Datenbearbeitung mit Modellen

Konfigurierbare Formulare in Rich-Client-Anwendungen

Tim Wellhausen

Lilienstraße 20
81669 München
Tel. 0178 / 846 93 55
kontakt@tim-wellhausen.de
http://www.tim-wellhausen.de

Version vom 05.05.2005

Zusammenfassung: Zu den Einsatzgebieten der modellbasierten Entwicklung gehören grafische Bedienoberflächen. Für eine Modellierung innerhalb dieses Gebiets eignen sich insbesondere Formulare, mit welchen Geschäftsdaten angezeigt und bearbeitet werden. Solche Formulare bestehen aus wenigen unterschiedlichen Elementen, die in Variationen zusammengefügt werden. Viele modellbasierte Ansätze konzentrieren sich nur auf die Struktur solcher Formulare; dynamische Aspekte, die über eine einfache Ereignisverarbeitung hinausgehen, werden nur selten unterstützt. Daher stellt dieser Artikel ein Konzept vor, das neben der Gestaltung von Formularen auch deren dynamische Eigenschaften umfasst.

Einführung

Datenbankbasierte Geschäftsanwendungen enthalten zahlreiche Formulare, die Geschäftsdaten anzeigen und es den Anwendern ermöglichen, diese zu manipulieren. Die Formulare erfüllen über die reine Datenbearbeitung hinaus häufig zusätzliche Anforderungen:

- Die Rechte der Anwender zum Zugriff auf die Geschäftsdaten legen fest, ob Felder lese- bzw. schreibgeschützt sind.
- Die Formulare reagieren auf die Eingaben, je nach Anwendungslogik, dynamisch. Beispielsweise werden manche Felder unter bestimmten Bedingungen individuell gesperrt.
- Um den Anwendern Fehler bei der Eingabe sofort anzuzeigen, werden die Eingaben schon im Formular überprüft.
- Die Formulare enthalten nicht nur Daten aus der Datenbank, sondern auch davon abgeleitete Informationen, deren Darstellung bei jeder Datenänderung sofort aktualisiert wird.
- Alle Formulare sehen einheitlich aus und lassen sich gleichartig bedienen. Eine GUI-Richtlinie schreibt Details vor, beispielsweise Schriftgrößen und Abstände zwischen grafischen Komponenten.

Wenn Formulare manuell entwickelt werden, ist es schwierig, sie konsistent und fehlerfrei zu halten. Werkzeuge, wie zum Beispiel *GUI-Builder*, können die Entwicklung unterstützen; ihre Wirksamkeit beschränkt sich jedoch auf die Gestaltung der Formulare, nicht auf die obigen Anforderungen.

Mit einem modellbasierten Ansatz werden die grundsätzlichen Anforderungen zentral an einer Stelle umgesetzt. Eine domänenspezifische Sprache auf Basis der *eXtended Markup Language (XML)* spezifiziert die konkreten Ausprägungen der Formulare. Dadurch wird sichergestellt, dass sich das Aussehen und das Verhalten aller Formulare gleicht.

Für einen modellbasierten Ansatz gibt es zwei Varianten. Bei einem *generativen Ansatz* werden die Spezifikationen der Formulare zur Kompilierzeit in ausführbaren Programmcode transformiert. Bei einem *interpretationsbasierten Ansatz* wertet ein generisch arbeitender Kern die Spezifikationen zur Laufzeit aus und instanziiert die Formular-Instanzen entsprechend der Vorgaben. Das in diesem Artikel vorgestellte Konzept folgt dem interpretationsbasierten Ansatz, da er flexibler ist; dynamische Aspekte lassen sich einfacher umsetzen. Dies entspricht auch dem Vorgehen bei vergleichbaren Projekten (vgl. [RuSc05], [SML], [XUL], [XAML]).

Das Konzept entstand im Rahmen eines Industrieprojekts, bei dem eine Plattform für eine Familie von Geschäftsanwendungen entwickelt wurde. Auf dieser Plattform wurden sowohl vorhandene Anwendungen portiert als auch neue Anwendungen erstellt. Wichtig war, dass sich die Formulare ohne tief greifende Kenntnisse in der GUI-Programmierung entwickeln lassen.

Überblick

Der Kern des Systems besteht im Wesentlichen aus zwei generisch arbeitenden Komponenten: einer *Ansicht* und einem *Datenmodell*. Die Ansicht-Komponente erstellt die Formulare, die Datenmodell-Komponente verwaltet deren Daten (s. Abb. 1). Zur Laufzeit wird für jedes Formular je eine Instanz beider Komponenten benötigt: Eine Datenmodell-Instanz enthält alle Daten und Zustandsinformationen, eine Ansicht-Instanz alle Eingabekomponenten und sonstige Bestandteile des Formulars. Wenn im Folgenden von „Datenmodell“ und „Ansicht“ die Rede ist, sind Instanzen dieser Komponenten gemeint.

Eine generische Steuerungskomponente wird nicht benötigt. Vielmehr sind die Eingabekomponenten so an die Elemente des Datenmodells gekoppelt, dass die Komponenten durch Änderungen im Datenmodell gesteuert werden. Die Aufgabe der Entwickler besteht darin, für jedes Formular Instanzen beider Komponenten durch Programmcode zu initialisieren und zu verbinden. Zusätzlich kann für Sonderfälle, die nicht in der Beschreibungssprache modellierbar sind, spezieller Steuerungscode geschrieben werden.

