Abgekapselt

Domänenmodelle und Interaktion

Phillip Ghadir

Mit Domänenmodellen strukturieren wir fachliche Zusammenhänge: Auf Basis eines fachlichen Kerns werden applikationsspezifische Teile realisiert, die wir dann mit Benutzerschnittstellen versehen. Während der fachliche Kern die Datenkonsistenz sicherstellt, ermöglichen die Benutzerschnittstellen dem Anwender mit dem Kern zu interagieren.

Zu selten spiegeln Domänenmodelle die Interaktionsmetaphern der Benutzungsoberfläche wider, sodass die Anbindung holprig wird. Immer wieder begegne ich Projekten, in denen dann an der Oberfläche gefummelt wird: Von außen betrachtet – nach gewissen Kriterien – schön, aber mit schlechter interner Qualität.

In diesem Beitrag möchte ich zeigen, wie Sie drei Ziele auf einmal erreichen können:

- verbesserte Softwarestruktur durch modellhafte Repräsentation der Interaktionen,
- hohe Entwicklungsgeschwindigkeit,
- Anpassbarkeit.

In dem zum Artikel gehörenden Projekt auf GitHub – verwenden wir Swing. Die Konzepte sind unabhängig vom Technologie-Stack und übertragbar auf andere.

Wo finden Sie den Quellcode?

Sie finden die kommentierten Quelltexte auf der JavaSPEKTRUM-Website und bei GitHub. Unter `www.sigs.de/download/SC/ghadir_SC_JS_05_11.zip` bzw. `http://github.com/pgh` gibt es das Git-Repository *practitioner-at-javaspektrum*, das die hier vorgestellten Experimente in einer lauffähigen Anwendung demonstriert. Suchen Sie dazu das Package `de.ghadir.practitioner.js_2011_05`. Wer kein Git verwenden möchte, kann den Quelltext über die Weboberfläche ansehen.

Sollten sich in den Code Fehler oder aber klare Don'ts eingeschlichen haben, bitte ich um eine kurze E-Mail oder besser noch um entsprechende Pull-Requests oder um das Erstellen eines Tickets. (Ein entsprechendes Issue-Tracking ist bei GitHub direkt integriert und lässt sich ebenfalls über die Weboberfläche verwenden.)

Wer erzeugt wen?

Haben Sie schon einmal mit einem ausgewachsenen UI-Framework versucht, zu Fuß und ohne GUI-Builder Benutzungsoberflächen zu bauen? Je nach Framework variiert der Aufwand. Aber es gibt stets Grenzen, ab denen das systematische Konstruieren ohne GUI-Builder schneller und gleichzeitig zu anpassbareren Ergebnissen führt.

Ein Grund liegt sicherlich darin, dass Verknüpfungen zwischen Oberflächenelementen mit einem GUI-Builder effizient erfasst werden können, aber das Restrukturieren dieser Abhängigkeiten stets den gleichen Ablauf und damit Aufwand erfordert. Dahingegen erfordert eine Anpassung von Konstruktionsregeln häufig nur einen Bruchteil des Aufwands.

Insbesondere für Benutzerschnittstellen ist es häufig sinnvoll, grobgranulare Komponenten zu bauen, die hart komponiert sind. Das Team kann über die Oberfläche sprechen, hat einen klaren Kontext für jede Information und kann die Konfigurationsdetails vernachlässigen. Abbildung 1 und Listing 1 verdeutlichen dies.

