diskutiert werden, was ebenfalls dem Silodenken entgegenwirkt. Durch die Sprachelemente ist konsistent geregelt, welche Aufgaben wo abgebildet wurden. Die Gestaltung von Oberflächen wurde mit Formularen festgelegt, Geschäftslogik findet sich ausschließlich in Kommandos, alle Datenbankzugriffe und der Aufbau der Datenstrukturen müssen in Repositories durchgeführt werden.

Die Verwendung von domänenspezifischen Sprachen führt zu einer starken Trennung von fachlichen Aspekten und deren technischer Implementierung. Dadurch wird eine Teilung von Zuständigkeiten erreicht, unterschiedliche Anforderungsprofile an Beteiligte sind die Folge. Während die Erstellung von Codegeneratoren Softwarespezialisten erfordert, werden Anwendungsentwickler entlastet. Sie benötigen weder Wissen über Bibliotheken noch Wissen über den technischen Aufbau der Anwendung. Stattdessen wird das Analysieren von relevanten Geschäftsabläufen und deren Modellierung zentrale Anforderung an Anwendungsentwickler. Er wird Fachexperte im Prozess, nicht in der zugrunde liegenden Technologie. Der Dialog mit Fachabteilungen und Kommunikationsstärke rücken in den Vordergrund.

Anforderungen: Die von uns vorgeschlagene DSL für Geschäftsanwendungen ermöglicht eine systematische Vorgehensweise zur Anforderungserfassung. Meist ist zu Beginn kein vollständiges Bild aller Anforderungen bekannt. Erst durch intensive Auseinandersetzung mit der Problemstellung entsteht in weiterer Folge dann dieses Bild. Die Auseinandersetzung unterstützen wir mit der DSL durch eine bewusst geforderte iterative Vorgehensweise bei der Entwicklung. Fachliche Aspekte können einfach formuliert und schnell verändert werden. Der Programmcode wird schließlich automatisch generiert, wodurch sich die Entwicklungszyklen drastisch verkürzen. Prototypen – genauer gesagt: erste Testszenarien – lassen sich im Diskurs mit Beteiligten auf hoher Abstraktionsebene formulieren. Details zu den Kommandos und deren Implementierung können zu einem späteren Zeitpunkt ausgearbeitet werden. Von Beginn an wird die Geschäftsanwendung grob skizziert und ihre Funktionalitäten erfasst, sodass sich auch ein Grad der Anforderungserfüllung ermitteln lässt. Ist ein Bereich der Anforderungen durch den Prototyp erfolg-

reich abgedeckt, wird der Prototyp sukzessive ergänzt. Während beim Arbeiten mit Dokumenten Anforderungen häufig nur grob skizziert und gleichzeitig tief greifendere Überlegungen aufgeschoben werden (Stichwort „Lastenheft"), fördern Modelle geradezu eine intensive Auseinandersetzung. Ein hoher Grad an Reflexion ist von Beginn an sichergestellt. Zusätzliche Sprachkonzepte für ein Anforderungsmanagement könnten problemlos integriert werden.

Systemdesign: Zentral für die Entwicklung von Geschäftsanwendungen ist für uns das Sprachelement Kommando. Geschäftslogik wird in Kommandos abgebildet und mithilfe von ausdrucksstarken Mengenoperationen formuliert, die durch Java-Code ergänzt werden können. Benutzeroberflächen mit Tabellen und Eingabeformularen lassen sich bei Geschäftsanwendungen besonders effektiv modellieren und sind dadurch gleichzeitig über Anwendungen hinweg konsistent. Das Systemdesign findet sich ausschließlich in den Sprachelementen und in den Codegeneratoren wieder. Damit ist es über Projekte hinweg vereinheitlicht. So ist z. B. klar geregelt, welche Informationen neben der eigentlichen Logik bei einem Kommando hinterlegt werden müssen.

