Einsatz domänenspezifischer Sprachen zur Migration von Datenbankanwendungen

Sven Efftinge[1], Sören Frey[2], Wilhelm Hasselbring[2], Jan Köhnlein[1]
[1]itemis AG
Schauenburgerstraße 116, 24118 Kiel
http://www.itemis.de/
[2]Universität Kiel
Institut für Informatik, Software Engineering, 24118 Kiel
https://se.informatik.uni-kiel.de/

Abstract: In der modellgetriebenen Softwareentwicklung von Datenbankanwendungen werden domänenspezifische Sprachen zusammen mit Codegeneratoren eingesetzt, welche aus formalen Modellen automatisiert lauffähige Software erzeugen. Selbstdefinierte domänenspezifische Sprachen werden zusammen mit dazu selbstentwickelten Codegeneratoren eingesetzt, sie können aber auch direkt in Wirtsprachen integriert werden, ohne eine spezielle Werkzeugumgebung zu benötigen. Sogenannte *externe* domänenspezifische Sprachen werden durch einen selbst erstellten Parser verarbeitet, während *interne* domänenspezifische Sprachen in eine Wirtsprache eingebettet werden und somit deren Parser mit nutzen.

Für das Szenario der Migration von Datenbankanwendungen (von OracleForms zu Java Swing) präsentieren wir den Einsatz externer und interner domänenspezifischer Sprachen für unterschiedliche Aufgaben eines größeren Migrationsprojektes. Wir betrachten die Überlegungen zur Migrationsarchitektur für den konkreten Kontext des Projekts Forms2Java anhand des Dublo-Migrationsmusters. Neben der eigentlichen Vorstellung der Sprachen diskutieren wir auch die generelle Fragestellung, wann sich der mit dem Einsatz domänenspezifischer Sprachen verbundene Aufwand lohnt.

1 Einleitung

Die Stärken eines großen Teils der deutschen Softwareindustrie sind geprägt durch ein tiefes Verständnis der jeweiligen Anwendungsdomänen. Das wohl bekannteste Beispiel dazu ist die SAP mit ihren betriebswirtschaftlichen Anwendungssoftwaresystemen. Es gibt aber auch sehr viele weitere Unternehmen, die erfolgreich Standard- und Individualsoftware für bestimmte Domänen entwickeln. Eine zentrale Frage ist dann, wie diese Stärken in den jeweiligen Anwendungsdomänen effektiv und effizient in der eigentlichen Softwareentwicklung unterstützt werden können. Dazu bieten sich u.a. die modellgetriebene Softwareentwicklung und der damit verbundene Einsatz sogenannter domänenspezifischer Sprachen (englisch: Domain-Specific Languages, DSL) als viel versprechende Ansätze an.

Modellgetriebene Softwareentwicklung ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen [SVE07]. Dabei werden DSLs zusammen mit entsprechenden Codegeneratoren eingesetzt. Im Bereich der modellgetriebenen Entwicklung richten sich DSLs verstärkt auf die Lösung von Problemen in einer

bestimmten Anwendungsdomäne aus. Oft wird ein generativer Ansatz verfolgt, um die Modelle der DSLs durch eine automatische Transformation in ausführbaren Code zu überführen. Im Gegensatz zu sogenannten „General Purpose Languages“ sind DSLs auf ein Anwendungsgebiet optimal zugeschnitten und abstrahieren von der zugrunde liegenden Programmierplattform. Somit können domänenspezifische Sachverhalte viel präziser und knapper formuliert werden.

DSLs können anhand des Einbettungsgrades unterschieden werden: Sprachen, deren eigene Syntax durch einen selbst erstellten Parser verarbeitet wird, nennt man *extern*. Sprachen, die in ihre Wirtsprache eingebettet sind und damit lediglich eine Spezialisierung dieser Wirtsprache darstellen, nennt man *intern*. In diesem Papier stellen wir einen Ansatz zur Migration von Datenbankanwendungen vor, in dem die folgenden Arten von DSLs eingesetzt werden:

- Eine externe DSL dient zur Datenmodellspezifikation, aus der der Datenbankzugriffscode generiert wird (hier implementiert mit der Java Persistence API).
- Eine interne DSL dient zur Spezifikation der Integritätsbedingungen, die bei der Dateneingabe überprüft werden (hier mit Java als Wirtsprache).
- Eine externe DSL dient zur Spezifikation des Layouts der Benutzungsschnittstellen, aus der Eingabemasken für die Datenbank generiert werden (hier implementiert mit Java Swing und unterstützt durch einen speziellen GUI-Editor).

Dieser Ansatz wurde in einem größeren Projekt erfolgreich eingesetzt. Eine wichtige Frage in diesem Zusammenhang ist es immer, ab wann sich der mit dem Einsatz einer DSL verbundene Aufwand rechnet; dieser Fragestellung werden wir ebenfalls diskutieren.

Im Folgenden werden wir zunächst in Abschnitt 2 auf den Projektkontext mit einigen quantitativen Angaben zum Projektumfang und einer Beschreibung der Migration von Datenbankanwendungen mit dem Dublo-Migrationsmuster eingehen. Die oben erwähnten internen/externen DSLs werden in den Abschnitten 3 bis 5 vorgestellt. In Abschnitt 6 diskutieren wird die Kosten und das Einsparpotenzial des vorgestellten Ansatzes, Abschnitt 7 stellt verwandte Arbeiten vor, bevor Abschnitt 8 das Papier zusammenfasst und einen Ausblick auf geplante Arbeiten liefert.

2 Projektkontext und Migrationsarchitektur

Im Projekt *Forms2Java* wurde eine Datenbankanwendung der APG Affichage[1] zur Verwaltung von Geschäfts- und Kundendaten migriert. Das Altsystem *Gepard* ist eine Datenbankanwendung, die mittels der Oracle-Datenbank und OracleForms für die Entwicklung der Benutzungsschnittstellen programmiert wurde.

Weil die über Jahre gewachsenen OracleForms sich zunehmend schlechter warten ließen und gleichzeitig die Zukunft der OracleForms-Technologie selbst infrage stand, entschloss

[1] `http://www.apgsga.ch/`

man sich zu einer Migration auf eine besser skalierende, zukunftssichere Plattform unter Beibehaltung des unternehmenskritischen Datenbestands. Die Altanwendung umfasst 1722 Tabellen mit 19572 Spalten und über 300 Forms.