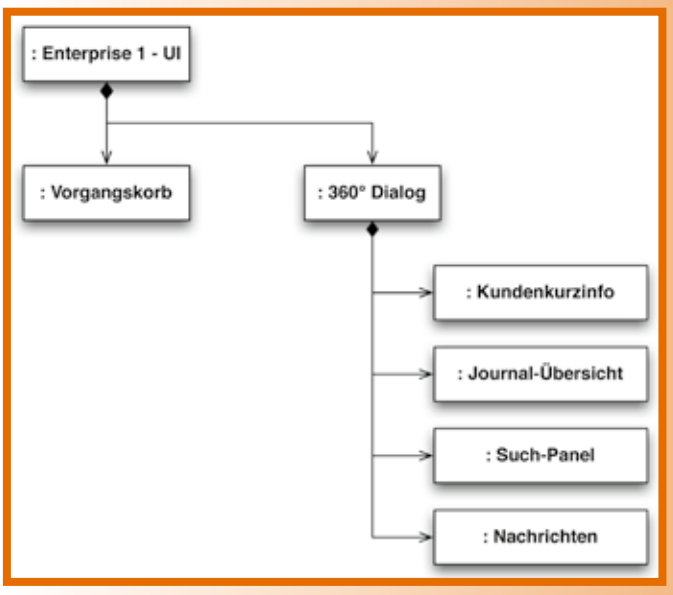

Abb. 1: Hierarchische Zerlegung eines UIs

Abbildung 1 zeigt die Struktur einer Benutzerschnittstelle. Die Komposition zeigt, dass die Benutzungsschnittstelle („Enterprise 1 – UI“) aus zwei Komponenten besteht. Diese werden unmittelbar von der Top-Level-Komponente erzeugt und verwaltet.

Ein aufwendiges Erzeugen, Hinzufügen und Konfigurieren der Einzelteile in einer „unbeteiligten“ Klasse entfällt. Die Einzelteile bestehen wiederum aus anderen Komponenten, die sie selbst erzeugen. Der Quelltext des Hauptfensters sieht also denkbar einfach aus.

```
public Enterprise1UI() throws HeadlessException {
  super( "Enterprise 1 - UI" );
```

```
    add(new Vorgangskorb( this ) );
    add( new Dialog360Grad( this ) );

    linkComponents();
  }
```

Listing 1: Dank der Komposition kann das Objektgeflecht bereits bei Objekt-Konstruktion erzeugt werden

Neulich in einem gar nicht weit entfernten Projekt

Wenn die Komponenten der Benutzerschnittstelle erzeugt, konfiguriert und registriert sind, kann die Anwendung Daten verarbeiten. Diese werden üblicherweise in Modellklassen gehalten. Modellklassen haben je nach Architektur mal mehr oder weniger Fachlichkeit, verfügen aber im Normalfall über eine Reihe von Attributen, die sie über Zugriffsmethoden exportieren.

```
public Enterprise1UI() throws HeadlessException {

@Entity
public class ProjektEntitaet {

  private BigDecimal umsatz;
  private BigDecimal umsatzInput;
  private BigDecimal ertrag;
  private BigDecimal ertragInput;
  private BigDecimal ertragInProz;
  private BigDecimal ertragInProzInput;

 //...
}
```

Listing 2: Ausschnitt aus einem Domänenobjekt aus einem beinahe realen Projekt

Der Code aus Listing 2 ist einem realen Projekt entliehen. Er zeigt eine Reihe von Attributen eines Domänenobjekts. Einige der Attribute dienen dazu, Benutzereingaben zu speichern. Sie enden mit Input. Die anderen Attribute enthalten den jeweils gültigen Wert. Abhängig vom Applikationszustand ist der gültige Wert mal berechnet oder ergibt sich aus dem Input-Wert.

Die dargestellte Entität enthält keinerlei weitere Logik, um die Konsistenz zwischen den Feldern zu erhalten. Die Logik sollte intuitiv wie folgt sein:

- Wird **`ertragInput`** eingegeben, wird **`ertragInProz`** neu berechnet.
- Wird **`ertragInProzInput`** eingegeben, wird **`ertrag`** neu berechnet.
- Die anderen Felder **`ertrag`** und **`ertragInProz`** dienen als Cache.

Es gibt Attribute (die Input-Felder), die eigentlich nichts mit der Fachlichkeit zu tun haben, aber dennoch sichtbar und zugreifbar sind. Zudem ist unklar, wo welche Art von Funktionalität zur Konsistenzerhaltung realisiert werden soll. So gerät selbst ein kleines Softwaregebilde - dank schwacher Kohäsion, hoher Kopplung, schlechter Konstruktionsregeln und mangelnder Trennung der Verantwortlichkeiten - schnell in eine fiese Wartungsfalle.

Am Ende kann eine solche Implementierung allein die Struktur der gesamten Software schwächen, wenn auf ihr eine Applikation gebaut wird.

Übliche Verdächtige

In Geschäftsanwendungen sind es eigentlich immer die üblichen Verdächtigen, die zur Degeneration führen (Natürlich habe ich dazu keine stichhaltige Untersuchung durchgeführt.):

- automatisch berechnete Felder,
- direkt editierbare/berechnete Felder,
- Eingabefelder für komplexe Domänenobjekte.

Automatisch berechnete Felder

Berechnete Felder zeichnen sich dadurch aus, dass sie von Feldwerten anderer Felder abhängig sind. Diese Abhängigkeit realisiert man typischerweise über Listener-Mechanismen: Jedes Feld, das einen Wert berechnen können muss, wird bei den Feldern registriert, die für die Berechnung benötigt werden. Jedes Feld ruft bei Werteänderungen stets alle Listener auf, sodass die Felder im Anschluss rechnen können.