Das *Datenmodell* besteht aus sämtlichen Daten, die für ein Formular von Bedeutung sind, wobei alle Daten lokal innerhalb der Client-Anwendung vorliegen. Es enthält sowohl Geschäftsdaten als auch die Zustandsinformationen der Eingabekomponenten (Sichtbarkeit, Aktivierung, Pflichtfeld, Selektionen). Eine einheitliche Zugriffsschicht abstrahiert vom Typ und von der Herkunft der Daten. Ein Datenmodell ist jedoch nicht nur ein passiver Datenspeicher, sondern enthält darüber hinausgehende Funktionalität:

- Es lädt und speichert Geschäftsdaten, beachtet die Zugriffsberechtigungen der Anwender und führt einfache Geschäftslogik aus. Beispielsweise überprüft es die Eingaben und berechnet abgeleitete Informationen.
- Es wertet Regeln für Abhängigkeiten zwischen den Eingabekomponenten eines Formulars aus. Eine Abhängigkeit liegt dann vor, wenn eine Komponente ihren Wert oder ihren Zustand ändert, sobald der Wert oder der Zustand einer anderen Komponente sich geändert hat. Eine solche Abhängigkeit besteht etwa, wenn ein Textfeld genau dann aktiviert sein soll, wenn ein Kontrollkästchen selektiert ist.

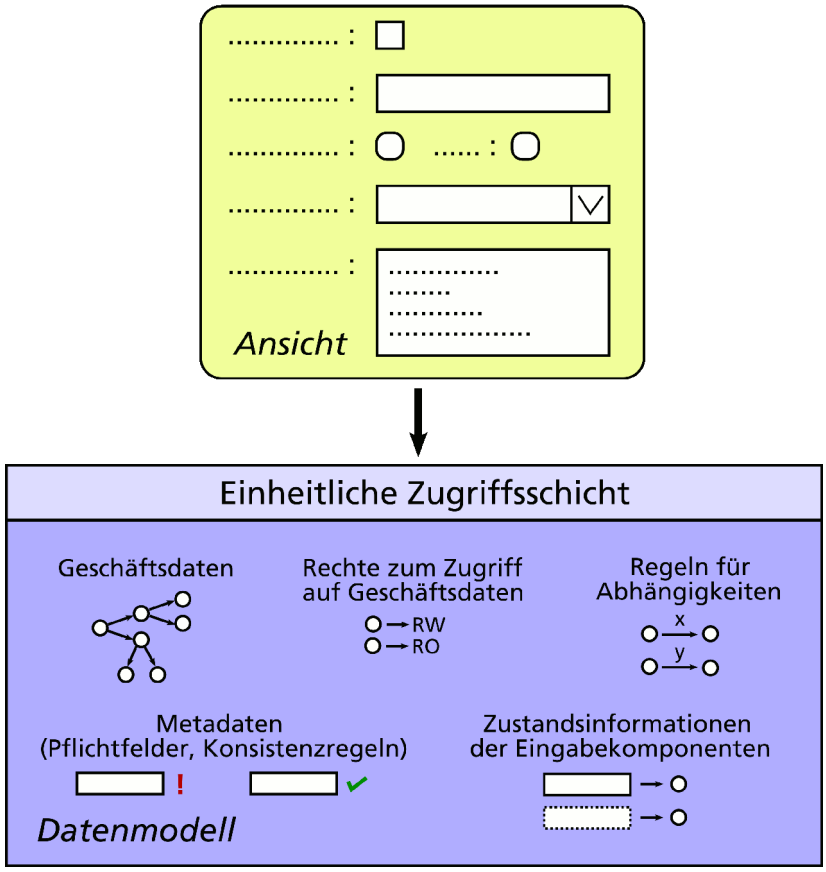

Abb. 1: Ansicht und Datenmodell

- Es ermöglicht die Überwachung der Daten. Sobald sich Daten ändern, werden die registrierten Ereignisempfänger benachrichtigt (*Publisher-Subscriber* Muster). Dieser Mechanismus bezieht sich sowohl auf einfache Werte (z. B. Zahlen oder Zeichenketten) als auch auf Datenstrukturen (z. B. Listen).

Die Definition eines Datenmodells in einer XML-Beschreibungssprache enthält die Art und die Struktur der Geschäftsdaten, Konsistenzregeln und Abhängigkeiten. Manche Informationen werden erst zur Laufzeit dynamisch hinzugefügt, z. B. die Berechtigungsinformationen der jeweils angemeldeten Anwender.

Die Zugriffsschicht eines Datenmodells bietet über eine einheitliche Schnittstelle Zugang zu den enthaltenen Daten. Sie wird deklarativ genutzt, sodass die Struktur der Daten verborgen bleibt. Dazu ein Beispiel: Ein Geschäftsobjekt der Klasse „`Kunde`“ enthält das Attribut „`nachname`“. Folgender Aufruf liefert den Namen des Kunden-Objekts: `dataModel.getValueHolder ("Kunde.nachname").toString()`. Der deklarative Ansatz ermöglicht beliebig verschachtelte und kombinierte Datenmodelle. Details hierzu folgen weiter unten.