Es gibt verschiedene Möglichkeiten, ein Altsystem in eine neue Architektur zu migrieren. Altsysteme stellen wichtige Investitionen dar, die nicht einfach außer Betrieb genommen werden können. Der Betrieb muss während des Übergangs weitergehen. Folglich sind sanfte Migrationspfade und die Integration von Alt- und Neusystemen essenziell für die Praxis der Integration von Informationssystemen [BS95, HKRS08].

In [HBG+08] wurden auf Basis der Erfahrungen aus mehreren Migrationsprojekten, in denen insbesondere im Bereich der Datenbankintegration immer wiederkehrende Probleme auftraten, drei Varianten des ursprünglichen Dublo-Musters [HRJ+04] abgeleitet:

Dublo Service Die Dublo-Service-Lösungsstruktur ist in Abbildung 1(a) dargestellt. Die Grundidee besteht in der Entwicklung der Geschäftslogik in der neuen Geschäftslogikschicht, der Erstellung eines Legacy-Adapters für den Zugriff der neuen Geschäftslogik auf die existierende Legacy-Geschäftslogik und der Benutzung dieses Adapters für den Datenzugriff. Folglich wird auf die Datenbank nur indirekt durch den vorhandenen Legacy-Code zugegriffen.

Dublo Old Database Die Dublo-Old-Database-Lösungstruktur ist in Abbildung 1(b) dargestellt. Die Grundidee besteht darin, dass im Gegensatz zum Dublo-Service-Muster direkt auf die Legacy-Datenbank zugegriffen wird. Diese Strategie erhält die alte Datenbank und ersetzt die alte Kombination aus Präsentations-, Geschäfts- und Datenzugriffsebene.

Dublo New Database Die Dublo-New-Database-Lösungstruktur ist in Abbildung 1(c) dargestellt. Die Grundidee besteht darin, parallel zum Altsystem eine ganz neue Infrastruktur aufzubauen.

Die Anwendung des Dublo-Service- und des Dublo-Old-Database-Musters ist sinnvoll, wenn ein inkrementeller Austausch alter Geschäftslogik- und Client-Software durch neue Geschäftslogik in der Mittelschicht angestrebt wird. Da keine zusätzliche Datenbank eingeführt wird, entstehen keine Konsistenz- oder Abgleichsprobleme zwischen neuer und alter Datenbank. Mit dem Dublo-Service-Muster ist es für die neuen Clients transparent, ob Geschäftslogik bereits in der neuen Mittelschicht oder noch im alten Legacy-Code implemetiert ist.

In [HBG+08] werden Kriterien und Erfolgsfaktoren zum Einsatz der jeweiligen Dublo-Variante angegeben. Für das Projekt *Forms2Java* ist Dublo Old Database die richtige Variante, weil ein Parallelbetrieb von Alt- und Neusystem in einer Übergangszeit erforderlich ist und das existierende Datenbankschema ausreichend gut strukturiert ist. Das Datenbankschema von Gepard wurde stets diszipliniert weiterentwickelt, so dass es auch zukünftig eingesetzt werden kann. Die darauf aufsetzenden Anwendungen wurden nun migriert, von OracleForms zu Java Swing. Ein weiterer Erfolgsfaktor für das Dublo-Old-Database-Muster besteht darin, dass davon ausgegangen werden kann, dass das DBMS vom Hersteller weitergepflegt werden wird.

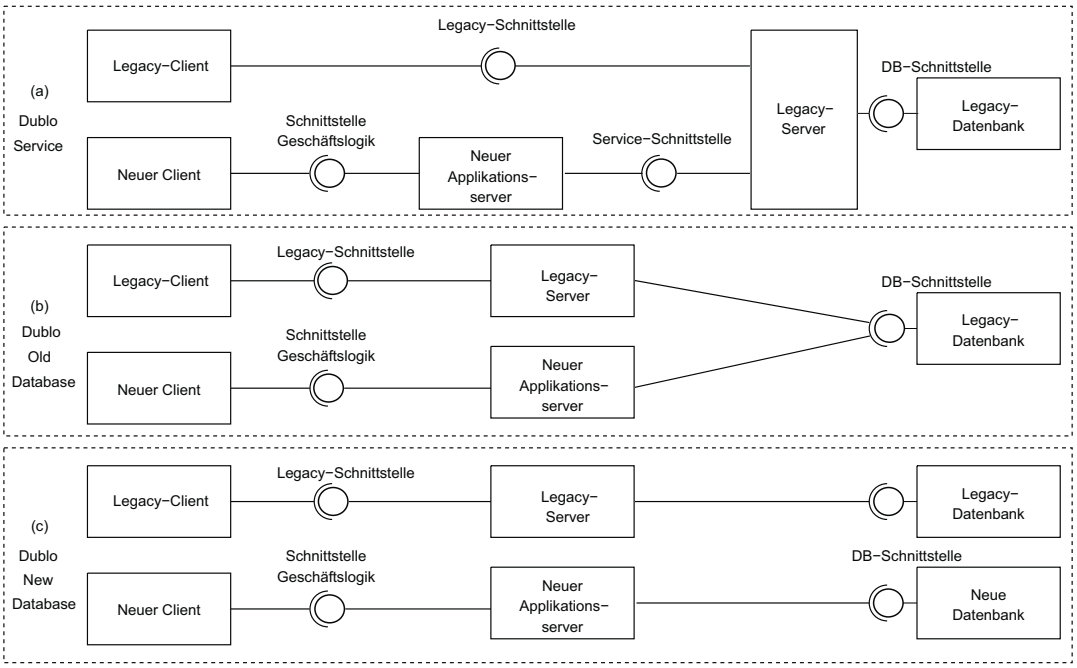


Abbildung 1: Die Varianten des Dublo-Musters [HBG+08].

Die Zielarchitektur für das neue *Gepard*-System nutzt, wie das Altsystem, die Oracle-Datenbank, für die Anwendungsprogrammierung werden jedoch die Java Persistence API[2] und das Spring Framework[3] verwendet. Die neuen Benutzungsschnittstellen werden als „Rich Clients“ mit der Programmierschnittstelle und Grafikbibliothek Swing[4], sowie dem JGoodies Framework[5] realisiert.