Bei unglücklicher Implementierung außerhalb des Domänenmodells kommt es hier in komplexen Formelsystemen häufig zu unerwünschten Echos und damit zu einer Häufung von Nachrichten, die Rechenorgien auslösen können. Das Problem der vielen Echos lässt sich auch nicht durch geschicktes Cachen in den Griff bekommen, sofern dafür nicht eine spezifische Stelle geschaffen wird.

Im Folgenden schlage ich vor, spezifische UI-Domänenobjekte um das Wissen anzureichern, wie die Änderungspragmatik auf den Datenfeldern aussehen soll. Es gibt Felder, in die man Werte eingeben kann, die aber auch berechnete Werte enthalten können, wenn keine Werte eingegeben werden: Hier nennen wir sie editierbare berechnete Felder. Für diese gibt es zwei Interaktionsmöglichkeiten: das explizite Setzen durch einen Anwender sowie das implizite Berechnen aufgrund bereitgestellter Werte.

Dazu führen wir die Klasse **`EditierbaresBerechnetesFeld`** ein. Sie dient dazu, einerseits die Berechnungsvorschrift für das automatisierte Berechnen des Wertes zu kapseln und andererseits die Abhängigkeit zu den Daten liefernden Feldern automatisch zu pflegen. Mit wenig Aufwand kann man generisch die Anbindung an das Domänenmodell vornehmen und gleichzeitig alle notwendigen Abhängigkeiten registrieren.

```
public class ECField<T extends Comparable>
    extends Field<T>implements FieldListener {
  private static final Parser parser = new Parser();

  Binding env;
  Expression<T> expr;
  private boolean isCalculating = false;
  private T calculated;

  public ECField(Binding env, String formula) {
    this.env = env;
    this.expr = parser.evaluate( formula );
  }

  @Override
  public T getValue() {
    if ( isCalculating ) {
      throw new IllegalStateException( "cyclic dependency!" );
    }

    T temp = super.getValue();
    if ( temp != null )
      return temp;
    else
      return getCalculated();
  }

  @Override
  public void setValue(T input) {
    internalSetValue( input );
    this.calculated = null;

    notifyListeners();
```

```
  }
  public T getCalculated() {

    if ( calculated == null ) {
      calculated = doCalculate();
    }
    return calculated;
  }

  @Override
  public void notifyChange(Field source) {
    calculated = null;
  }

  public T calculate() {
    if (expr != null)
      return expr.evalInCtx(env);
    return null;
  }

  protected void setCalculated(T calculated) {
    this.calculated = calculated;
    internalSetValue( null );

    notifyListeners();
  }

  protected final T doCalculate() {
    try {
      isCalculating = true;
      setCalculated(calculate());
      return getCalculated();
    } finally {
      isCalculating = false;
    }
  }
}
```

Listing 3: Die Klasse EditierbaresBerechnetesFeld (ECField)

Die dargestellte Klasse **`EditierbaresBerechnetesFeld`** (**`ECField`** in Listing 3) erbt verschiedene funktionale Merkmale von der Super-Klasse **`Feld`** (**`Field`** in Listing 3) und fügt weitere Merkmale hinzu:

- Transformation der textuellen Berechnungsvorschrift in ein ausführbares Modell,
- private Aktualisierung des internen Zustands mit verzögerter Benachrichtigung der Listener,
- Erkennen von Zyklen während der Berechnung,
- Benachrichtigung bei Werteänderung,
- Zwischenspeichern des errechneten Werts.

Für die Realisierung einer interaktionsorientierten Domänenklasse möchte ich im Folgenden zwei Strategien gegenüberstellen, die beide die gerade eingeführte **`EditierbaresBerechnetesFeld`**-Klasse verwenden. Jede Strategie hat ihren eigenen Charme, aber auch ihre eigenen Nachteile.