Eine *Ansicht* ist dafür zuständig, alle Eingabekomponenten zu initialisieren und zu einem Formular zusammenzufügen. Die Eingabekomponenten stellen Daten dar und weisen Zustände auf. Manche Komponenten zeigen einfache Werte an (z. B. Textfeld, Kontrollkästchen), andere Datenstrukturen (z. B. Liste, Tabelle). Zustandsinformationen bestimmen beispielsweise, ob eine Komponente schreibgeschützt ist oder ob sie als Pflichtfeld behandelt wird.

Die Ansicht ist auch dafür verantwortlich, die Vorgaben einer GUI-Richtlinie über Größen, Abstände und Ausrichtungen umzusetzen. Während die Ansicht das Formular aufbaut, wird

jede Eingabekomponente anhand dieser Vorgaben einheitlich initialisiert und dem Formular hinzugefügt.

Die Eingabekomponenten lesen die anzuzeigenden Werte und ihre Zustandsinformationen aus dem Datenmodell aus und schreiben sie bei jeder Änderung dorthin zurück (s. Abb. 2). Sie registrieren sich beim Datenmodell, sodass sie darüber informiert werden, wenn sich dort Daten ändern. Wenn sich beispielsweise die Selektion eines Kontrollkästchens ändert und eine passende Regel definiert wurde, dann verändert das Datenmodell den Wert, der angibt, ob das Textfeld aktiv ist. Diese Änderung wird vom Textfeld wahrgenommen, sodass es daraufhin seinen Aktivierungszustand anpasst.

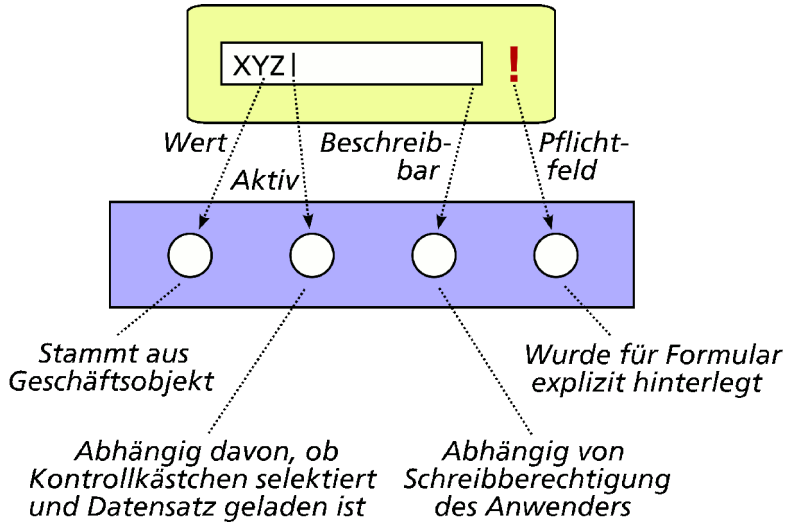

Abb. 2: Verbindung zwischen Eingabekomponente und Datenmodell

Eine eigene Beschreibungssprache definiert, wie eine Ansicht aus grafischen Komponenten zusammengesetzt ist. In dieser Sprache wird angegeben, mit welchen Daten aus dem Datenmodell die Eingabekomponenten verbunden werden, also welche Informationen sie anzeigen und wo im Datenmodell ihre Zustandsinformationen abgelegt sind.

Die Trennung eines Formulars in seine Ansicht und sein Datenmodell ähnelt dem *Model-View-Controller*-Muster (MVC) (vgl. [POSA96]). Im Unterschied zu MVC enthält das hier vorgestellte Datenmodell durch die Definition der dynamischen Eigenschaften auch die Steuerung. Eine Ansicht besteht aus vielen Eingabekomponenten, die wiederum nach dem MVC-Muster aufgebaut sein können. Deren Modelle sind mit dem umfassenden Datenmodell verbunden.

Der Vorteil der Trennung auf dieser Abstraktionsebene liegt darin, dass sie eine ganzheitliche Sicht auf alle Daten eines Formulars einschließlich ihrer Beziehungen untereinander schafft. Die Ansicht ist von den Daten und deren Verarbeitung entkoppelt, wird jedoch bei Änderungen am Datenmodell automatisch aktualisiert. Da auch die Zustandsinformationen der Eingabekomponenten im Datenmodell abgelegt sind, kann ein Formular durch Manipulationen im Datenmodell vollständig gesteuert werden.

Die Umsetzung

Im Folgenden werden Details der Implementierung des vorgestellten Konzepts erläutert. Die beschriebene Umsetzung wird durch ein Beispiel illustriert, das auf einem sehr vereinfachten Geschäftsobjektmodell basiert: einer Verwaltung von Kunden mit Adressdaten und Verträgen (s. Abb. 3).

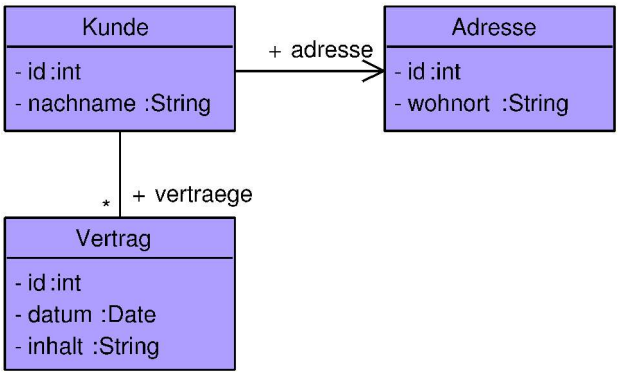

Abb. 3: Klassenstruktur der Beispiel-Kundenverwaltung

Anhand des Beispiels werden im Folgenden die Schritte vorgeführt, die man durchführen muss, um ein Formular für die Pflege der Kundendaten zu erstellen. Zunächst wird ein Datenmodell definiert, das die Geschäftsobjekte aufnimmt. Anschließend wird die Zuordnung der Daten zu den Eingabekomponenten beschrieben, danach das Aussehen des Formulars. Zum Schluss wird gezeigt, wie eine Instanz des Formulars erstellt und angezeigt wird.