Abbildung 2 zeigt eine Übersicht über die aktualisierte Architektur der Anwendung *Gepard* und die Rollen der eingesetzten DSLs im Konstruktionsprozess. Auf der rechten Seite sind die vier Schichten des Anwendungssystems dargestellt, welches entwickelt werden soll. Die interne DSL für die Eingabeprüfung wird in Abschnitt 4 vorgestellt, bevor die GUI-DSL für das Layout der Eingabemasken in Abschnitt 5 erläutert wird. Im folgenden Abschnitt 3 zeigen wir, wie eine mittels TMF Xtext[6] implementierte externe DSL für die Beschreibung des Datenmodells verwendet werden kann. Diese externe DSL referenziert aus dem Datenbanksystem exportierte Schema-Informationen und wird mittels XPand aus dem openArchitectureWare-Projekt[7] transformiert, welches inzwischen in das Eclipse Modeling Projekt überführt wurde.[8] Die interne Java-DSL wird über eine Java-5-Annotation in die Eingabeprüfung integriert, siehe Abschnitt 4. Für die GUI-DSL wurde ein spezifischer graphischer Editor auf Basis des Graphical Modeling Frameworks[9] entwickelt, siehe Abschnitt 5.

[2] http://www.oracle.com/technetwork/java/javaee/tech/
[3] http://www.springsource.org/
[4] http://download.oracle.com/javase/tutorial/uiswing/
[5] http://www.jgoodies.com/
[6] http://www.xtext.org/
[7] http://www.openarchitectureware.org/
[8] http://www.eclipse.org/modeling/
[9] http://www.eclipse.org/gmf/

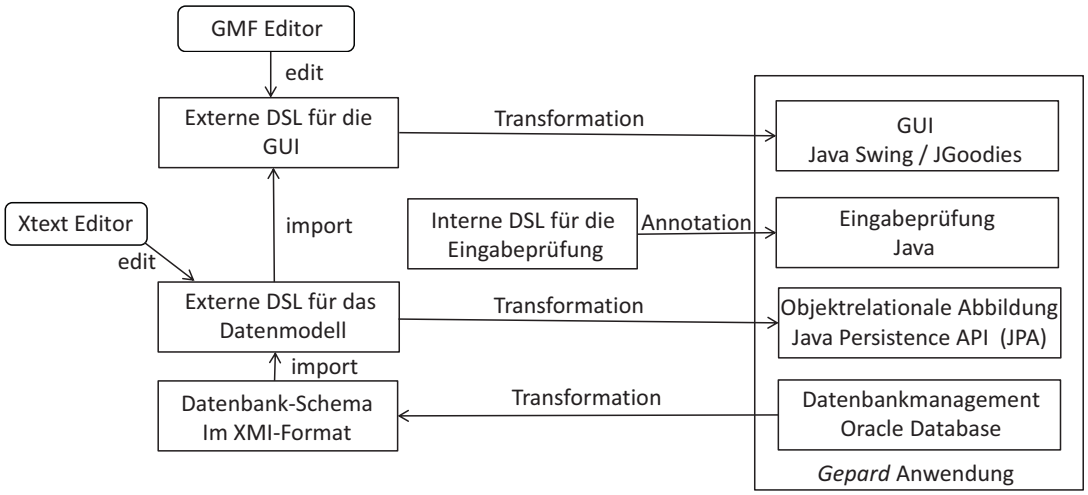


Abbildung 2: Übersicht über die aktualisierte Architektur der Anwendung *Gepard* (rechts) und die Rollen der eingesetzten DSLs im Konstruktionsprozess.

3 Einsatz einer externen DSL für das Datenmodell

Einen kleinen Ausschnitt des Programmcodes für den Zugriff aus Java mit der Java Persistence API (JPA) auf das relationale DBMS zeigt Abbildung 3(a). Die JPA wurde mit Version 5 der Java EE Plattform eingeführt. Sie stellt eine standardisierte Schnittstelle für Java-Anwendungen dar, die die Zuordnung und die Übertragung von Laufzeit-Objekten einer Java-Anwendung in relationale Datenbanken ermöglicht, also eine objektrelationale Abbildung implementiert. Die JPA zeichnet sich durch einen umfangreichen Einsatz von Java 5 Annotationen aus, wie es in Abbildung 3(a) zu sehen ist. Da die Datenbank ohne Änderungen migriert werden soll, können die Voreinstellungen der JPA in vielen Fällen nicht benutzt werden, was die Anzahl der benötigten Annotationen drastisch erhöht. Dieser Code wiederholt sich vom Grundmuster her immer wieder, so dass es bei der Größe der zu migrierenden Anwendung lohnenswert erscheint, diese Zugriffsschicht aus einer Datenmodell-DSL zu generieren.

Eine externe DSL zur Beschreibung des Datenmodells der *Gepard*-Datenbank wurde mittels Xtext realisiert. Ein Beispiel für die Programmierung mit dieser DSL ist in Abbildung 3(b) dargestellt. Generiert wird aus dieser Datenmodell-DSL die objektrelationale Abbildung, wie sie in Abbildung 3(a) zu sehen ist.

Defaults für das Datenmodell werden aus dem Schema der Datenbank extrahiert, siehe Abbildung 2. Um eine gute Referenzierbarkeit des Datenbankschemas aus der Datenmodell-DSL zu ermöglichen, wird die Schemainformation aus der Datenbank in eine XMI-Datei transformiert. XML Metadata Interchange (XMI)[10] ist ein Standard-Austauschformat für Modelle zwischen Software-Entwicklungswerkzeugen, welcher insbesondere für den Austausch von UML-Modellen entwickelt wurde.