Strategie 1: Ausdekliniertes Domänenobjekt

Die Initialisierung einer solchen Domänenklasse erfolgt sequenziell beim Instanziieren per **`new OrderItem()`** in zwei Phasen: Zuerst werden die Felder definiert und im Anschluss dann untereinander verbunden. Dafür trägt der Block unterhalb von **`// instance initializer`** in Listing 4 Sorge. So kann man sicher sein, dass alle Felder und vorwärts deklarierte Abhängigkeiten definiert sind.

```
public class OrderItem extends Binding {

  private static final OrderItemInitializer initializer =
    new OrderItemInitializer();
  final Field<BigDecimal>
    pricePerUnit = new Field<BigDecimal>(),
    numberOfUnits = new ECField<BigDecimal>( this,
      "totalPrice / pricePerUnit" ),
      total = new ECField<BigDecimal>(
        this, "pricePerUnit * numberOfUnits" );
  // instance initializer
  {
    initializer.registerECFieldListeners(OrderItem.class, this);
  }
}
```

Listing 4: Eine beinahe alltägliche Domänenklasse namens OrderItem

Abb. 2: Die IDE zeigt die Felddefinition

Dieser Ansatz ist deshalb sehr reizvoll, weil gängige Entwicklungsumgebungen per MouseOver über den definierten Feldattributen zeigen, ob das Feld berechnet ist oder eingegeben werden kann und wovon gegebenenfalls die Berechnung abhängt (s. Abb. 2).

Strategie 2: Auslagern der Attribut-Deklaration

Angenommen, der Attribut-Zugriff erfolgt von dem UI und dem Persistenz-Framework stets indirekt, dann könnte man eine generische assoziative Datenstruktur (wie zum Beispiel eine HashMap) mit einer separaten Definition der Attribute ohne Stilbruch kombinieren (s. Abb. 3).

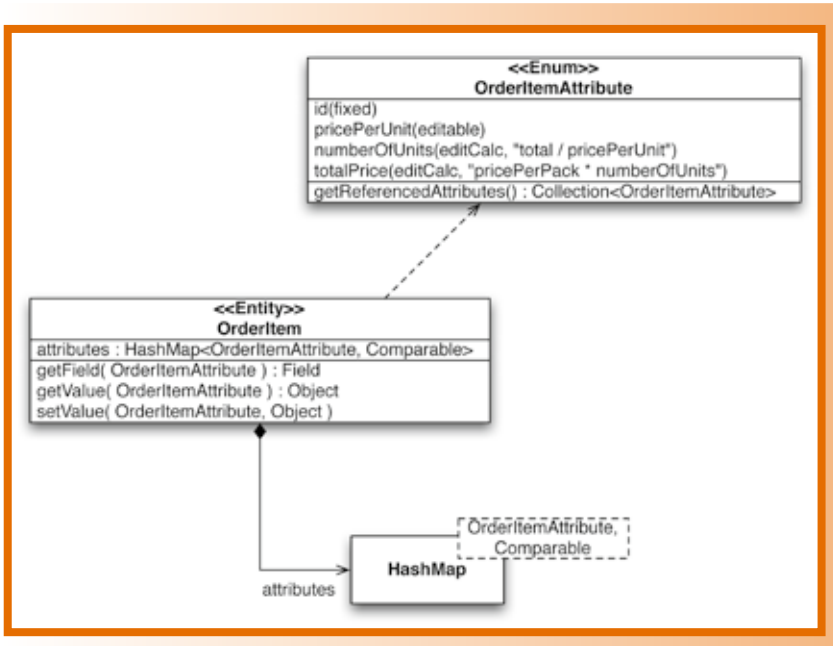

Abb. 3: Struktur von OrderItem gemäß Strategie 2

Das Setzen oder Abfragen der Attribut-Werte erfolgt in diesem Fall dann per

```
orderItem.getValue( OrderItemAttribute.numberOfUnits );
```

bzw.

```
orderItem.setValue( OrderItemAttribute.numberOfUnits, 10 );
```

Ähnlich wie bei der ersten Strategie lässt sich sehr einfach überprüfen, wie die Werte berechnet werden. Die Softwareentwicklung und -wartung wird damit erleichtert. Im Vergleich zur ersten Strategie kann der Zugriff aus UI-Komponenten jetzt einfach über die generischen Zugriffsmethoden erfolgen, ohne dass Reflektion verwendet werden muss.

Leider zeigt die Entwicklungsumgebung nicht bei Mouse-Over die Definition der Attribute, sondern erst beim Sprung zur Definition.

Erst zur Laufzeit

Beide Umsetzungsstrategien erlauben das elegante und redundanzfreie Beschreiben der Abhängigkeiten von Attributen untereinander. Der Preis dafür ist allerdings, dass erst zur Laufzeit festgestellt werden kann, ob die angegebenen Berechnungsvorschriften syntaktisch korrekt sind.