Datenmodell

Der hier vorgestellten Umsetzung steht eine Technologie zur Verfügung, mit der eine Client-Anwendung Geschäftsdaten über eine generische Schnittstelle eines Applikationsservers laden und speichern kann. Geschäftsdaten können in Form von typisierten Geschäftsobjekten oder untypisiert durch generische Datentransferobjekte geladen werden. Mit Hilfe der generischen, untypisierten Funktionalität kann ein Datenmodell Geschäftsdaten selbstständig laden und speichern, ohne an die Klassen des fachlichen Geschäftsobjektmodells gebunden zu sein.

Da die Verantwortung dafür, wann Geschäftsdaten geladen bzw. nachgeladen werden, auf der Client-Seite liegt, wurden verschiedene Strategien zur Datenübertragung implementiert, um auch bei großen Mengen von Geschäftsdaten eine hohe Ausführungsgeschwindigkeit zu erhalten:

- Die Zugriffsschicht eines Datenmodells lädt Geschäftsdaten wahlweise sofort oder verzögert im Hintergrund nach. Die Ansicht stellt die Daten konsistent dar.
- Falls die benötigten Geschäftsdaten über mehrere Geschäftsobjekte verteilt sind, lädt die Zugriffsschicht die Daten assoziierter Geschäftsobjekte entweder sofort oder erst bei Bedarf.
- Geschäftsdaten, die sich über mehrere Tabellen erstrecken, werden entweder einzeln oder durch einen Datenbank-Join transparent auf einmal geladen.

Die geeignete Strategie beziehungsweise eine Kombination dieser Strategien trägt man in die Konfigurationsdaten ein, oder das System bestimmt sie zur Laufzeit anhand von Heuristiken.

Ein Datenmodell enthält neben den Geschäftsobjekten mit ihren Attributen und Assoziationen auch andere Arten von Daten. Dazu gehören:

- Zugriffsrechte auf Geschäftsobjekte,
- Metadaten, z. B. Konsistenzregeln und Pflichtfeldinformationen,
- nicht-persistente Daten, die nur zur Laufzeit einer Anwendung existieren, und
- Eigenschaften der Eingabekomponenten, z. B. Aktivierung und Selektion

Die Zugriffsschicht der Datenmodell-Komponente stellt eine Schnittstelle mit der Bezeichnung `DataModelNode` zur Verfügung, die einen Knoten innerhalb des Datenmodells repräsentiert (s. Abb. 4). Jeder Knoten hat einen Wert und kann auf weitere Knoten verweisen. Ein Knoten kann eine Liste von Geschäftsobjekten, ein einzelnes Geschäftsobjekt, ein Attribut eines Geschäftsobjekts oder eine Zustandsinformation einer Eingabekomponente repräsentieren. Die Herkunft und Art der Daten bleiben hinter der Schnittstelle verborgen.

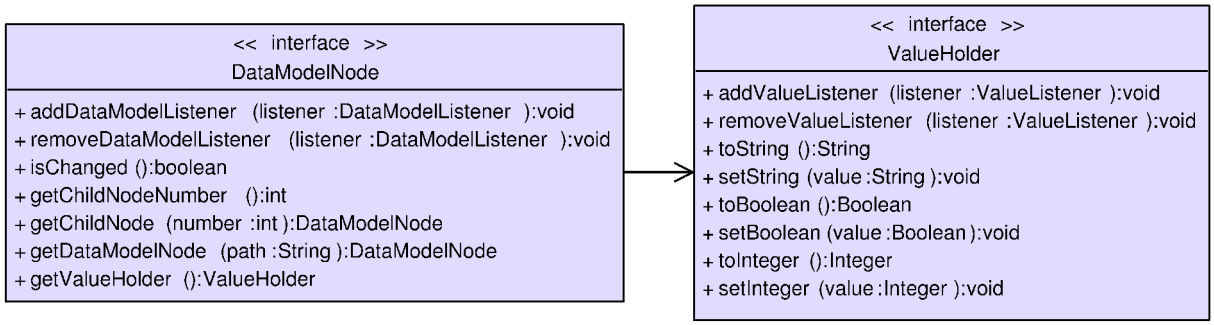

Abb. 4: Die Schnittstellen der Datenmodell-Zugriffsschicht

Jeder Knoten lässt sich durch einen Ausdruck eindeutig referenzieren. Mit der Methode `getDataModelNode(String)` kann man deklarativ von einem Knoten zu einem anderen navigieren. Diese Methode erwartet einen Pfad, der aus den Namen von aufeinander folgenden Knoten zusammengesetzt ist, und liefert dem Aufrufer den Knoten zurück, der durch den Pfad referenziert wird. Diese Art der Referenzierung ähnelt der, die in XPath ([XPath]) gebräuchlich ist. Dazu ein paar Beispiele:

```
getDataModelNode("Kunde.id");
getDataModelNode("Kunde.nachname.schreibrecht");
getDataModelNode("Kunde.adresse.wohnort");
getDataModelNode("Kunde.vertraege[2].datum");
```