Falls die Datenstrukturinformationen aus dem Datenbankschema ausreichen, um die objektrelationale Abbildung zu generieren, genügt die folgende Datenbank-DSL-Spezifikation:

[10] `http://www.omg.org/spec/XMI/`

(a)

```
@SuppressWarnings("serial")
@Entity
@Table(name = "BUCHUNGSKREISE_F")
public class BuchungskreiseF
extends AbstractEntity implements Serializable {

@SuppressWarnings("unused")
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE,
generator = "bkrIdSeq")
@SequenceGenerator(name = "bkrIdSeq", sequenceName = "SEQ",
allocationSize = 1)
@Column(name = "BKR_ID", nullable = false)
private Long bkrId;
public Long getBkrId() {
  return bkrId;
}
public void setBkrId(final Long bkrId) {
  this.bkrId = bkrId;
}
@Column(name = "KONTO_NR", nullable = false, length = 45)
private String kontoNr;
public String getKontoNr() {
  return kontoNr;
}
public void setKontoNr(final String kontoNr) {
  String oldValue = this.kontoNr;
  this.kontoNr = kontoNr;
  firePropertyChangeEvent("kontoNr", oldValue, this.kontoNr);
}
```

(b)

```
import dbschema.xmi  // Das aus der Datenbank extrahierte
                     // Schema im XMI-Format

entity BuchungskreiseF
  (id=bkrId sequenceName=BKR_SEQ) {
  String kontoNr (notNull, length=45)
  Long rgNrBkrIdentifikator (notNull, length=1)
  String referenzcodeKontoNr (notNull, length=45)
}
```

Abbildung 3: Generierter Programmcode für den Zugriff auf das DBMS aus Java mit der JPA (a) und das Datenmodell spezifiziert mittels der externen Datenmodell-DSL (b).

```
entity BuchungskreiseF
   (id=bkrId sequenceName=BKR_SEQ) {
}
```

Nur dann, wenn es Abweichungen gibt, beispielsweise zwischen der Namensgebung im Java-Programm und dem Datenbankschema, müssen diese explizit spezifiziert werden, um den Code aus Abbildung 3(a) zu generieren, wie es in Abbildung 3(b) zu sehen ist. Durch dieses Prinzip werden die DSL-Programme klein gehalten und die Ausnahmen sind explizit sichtbar. Abbildung 4 zeigt, wie die aus der Datenbank importierten Schemainformationen im Xtext-Editor genutzt werden können, um diese Schemainformationen in der Datenmodell-DSL bei Bedarf anpassen zu können.

Abbildung 4: Anpassung der Schemainformationen im Xtext-Editor.

4 Einsatz einer internen DSL für die Eingabeprüfung

Bei der Implementierung von Validierungsregeln für die Eingabeprüfung geht es im Wesentlichen darum Bedingungen zu definieren, deren Überprüfung geeignete Fehlermeldungen generiert, falls diese Bedingungen nicht erfüllt sind. Abbildung 5(a) zeigt die Eingabeprüfung mit Java, wie sie zunächst im vorgestellten Migrationsprojekt ohne eine speziell zugeschnittene interne DSL durchgeführt wurde. Auch hier gibt es für alle Eingabeprüfungen immer wiederkehrende Konstrukte. Nach eingehender Analyse der Überprüfungsfunktionen wurde eine interne DSL für Java entworfen, für die ein Beispiel in Abbildung 5(b) dargestellt ist.

Ein Kernkonzept zur Lösung dieses Problems besteht in der Möglichkeit, Boolesche Ausdrücke zu formulieren. Solche Ausdrücke können in Java selbst gut formuliert werden. Bei einer externen DSL müsste man dies neu implementieren, was sehr aufwändig ist. Daher macht es hier Sinn die Beschreibung von Validierungsregeln innerhalb von Java so prägnant wie möglich zu machen – sprich die API optimal in Form einer internen DSL auf das

(a)

```
addValidator(new Validator<Institutionen>() {
  @Override
  public ValidationResult validate(final Institutionen
    institution) {
    final ValidationResult result = new ValidationResult();
    if (institution != null && institution.getEsrNr() != null
        && !CheckUtils.checkPcKontoNrPruefziffer(
           Long.parseLong(institution.getEsrNr())) {
       result.add(new SimpleValidationMessage(
           getResourceMap().getString("validation.esr.msg"),
           Severity.ERROR, getModel(Institutionen.DESC.
           esrNr())));
    }
    return result;
  }
});
```

(b)

```
@Check
void checkEsrMsg() {
  if (!checkKontoNrPruefziffer(parseLong(this.getEsrNr())))
    error("validation.esr.msg",desc.esrNr());
}
```

Abbildung 5: Eingabeprüfung mit Java ohne interne DSL (a) und mit interner DSL (b).

Problem zu zu schneiden.

Dazu wurde zunächst eine Java-5-Annotation `@Check` eingeführt, mit der Validierungsmethoden ausgezeichnet werden können, siehe Abbildung 5(b). Die so annotierten Methoden werden dann durch ein speziell dazu implementiertes Framework reflektiv identifiziert und automatisch für Instanzen des angegebenen Typs (hier: `Konto`) aufgerufen. Bei genauerer Betrachtung der in Abbildung 5(a) gezeigten Validierungsregel fällt auf, dass viel Code durch geeignete Bibliotheksfunktionen vermieden werden kann. Beispielsweise kann das explizite Erzeugen von `ValidationResult` und `SimpleValidationMessage` implizit durch das Framework geschehen. Da die Methode vom Framework aufgerufen wird, reicht ein Aufruf einer `error(String,Object)`-Methode. Auch die Prüfungen auf Nullreferenzen, wie wir sie in Abbildung 5(a) finden, waren in allen bestehenden Validierungsregeln vorhanden. Die Semantik war dabei immer gleich: Wenn es eine Nullreferenz gibt, dann gilt die Bedingung nicht. Um auch diese Redundanz zu entfernen, wurde die Semantik der `@Check`-Methoden so geändert, dass `NullPointerExceptions` gefangen werden und dazu führen, dass der Check nicht ausgeführt wird. Über eine weitere optionale Annotation (`@NullPointerAware`) kann diese Semantik wieder abgeschaltet werden. Die in Abbildung 5(a) dargestellte Validierungsregel sieht in der internen DSL dann wie in Abbildung 5(b) dargestellt aus.

```
public class PersonenForm extends
Form<Personen> { ... }

public class PersonenHauptSubForm extends
SubForm<Personen> {

private JComponent vornameTextField;

@Override
protected void initComponents() {
  ...
  vornameTextField = builder.createTextField(desc.vorname(),
    Editable.PROPERTY_DEFAULT, MANDATORY);
  gepardBuilder.setNoLeadingBlanks (vornameTextField);

@Override
  protected JComponent buildPanel() {
    TwoColumnsPanelBuilder builder =
      TwoColumnsPanelBuilder.instance(getBuilderFactory(),
      getResourceMap());
    ...
    builder.add("vorname", vornameTextField);
}
```

Abbildung 6: Programmcode für die Eingabemasken mit Java Swing (kleiner Ausschnitt).