Dokumentation/Wartbarkeit: Fachliche Information muss nicht ständig repliziert, umgeformt und verteilt werden. Modelle und Sprachelemente werden zur gemeinsamen Basis für alle Beteiligten. Die Anforderungen (z. B. auch Datenstrukturen) werden nicht erst in einem Dokument erfasst (Lastenheft) und zu einem späteren Zeitpunkt von Entwicklern in Programmiercode übersetzt. Alle relevanten Informationen werden direkt im Modell erfasst. Fachliche Dokumentation zu Datenstrukturen, Kommandos und zu Prozessen erfassen wir beispielsweise direkt bei den Sprachelementen. Die Dokumentation zeigen wir dann gleichzeitig in der Geschäftsanwendung selbst an, was doppelten Nutzen stiftet. Fachliche Aspekte, die für den Anwendungsentwickler bei seinen Überlegungen wesentlich waren, sind meist eben auch für die Key-User später wichtig.

Zudem ist auch die Namensgebung von Sprachelementen in der Entwicklungsumgebung und der generierter Anwendung transparent, sodass z. B. ein Kommando in der Geschäftsanwendung über dessen Namen in der Entwicklungsumgebung gesucht werden kann. Die einfache Nachverfolgung, vor allem aber die Konzentration auf fachliche Aspekte, vereinfacht auch die Wartung. Während die technische Umsetzung an einer Stelle – nämlich im Codegenerator – gewartet wird, können die fachlichen Aspekte in Modellen überarbeitet werden. Definierte Sprachelemente und übersichtliche Ansichten (z. B. Tabellen und Projektansichten mit Kommandos in der Entwicklungsumgebung) garantieren dabei eine einheitliche Strukturierung von Projekten.

Erfahrungen und vorhandenes Wissen können explizit in DSLs abgebildet werden. Ein Anwendungsentwickler hat uns kürzlich darauf hingewiesen, die in den Oberflächen sichtbaren Bezeichnungen (z. B. Spaltenüberschriften etc.) nicht in den Formularen, sondern direkt in der Datenstruktur zu spezifizieren; eine gute Idee. Die Konsistenz bei Bezeichnungen wird erhöht, die Anwender werden es danken. Zudem wird die einheitliche Sprachregelung dadurch wieder gefestigt.

Testen: Die Wartbarkeit von Anwendungen wird auch maßgeblich von der Testinfrastruktur bestimmt. Anwendungen systematisch zu testen, ist von besonderer Bedeutung. Denn nur mit systematischen Tests können Aussagen zur Qualität einer Anwendung getroffen und diese Aussagen dann langfristig geprüft werden. Barrieren, die das systematische Testen in der Praxis verhindern, müssen aufgelöst werden. Als Konsequenz muss das Testen durch die DSL aktiv – vom ersten Prototyp und ohne zusätzlichen Aufwand – unterstützt werden.

Kommandos lassen sich bei unserer DSL mit so genannten Mocks (Fake-User-Interfaces) ausführen, passende Sprachelemente sind vorhanden. Kommandos können dadurch zusammen mit häufigen Interaktionsmustern einzeln getestet werden. Es lassen sich aber auch gesamte Abläufe nachstellen (Anforderungstests). Die verschiedenen Testdaten werden bereits bei der Modellierung der Datenstruktur erfasst, d. h. die Modellierung der Datenstruktur wird zu einem Wechselspiel zwischen dem Erfassen von plausiblen Testdaten und den dazu notwendigen Anpassungen an der Datenstruktur selbst. Die Repositories greifen beim Testen nicht auf die Datenbank, sondern auf diese Testdaten zurück. Mit einem Knopfdruck – ohne Änderungen an der Geschäftsanwendung – lassen sich alle Testfälle vor der Auslieferung durchspielen.

Unabhängigkeit: Die Trennung der Implementierung durch Codegeneratoren schafft langfristige Unabhängigkeit zu technischen Frameworks und Plattformen. IT-Technologien entwickeln sich mit rasanter Geschwindigkeit. Die Potenziale zukünftiger Technologien einfach in Eigenentwicklungen nutzen zu können, das kann bei den noch nicht absehbaren technischen Entwicklungen entscheidend sein. Die von uns vorgestellte Anwendung zur Rechnungskontrolle kann ohne Änderungen am fachlichen Modell problemlos in die Eclipse-Plattform, in eine Webanwendung oder in eine native Tablet-App migriert werden.

Daniel Stieger ist Partner der modellwerkstatt (www.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, einem auf Eclipse und JavaFX spezialisierten Softwarehauses in Österreich. Er ist e4-Committer und Project Lead des e(fx)clipse-Projekts.

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

2/1 MTC Spring

2/1 MTC Spring