Die Struktur eines Datenmodells wird durch eine XML-basierte Beschreibungssprache definiert. Das folgende Beispiel zeigt einen vereinfachten Ausschnitt aus der Konfiguration für die Kundenverwaltung: Definiert wird die Struktur der Geschäftsobjekte.

```
<datamodel>
  <column>
    <businessobject class="Kunde">
      <field name="id" type="Integer"/>
      <field name="nachname" type="String"/>
      <association type="toone">
        <field name="adresse"/>
        <businessobject class="Adresse"/>
      </association>
      <associaton type="tomany">
        <field name="vertraege"/>
        <businessobject class="Vertrag"/>
      </association>
    </businessobject>
  </column>
</datamodel>
```

In der konkreten Umsetzung des beschriebenen Konzepts wurde ein objekt-relationales Mapping-Tool mit einer eigenen Beschreibungssprache für die Abbildung zwischen Geschäftsklassen und Datenbanktabellen verwendet. Um Redundanzen zu vermeiden, wurden die dort vorhandenen Informationen über die Geschäftsklassen zur Laufzeit in die Beschreibungssprache eines Datenmodells transferiert und dort dynamisch eingebunden.

Nicht nur der Zugriff auf die Datenstruktur ist transparent hinter einer Schnittstelle verborgen, sondern auch der Zugriff auf die eigentlichen Werte, also Instanzen von Datentypen wie Zahlen, Zeichenketten oder Datumsangaben. Im Datenmodell ist jeder Knoten mit einem Wert verknüpft; die Schnittstelle `DataModelNode` bietet Zugriff auf den dazugehörigen Wert eines Knotens. Auch Werte werden durch eine Schnittstelle gekapselt. Der Name dieser Schnittstelle lautet `ValueHolder`; ein Ausschnitt findet sich ebenfalls in Abb. 4.

Für jeden Datentyp existiert eine Implementierung der Schnittstelle `ValueHolder`, beispielsweise `StringHolder` für Zeichenketten oder `IntegerHolder` für Zahlen. Die Schnittstelle `ValueHolder` ermöglicht den Zugriff auf die Werte über generische Funktionen (`setString`, `setInteger` etc.). Daher finden in den Klassen gegebenenfalls Typkonvertierungen statt. Ein `ValueHolder`-Objekt ist somit ein Proxy für ein dahinter liegendes Datenobjekt.

Neben der Kapselung von Datenobjekten eignet sich die `ValueHolder`-Schnittstelle auch für dynamische Aspekte. Ein Datenmodell kann spezialisierte Implementierungen dieser Schnittstelle aufnehmen. Beispielsweise lassen sich Berechnungen für abhängige Werte in ein Datenmodell integrieren. Das spezialisierte `ValueHolder`-Objekt greift auf andere Werte im Datenmodell zu und berechnet auf deren Grundlage einen eigenen Wert.

Für häufig wiederkehrende Funktionen enthält der Kern spezialisierte Implementierungen, etwa für Boolesche Operationen. Damit lässt sich unter anderem realisieren, dass der Wert für den Aktivierungszustand eines Textfelds genau dann wahr ist, wenn sowohl der Wert für die Selektion eines Kontrollkästchens als auch der Wert für das benutzerspezifische Schreibrecht auf das Datenfeld wahr ist.

Abb. 5 zeigt einen Ausschnitt der Objektstruktur eines Datenmodells zur Laufzeit. Der Ausschnitt enthält zwei Geschäftsobjekte – einen Kunden und seine Adresse – und zeigt, wie die Zugriffsschicht des Datenmodells die Struktur der Geschäftsobjekte und ihrer Attribute nachbildet. Für jedes Geschäftsobjekt und jedes Attribut gibt es eine `DataModelNode`-Instanz, für jeden Wert eine `ValueHolder`-Instanz.