Andererseits legt die hier vorgestellte Verwendung nahe, die Abhängigkeiten direkt zu deklarieren, sodass man eine vollständige Auswertung der Abhängigkeiten erhält, ohne komplexe Tests mit dem Domänenobjekt durchführen zu müssen.

Eines aus einer Menge

Die Deklaration im Kleinen erfolgt für den Entwickler intuitiv und kann sich einfach an einem beliebigen Fachkonzept orientieren. Manchmal gibt es aber auch Zusammenhänge, bei denen aus einer Menge verschiedener Optionen nur ein Wert eingegeben wird, während die anderen Werte automatisch berechnet werden müssen.

Hierzu gibt es wiederum verschiedene Strategien. Zum Beispiel könnten wir einen weiteren Konstruktor in die Klasse `EditierbaresBerechnetesFeld` (Klasse `ECField` in Listing 3) mit einem zusätzlichen Parameter hinzufügen, der angibt, welche anderen Felder (unabhängig von dem Datenfluss bei der Berechnung des Wertes) zurückgesetzt werden müssen, sobald das Feld editiert wird.

Eine bessere Lösung ist allerdings die Bereitstellung einer eigenständigen Äquivalenzrelation, mit der man – einmal – deklariert, welche Elemente zusammenhängen:

```
{
  mutex( pricePerUnit, priceTotal );
}
```

In beiden Strategien funktioniert diese Deklaration (es ist eigentlich eine Anweisung) wieder innerhalb eines Initializer-Blocks. Da dieser unterhalb aller Attribut-Deklarationen aufgeführt werden muss, kann man hier bereits auf die initialisierten Attribute (Strategie 1) zugreifen oder aber direkt die Aufzählungswerte (Strategie 2) verwenden. Das heißt, hier steht die Code-Vervollständigung zur Verfügung.

Fazit

Im Kleinen haben wir jetzt Strukturen geschaffen, die uns erlauben, unsere Benutzerschnittstelle robuster zu gestalten, weil unser Domänenmodell die Interaktionsmetaphern unterstützt, die wir vorhergesehen haben, ohne die Realisierung der Benutzungsoberfläche selbst zu überlassen.

Wir haben durch hierarchische (De-)Komposition komplexere UI-Elemente zusammengesteckt und erreichen über die stereotypische sequenzielle Initialisierung die passende Verknüpfung der UI-Elemente.

Manager-Funktionalität, die die gesamte Applikationslogik verschmutzt, kann durch so einen Ansatz vermieden werden.

Bleibt die Frage, was machen innovative Benutzerschnittstellen eigentlich noch aus?

Literatur und Links

[DDD] E. Evans, Domain-Driven Design, Addison-Wesley, 2003

[GitHub] `http://github.com/pgh`

[JavaCodeConvetions]
`http://www.oracle.com/technetwork/java/codeconvtoc-136057.html`

Eine Randbemerkung zu den Sichtbarkeiten

Beim Anschauen des Quelltextes (siehe GitHub) werden einige Leser feststellen, dass nicht selten die Sichtbarkeit auf *Package-Protected* gesetzt ist. Dies ist kein Flüchtigkeitsfehler, sondern Absicht: Java bietet die Möglichkeit, nicht nur Klassen zum Kapseln von Verantwortung zu verwenden. Wir können die Funktionalität geeignet über Frameworks und Hilfskonstrukte integrieren, ohne den Nutzern der Klassen aufzubürden, alle Annahmen zu kennen, die für die erfolgreiche Ausführung notwendig sind. Daher fassen Packages verschiedene Aspekte zusammen und ermöglichen nach außen hin einen aufgeräumten, schlanken Eindruck.

Das Deklarieren mancher Methoden als `final` garantiert grundsätzliche Zusicherungen – wie zum Beispiel, dass sich `doCalculate()` merkt, dass der Feldwert gerade berechnet wird (s. Listing 3).

Sourcen zu dieser Kolumne

`www.sigs.de/download/SC/ghadir_SC_JS_05_11.zip`
`https://github.com/pgh/practitioner-at-javaspektrum`

Phillip Ghadir baut am liebsten tragfähige, langlebige Softwaresysteme. Er ist Mitglied der Geschäftsleitung bei innoQ und hat sich früh auf Architekturen für verteilte, unternehmenskritische Systeme spezialisiert. Darüber hinaus ist er Mitbegründer und aktives Mitglied des iSAQB, des International Software Architecture Qualification Board.
E-Mail: phillip.ghadir@innoq.com