5 Einsatz einer externen DSL für die Eingabemasken

Abbildung 6 zeigt einen kleinen Ausschnitt des Programmcodes für die Eingabemasken mit Java Swing. Dieser musste bisher manuell programmiert werden. Abbildung 7 zeigt unsere GUI-DSL, aus der nun der Java Swing Code aus Abbildung 6 generiert wird. Eine Beispielmaske der neuen *Gepard*-Anwendung, die mit Java Swing realisiert wurde, wird in Abbildung 8(a) dargestellt. Auch der Java-Swing-Code wiederholt sich vom Grundmuster her immer wieder, so dass es genau wie für die Datenzugriffsschicht aus Abschnitt 3 lohnenswert erscheint, diesen Code aus einer GUI-DSL zu generieren.

Für die Festlegung des konkreten Layouts einer grafischen Benutzungsoberfläche bietet es sich an, GUI-Editoren zu nutzen, die mit dem ergonomischen Konzept der direkten Manipulation arbeiten [SPC09]. Für Java Swing stehen bereits leistungsfähige GUI-Editoren zur Verfügung. Diese haben im hier beschriebenen Szenario allerdings einige Nachteile. Sie bieten immer alle Einstellungsmöglichkeiten der Java-Swing-Elemente an, und verstellen dadurch den Blick auf die wesentlichen Punkte, welche im betrachteten Projekt relevant sind. Die automatische Generierung von Formularen wird zwar druch Java Swing unterstützt, basiert aber dann auf Java Beans. Die Abstraktionen der Datenmodell-DSL wären damit nicht mehr nutzbar.

Im vorgestellten *Forms2Java*-Projekt wird aus der Datemmodell-DSL automatisch eine Standarddarstellung von Datenmodellen in Java-Swing-Formularen generiert. Die Dar-

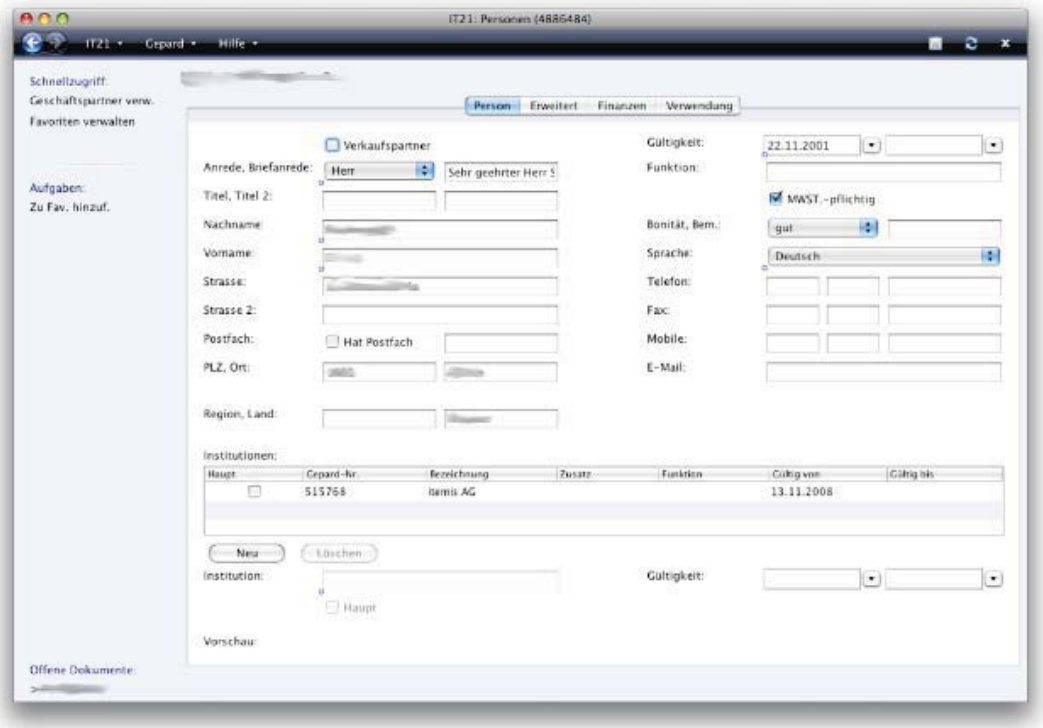
```
model : gepard;
import "platform:/src/main/model/types.dao"
com.affichage.it21.gp.dao {
  flaechen {
   readOnly entity WaehrungF (id =(rvLowValue)) {
   }
  readOnly entity GepardVerwendungPvF (id =(pvOid)) {
   temporal manyToOne GeschpartnerAllBsF geschpartner ()
    }
  }
  verkauf {
   readOnly entity GepardVerwendungKdvtF (id =(kdvtId)) {
    temporal notNull manyToOne
                      GeschpartnerAllBsF geschpartner ()
    notNull Number istLangfrist (castTo=Boolean)
    notNull Number istLokaldispo (castTo=Boolean)
  }
}
```

Abbildung 7: Beispiel für die GUI-DSL.

stellung von Datenmodellen in Formularen sollte im *Forms2Java*-Projekt aber auch selbst kontrolliert werden können. Daher wurde eine eigene GUI-DSL entwickelt, in der die Elemente der Datenmodell-DSL referenziert und zusätzliche Layout-Informationen spezifiziert werden können, sofern sie von der selbst bestimmten Standarddarstellung abweichen. Um die Vorteile der direkten Manipulation zu nutzen, wurde für die GUI-DSL statt eines textuellen Xtext-Editors ein spezifischer graphischer Editor auf Basis des Graphical Modeling Frameworks[11] entwickelt, siehe Abbildung 8(b). Angemerkt sei, dass es hierbei um eine ergonomische Oberfläche für die Entwickler geht, die Ergonomie der damit erstellten Oberflächen für die Fachanwender (Abbildung 8(a)) ist ein weiteres wichtiges Thema, das wir an dieser Stelle nicht behandeln. In der Mitte von Abbildung 9 ist ein kleiner Beispielextrakt aus der GUI-DSL zu sehen, welcher die Datenmodell-DSL importiert. Die GUI-DSL stellt somit die Verbindung zwischen dem Datenmodell und der GUI her.