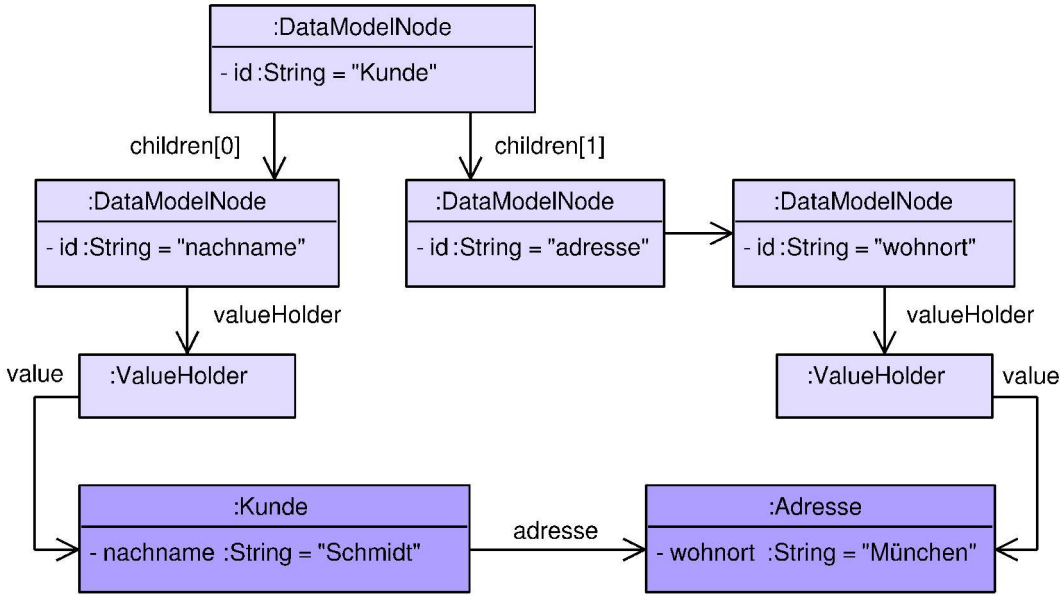

Abb. 5: Objektstruktur eines Datenmodells am Beispiel eines Kundenobjekts

In diesem Beispiel enthält das Datenmodell verdoppelte Strukturen: Zu den eigentlichen Geschäftsobjekten kommen die Objekte der Zugriffsschicht. Dadurch steigt der Speicherverbrauch

an. Aufgrund des vollständig generischen Ansatzes der Zugriffsschicht lassen sich dafür beliebige Daten aus unterschiedlichen Quellen transparent ansprechen, also auch solche, die nicht persistent sind, sondern nur zur Laufzeit benötigt werden (z. B. die Zustandsinformationen der Eingabekomponenten).

Eingabekomponenten

Formulare bestehen überwiegend aus Eingabekomponenten zur Datenbearbeitung. Die Umsetzung des hier vorgestellten Konzepts basiert auf Java Swing, dessen Eingabekomponenten (z. B. `JTextField`) nach dem MVC-Muster aufgebaut sind. Swing sieht für jede Eingabekomponente Schnittstellen vor, die das Modell der Komponente definieren. Diese Schnittstellen enthalten Operationen, mit denen die Ansichten der Komponenten Informationen abfragen und verändern können. Die Swing-Schnittstelle `Document` ist beispielsweise das Modell eines Textfelds; die Swing-Schnittstellen `ListModel` und `ListSelectionModel` sind das Modell einer Liste.

Um die Informationen aus dem Datenmodell in den Eingabekomponenten von Swing anzuzeigen, müssen die Eingabekomponenten mit den Werten, genauer: den `ValueHolder`-Objekten, verbunden werden. Dies lässt sich realisieren, indem für jede Modell-Schnittstelle aus Swing eine eigene Implementierung erstellt wird, die als Adapter zum Datenmodell dient. Abb. 6 verdeutlicht diese Struktur anhand eines Klassendiagramms, in dem ein Textfeld mit einem Wert, in diesem Fall einer Zeichenkette, verbunden ist.

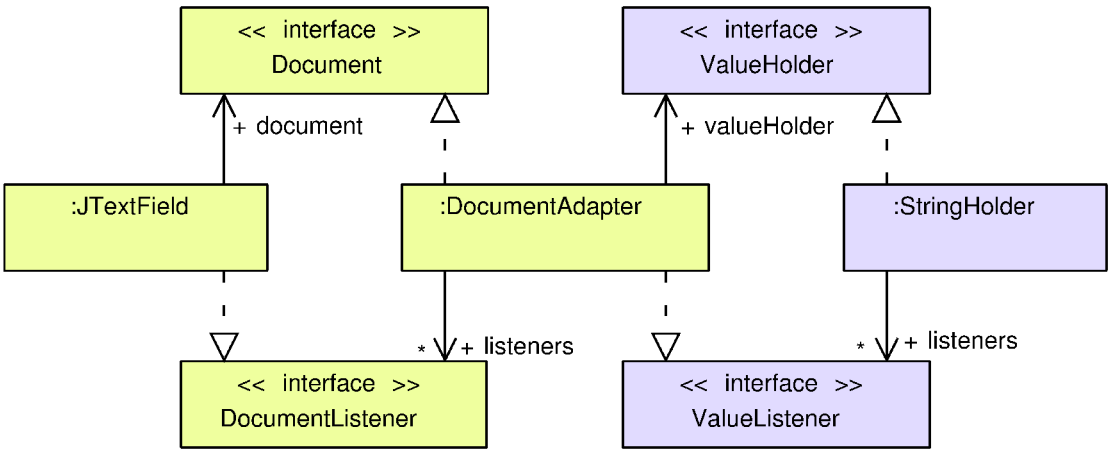

Abb. 6: Verbindung einer JTextField-Instanz mit einem Wert aus dem Datenmodell

Der Adapter (die Klasse `DocumentAdapter`) entkoppelt das Textfeld (`JTextField`) von der Zeichenkette im Datenmodell (`StringHolder`). Der Adapter benachrichtigt das Textfeld, wenn sich der Wert ändert, und schreibt Änderungen in das Datenmodell zurück.

Abb. 7 zeigt in einem Sequenzdiagramm, wie das Textfeld von einer Änderung der Zeichenkette benachrichtigt wird und sich daraufhin den aktuellen Wert besorgt. Das `StringHolder`-Objekt erhält einen neuen Wert und benachrichtigt alle angemeldeten `ValueListener`-Objekte, zu denen auch die `DocumentAdapter`-Instanz gehört. Der Adapter leitet die Benachrichtigung an alle Objekte weiter, die als `DocumentListener` angemeldet sind, also auch an das `JTextField`-Objekt. Damit das Textfeld den angezeigten Text ändern kann, benötigt es den geänderten Wert, wozu es auf sein `Document` zugreift, also auf den Adapter. Der Adapter holt sich den Wert von seinem `ValueHolder`-Objekt und übermittelt ihn zurück an das Textfeld.

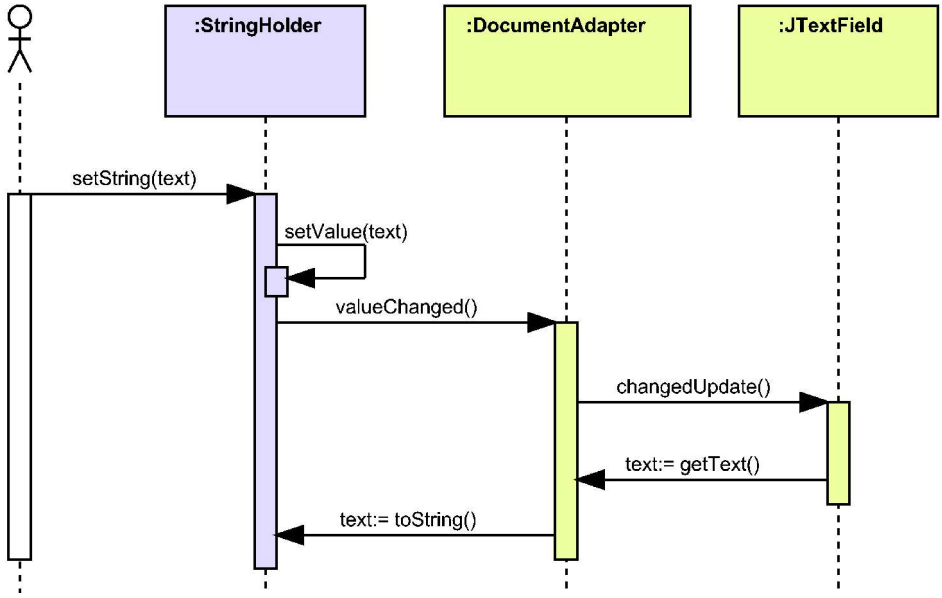

Abb. 7: Änderung eines Wertes im Datenmodell

Swing kennt kein Modell für Zustandsinformationen, sondern behandelt Zustände als Eigenschaften der Eingabekomponenten. Die Zustände lassen sich nur über die Komponenten selbst verändern (z. B. über die Methoden `setEnabled` und `setEditable`). Um Zustandsinformationen dennoch im Datenmodell abzulegen, wurde ein neues Paar an Schnittstellen eingeführt: `StatusModel` und `StatusListener`. Swings Komponenten wurden so erweitert, dass Änderungen im Zustandsmodell mit den internen Eigenschaften der Komponenten synchronisiert werden. Die Klassenstruktur dieser Lösung folgt der Klassenstruktur in Abb. 6: Ein `StatusAdapter` verbindet die erweiterte Swing-Komponente mit dem Datenmodell, ohne beide Akteure direkt zu koppeln.

Ansichten

Die Beschreibungssprache für das Aussehen der Formulare lässt sich am besten durch ein Beispiel erläutern. Der folgende Ausschnitt einer XML-Datei zeigt die Beschreibung eines Formulars mit Kundendaten: In einem tabellenbasierten Layout wird jede Komponente mit einem Element des Datenmodells verbunden.

```
<container type="panel" layout="table">
  <label id="NachnameLabel" text="Kundenname:">
    <layoutproperties x="0" y="0" w="1" h="1"/>
  </label>
  <textfield id="NachnameText" label="NachnameLabel">
    <layoutproperties x="1" y="0" w="1" h="1"/>
    <datamodel component="NachnameTextfeld"/>
  </textfield>
  <label id="WohnortLabel" text="Wohnort:">
    <layoutproperties x="0" y="1" w="1" h="1"/>
  </label>
  <textfield id="WohnortText" label="WohnortLabel">
    <layoutproperties x="1" y="1" w="1" h="1"/>
    <datamodel component="WohnortTextfeld"/>
  </textfield>
  <label id="VertraegeLabel" text="Verträge:">
```

```
    <layoutproperties x="0" y="2" w="1" h="1"/>
  </label>
  <list id="VertraegeList" label="VertraegeLabel">
    <layoutproperties x="1" y="2" w="1" h="1"/>
    <datamodel component="VertraegeListe"/>
  </list>
</container>
```

Die Konfigurationsdaten werden ausgewertet, bevor ein Formular zum ersten Mal aufgebaut wird. Im ersten Schritt werden alle Konfigurationsdaten ausgelesen und intern verarbeitet. Im zweiten Schritt werden die Eingabekomponenten initialisiert, mit dem Datenmodell verbunden und – unter Berücksichtigung der Layout-Eigenschaften – zu einem Formular zusammengesetzt. Dieser zweite Schritt berücksichtigt die Vorgaben einer GUI-Richtlinie, etwa den Abstand zwischen den Eingabekomponenten.