[11] http://www.eclipse.org/gmf/

(a)



(b)

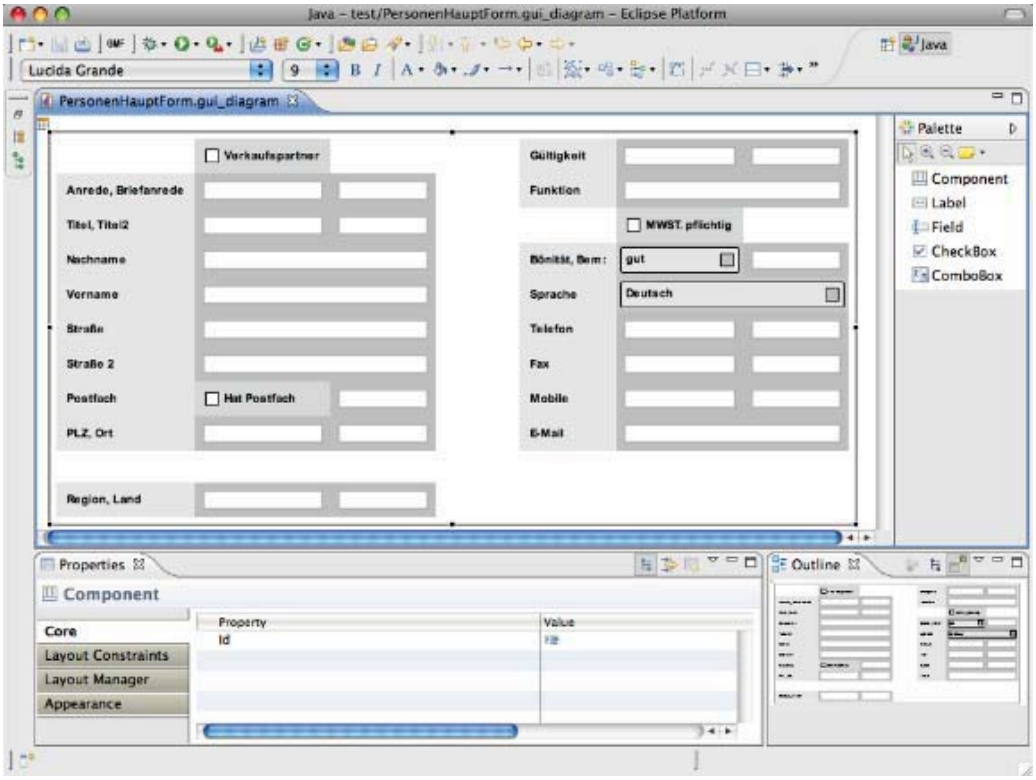


Abbildung 8: Beispielmaske (a) und GUI-Editor (b).

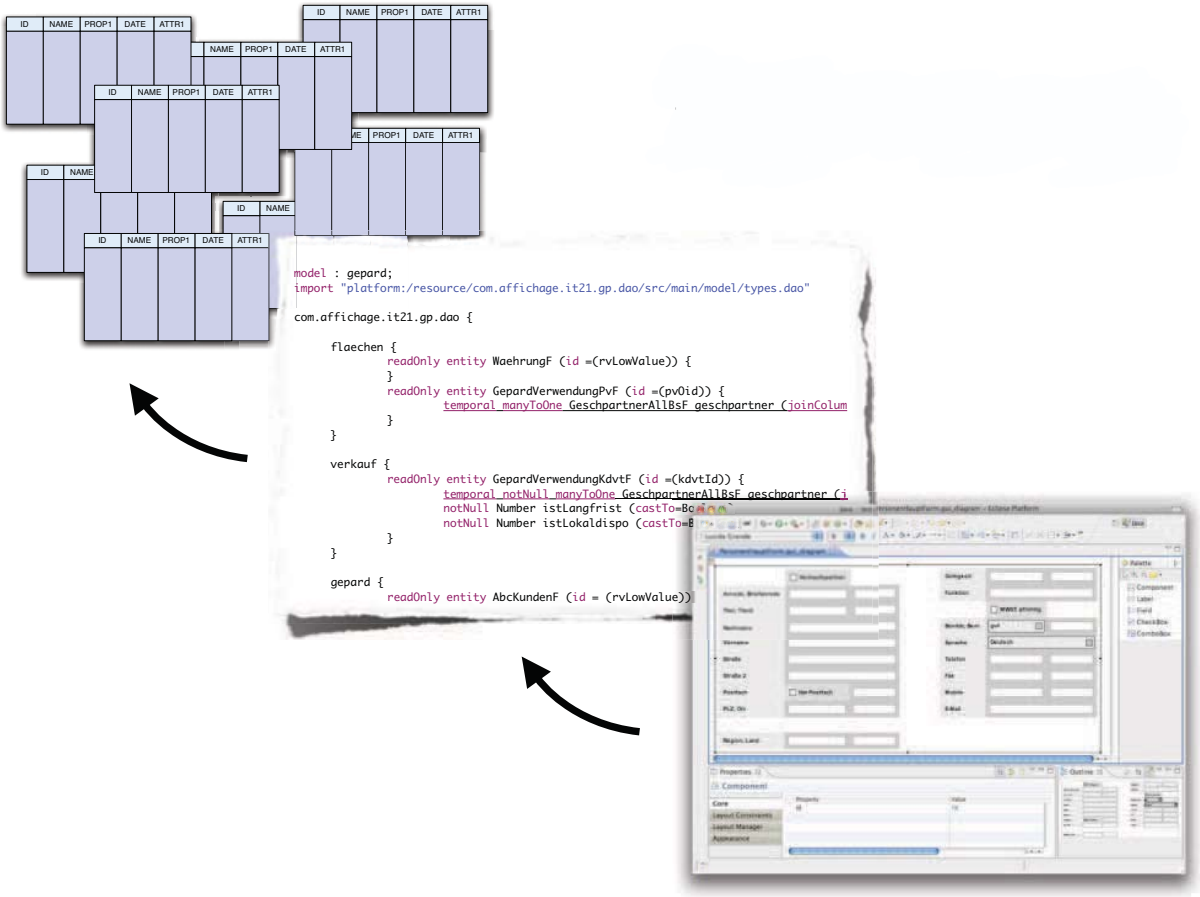


Abbildung 9: Die GUI-DSL als Verbindung zwischen dem Datenmodell und der GUI.

6 Wann lohnt sich der Einsatz welcher Art von DSL

Bei der Bewertung der modellgetriebenen Entwicklung im Vergleich zu traditionellem Vorgehen müssen grundsätzlich zahlreiche Faktoren beachtet werden. Da sich der Vorteil der modellgetriebenen Softwareentwicklung nur bedingt durch finanzwirtschaftliche Größen ausdrücken lässt, müssen sowohl monetäre als auch nicht monetäre Aspekte Berücksichtigung finden. Folgende, nicht monetär quantifizierbare Faktoren stellen einen Vorteil der modellgetriebenen Entwicklung dar [SVE07]:

1. Modellgetriebene Softwareentwicklung verschafft einen strategischen Geschäftsvorteil. Sobald eine DSL und die zugehörigen Transformationen und Generatoren für eine Plattform existieren, ist es möglich, sehr schnell neue Anwendungen aus demselben oder einem ähnlichen Problembereich zu implementieren.
2. Es wird primär fachliches Know-how zur Umsetzung neuer Anforderungen benötigt. Die Technologie tritt in den Hintergrund. Das Wissen über technische Projektrahmenbedingungen ist implizit im Generator abgelegt und kann sehr einfach wiederverwendet werden, ohne dass alle Mitarbeiter von vornherein ein tiefes Verständnis für die eingesetzte Technologie haben müssen.
3. Es findet eine klare und formale Trennung zwischen manuell erstelltem fachlichen und generiertem technischen Code statt (separation of concerns).
4. Automatisierung reduziert die Anzahl potenzieller Fehlerquellen.
5. Bestehende Modelle der Anwendung sind robust gegenüber Technologieänderungen. Dies entkoppelt den Anwendungslebenszyklus vom Technologielebenszyklus.

Darüber hinaus können verschiedene finanziell messbare Vorzüge erkannt werden, die teilweise eine Konsequenz der bereits aufgeführten Vorteile sind. Dazu zählen eine kürzere Implementierungsphase, geringere Kosten für den Einsatz neuer Technologien, reduzierte Kosten über den gesamten Produktlebenszyklus, geringerer Zeit- und Personalbedarf für die Umsetzung geänderter Anforderungen, eine kürzere Time-To-Market und geringere Wartungskosten. Die Menge des manuell erstellten Codes kann reduziert werden, woraus sich unmittelbar niedrigere Aufwände ergeben. Da der noch immer manuell in der DSL zu erstellende Code eine höhere Informationsdichte und Komplexität hat, als in traditionellen Projekten, kann nicht direkt von der Menge des manuell erstellten Quelltextes auf die Kosten geschlossen werden [SVE07]. Jedoch vergrößert sich das Einsparungspotenzial durch Wiederverwendung des Generators in mehreren Projekten (vgl. [RSB+04]). Eine Betrachtung der Aufwandsverteilung in klassisch durchgeführten Projekten in Gegenüberstellung mit dem modellgetriebenen Vorgehen zeigt, dass auf Grund der eingesetzten Werkzeuge ein Großteil der notwendigen, manuellen Tätigkeiten durch automatisierte Transformationen und Codegenerierung entfällt. Es ergeben sich zwar zunächst größere Aufwände zu Beginn der Projekte, doch wird dies durch die Ersparnisse im weiteren Projektverlauf aufgewogen.

Es stellt sich nun die Frage, für welchen Zweck sich der Einsatz welcher Art von DSL lohnt. Wir betrachten diese Frage für die drei Arten von DSLs, die im vorgestellten Projekt zum Einsatz kamen:

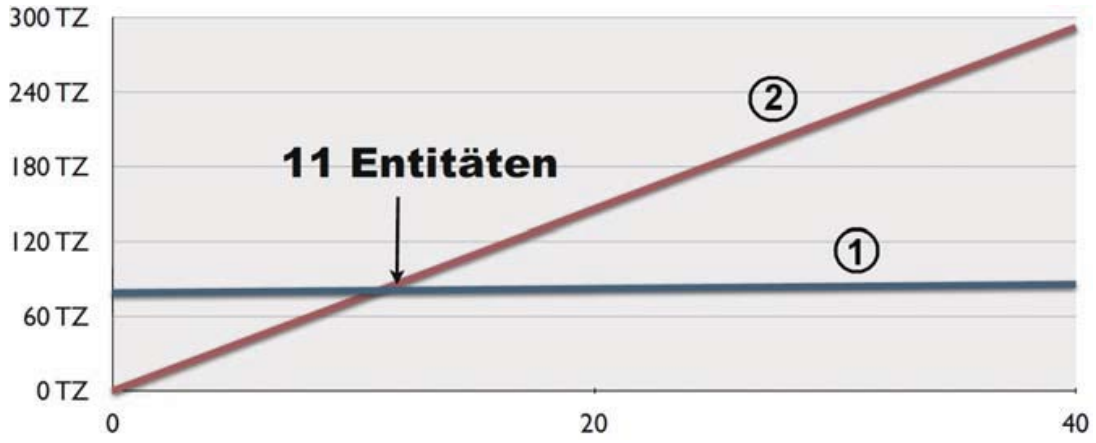


Abbildung 10: Vergleich der für das Datenmodell benötigten Programmzeilen mit DSL (1) und ohne DSL (2) in Abhängigkeit zu den implementierten Entitäten.

Externe DSL zur Beschreibung des Datenmodells Für die Entwicklung einer externen DSL ergeben sich zunächst initiale Kosten für die Implementierung dieser DSL, insbesondere auch für die Transformation in Programmcode. Dieser initiale Aufwand amortisiert sich erst bei einer bestimmten Größe des zu migrierenden (oder auch neu zu entwickelnden) Softwaresystems.