Anstatt die Layout-Eigenschaften der Eingabekomponenten explizit anzugeben, lassen sich tabellenbasierte Formulare vereinfacht beschreiben. Viele Formulare sind regelmäßig aufgebaut, sodass auf jeder Zeile eine Beschriftung und die dazugehörige Eingabekomponente nebeneinander stehen. Diese Regelmäßigkeit lässt sich durch eine kompaktere Beschreibungssprache ausnutzen:

```
<container type="panel" layout="form">
  <textfield id="NachnameText">
    <label text="Kundenname:"/>
    <datamodel component="NachnameTextfeld"/>
  </textfield>
  <textfield id="WohnortText">
    <label text="Wohnort:"/>
    <datamodel component="WohnortTextfeld"/>
  </textfield>
  <list id="VertraegeList">
    <label text="Verträge:"/>
    <datamodel component="VertraegeListe"/>
  </list>
</container>
```

In diesem Format ergeben sich die Koordinaten der Eingabekomponenten aus der Reihenfolge in der Beschreibung. Durch eine vorgeschaltete Transformation wird die kompakte Definition in die Standard-Beschreibungssprache mit expliziten Layout-Eigenschaften umgewandelt.

Im oben angeführten Beispiel wurde auf die Daten der Eingabekomponenten verwiesen. Deren Einträge definieren sich wie folgt:

```
<graphicalcomponent name="NachnameTextfeld">
  <datamodelnode reference="Kunde.nachname"/>
  <status>
    <activation>
      <datamodelnode reference="Kunde.nachname.schreibrecht"/>
    </activation>
  </status>
</graphicalcomponent>


<graphicalcomponent name="VertraegeListe">
  <datamodelnode reference="Kunde.vertraege"/>
</graphicalcomponent>
```

Jede Definition einer Eingabekomponente enthält einen Verweis auf die Daten, die die Komponente anzeigen soll. Der Verweis kann auf einfache Werte zeigen, wie bei der

Komponente „NachnameTextfeld“, oder auf Datenstrukturen, wie bei der Komponente „VertraegeListe“. Zusätzlich können zu jeder Komponente Zustandsinformationen hinterlegt werden.

Viele Einträge zur Definition eines Formulars lassen sich automatisieren. Beispielsweise sollte der Aktivierungsstatus jeder Eingabekomponente mit dem Schreibrecht des Anwenders auf die jeweiligen Daten gekoppelt werden. Dies lässt sich durch ein Tool realisieren, das diese Abhängigkeiten statisch in die Konfigurationsdateien aufnimmt. Alternativ können die fehlenden Einträge auch zur Laufzeit dynamisch hinzugefügt werden, etwa indem die Modellinformationen erst zur Laufzeit ausgewertet werden.

Wenn die Konfigurationsdaten für ein Formular vorliegen, reichen wenige Zeilen Programmcode, um ein Formular aufzubauen und darzustellen. Der folgende Java-Code erzeugt ein Fenster mit dem gewünschten Formular und zeigt es an:

```
DataModel dataModel = new DataModel("KundeModell.xml");
ViewDefinition view = new ViewDefinition("KundeAnsicht.xml", dataModel);

JDialog dialog = view.createDialog();
dialog.show();

dataModel.loadBusinessObject("Kunde", id);
```

In der letzten Zeile wird das Datenmodell angewiesen, ein bestimmtes Kunden-Objekt zu laden. Dies geschieht intern im Datenmodell durch einen Aufruf an den Applikationsserver. Sobald das Kunden-Objekt lokal vorliegt, benachrichtigen die Knoten, die mit Daten des Kunden-Objekts verbunden sind, ihre Ereignisempfänger. Wie weiter oben am Beispiel eines Textfelds beschrieben wurde, aktualisieren die Eingabekomponenten daraufhin die von ihnen angezeigten Daten, sodass nach kurzer Zeit das Formular die vollständigen Daten des Kunden-Objekts darstellt. Abb. 8 zeigt das Formular, das durch das vorgestellte Beispiel konfiguriert und mit Daten gefüllt wird.

Abb. 8: Der Beispiel-Dialog

Fazit

Dieser Artikel hat vorgestellt, wie sich mit einem modellbasierten Ansatz Formulare für Rich-Client-Anwendungen beschreiben lassen. Ein Datenmodell mit einer einheitlichen Abstraktionsschicht enthält nicht nur alle für ein Formular relevanten Daten, sondern steuert auch dessen dynamische Eigenschaften. Eine kompakte Konfigurationssprache für das Aussehen der Formulare sorgt dafür, dass Standardformulare zügig umgesetzt werden können.

Ein solches Konzept erfordert zu einem gewissen Maß einen Paradigmenwechsel. Die Entwickler haben nicht mehr die volle Verantwortung über die Implementierung der Formulare, sondern modellieren sie mittels einer Beschreibungssprache. Als Ergebnis sehen die Formulare einheitlich aus, arbeiten konsistent und lassen sich vergleichsweise leicht testen, da jeder Fehler nur einmal auftritt, nämlich im Kern. Der anfängliche Aufwand, ein solches Konzept umzusetzen, ist zwar nicht unerheblich. Der Aufwand zahlt sich jedoch insbesondere dann aus, wenn eine Familie von Geschäftsanwendungen entwickelt wird.

Literatur & Links

[POSA96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-oriented Software Architecture - A system of patterns, John Wiley and Sons, 1996

[RuSc05] B. Rumpe, J. Schmid, Oberflächen mit MDA: Beschreiben statt programmieren, in: OBJEKTspektrum 2/05

[SML] Swing Markup Language, siehe: http://swingml.sourceforge.net

[XUL] XML User Interface Language, siehe: http://www.mozilla.org/projects/xul

[XAML] Extensible Application Markup Language, siehe: http://msdn.microsoft.com/longhorn

[XPath] XML Path Language, siehe: http://www.w3.org/TR/xpath