Für die Datenmodell-DSL aus Abschnitt 3 kann ein Vergleich des zu programmierenden Codes ohne bzw. mit DSL durchgeführt werden. Ohne DSL werden im Projekt Forms2Java durchschnittlich 7.000 Programmzeilen Java / JPA zur Programmierung des Zugriffs auf eine fachliche Entität benötigt. Für die Implementierung der DSL mit Xtext wurden 30.000 Zeilen Code programmiert, für die Implementierung der Transformation mit Xpand weitere 50.000 Zeilen Code. Für die Spezifikation des Datenmodells für eine Entität werden dann durchschnittlich jedoch nur noch 170 Zeilen je Entität benötigt. Rein rechnerisch, bei ausschließlicher Betrachtung der Programmzeilen, lohnt sich hier der Einsatz unserer Datenmodell-DSL ab 11 Entitäten, wie es auch in Abbildung 10 veranschaulicht wird. Die geringe Steigung der Linie 1 für die externe DSL ist in dieser Abbildung kaum zu erkennen. Durch das *Gepard*-System werden knapp 120 fachliche Entitäten verwaltet, die in gut 1700 Datenbanktabellen gespeichert werden.

Das Maß der Programmzeilen ist zum Vergleich des Aufwands natürlich nur sehr bedingt aussagekräftig, aber dieser Vergleich ist doch ein klares Indiz für die Wirtschaftlichkeit. Für die Betrachtung der initialen Kosten muss bei Bedarf auch der Aufwand für die Einarbeitung in neue Technologien (in unserem Projektkontext Xtext und XPand) beachtet werden, was in obigem Vergleich nicht berücksichtigt wird.

Interne DSL für die Eingabeprüfung Für eine interne DSL muss keine Entwicklungsumgebung und keine Transformation implementiert werden, diese stehen z.B. für Java schon mit Standard-IDEs, JUnit und dem Java Compiler zur Verfügung. Damit ergeben sich initial geringere Kosten. Auch die Größe des Programmcodes kann deutlich sinken, wie der Vergleich von Abbildung 5(a) mit Abbildung 5(b) zeigt.

Ein weiterer Vorteil besteht darin, dass die Entwickler keine vollständig neue Spra-

che erlernen müssen, wie es bei einer externen DSL nötig ist (auch wenn sich externe DSLs sinnvollerweise an existierenden Sprachen orientieren werden). Dies kann die Akzeptanz bei den Entwicklern erhöhen.

Ein Nachteil interner DSLs besteht darin, dass es je nach Möglichkeiten der Wirtsprache Einschränkungen für die Mächtigkeit der DSL gibt. Java bietet hierzu nur recht beschränkte Möglichkeiten. Boo [Boo09] ist beispielweise eine objektorientierte Programmiersprache, die besonderen Wert auf die Erweiterbarkeit der Sprache und ihres Compilers legt und somit bessere Möglichkeiten zur Entwicklung interner DSLs bietet als dies mit Java der Fall ist. Erwähnenswert sind bei Boo vor allem die sogenannten syntaktischen Makros, die die Sprache mit neuen Konstrukten erweitern können. Algorithmen können somit gekapselt und unter Benutzung eines neu eingeführten Schlüsselwortes wiederverwendet werden. Ein Problem ist bei Boo jedoch darin zu sehen, dass ein Softwareentwickler auf identischer Abstraktionsebene sowohl wiederverwendbare Code-Module spezifiziert, als auch als Sprachbestandteil zur Verfügung stellt.

Insgesamt empfehlen wir den Einsatz interner DSLs, wann immer dies ausreichend ist. Häufig wird jedoch wegen der beschränkten Möglichkeiten der eingesetzten Wirtsprachen der Einsatz externer DSLs erforderlich sein, um gute Produktivitätssteigerungen zu erreichen. Auch für eine interne DSL ist der Aufwand diese zu entwerfen nicht zu unterschätzen.

Externe DSL für die Programmierung der Eingabemasken Die obigen Aussagen zur Datenmodell-DSL treffen im Prinzip auch auf die GUI-DSL zu. Für die GUI-DSL wurde statt eines Xtext-Editors ein GUI-Editor entwickelt, siehe auch die Übersichtsdarstellung in Abbildung 2. Dieser GUI-Editor liefert keine neuen Möglichkeiten in der Programmierung. Vielmehr geht es hierbei darum, die Akzeptanz bei den Anwendungsprogrammierern durch eine angemessen zu bedienende Sicht auf die GUI-DSL zu erhöhen. Der Aufwand für die Konstruktion des GUI-Editors ist, auch mit dem Eclipse GMF, nicht zu unterschätzen; kann aber für die Akzeptanz bei den Mitarbeitern, und damit den Erfolg eines Migrationsprojektes, wie in diesem Papier beschrieben, von großer Bedeutung sein.

7 Verwandte Arbeiten

Die modellgetriebene Softwareentwicklung bildet die Grundlage für eine effiziente Verwendung von DSLs. Mit ihrer Hilfe können beispielsweise aus abstrakten Systemmodellen konkrete Systemartefakte generiert werden. Auf dieser Basis existieren verschiedene Vorgehensweisen, wie DSLs gestaltet und deren Tauglichkeit für einen konkreten Projektkontext evaluiert werden können. Die praktische Anwendbarkeit modellgetriebener Techniken und DSLs wurde hierbei in einer Reihe von Fallstudien empirisch untersucht. Für den speziellen Einsatzzweck der Migration von Softwaresystemen ist darüber hinaus der Aspekt der Transformation von Modellen von besonderer Bedeutung. Des weiteren existieren einige Vorgehensmodelle und Muster für die Migration, sowie auch für Anwendungsfälle, bei denen der Fokus auf der Evolution, Transformation und Migration von Datenbankan-

wendungen liegt. Dieser Abschnitt gibt einen Überblick über die vorgenannten Themenbereiche.

Das von uns präsentierte Migrationsprojekt basiert auf Methoden der modellgetriebenen Softwareentwicklung. Mehrere Arbeiten bieten hierzu eine Einführung, beispielsweise [BG05, Sch06, SVE07]. Hierbei wird die Anwendbarkeit an überschaubaren Systemen mit Artefakten, die durch sich häufig wiederholende Strukturen geprägt sind, verdeutlicht. Potenzielle Herausforderungen und Probleme im Hinblick auf die Entwicklung von komplexeren Softwaresystemen, wie etwa das im Rahmen des Projektes *Forms2Java* migrierte Altsystem *Gepard* der APG Affichage, untersucht [FR07]. Die Analyse wird hierbei von den Autoren entlang einer Kategorisierung der Herausforderungen in die drei Bereiche der (1) Modellierungssprachen, (2) Trennung der Belange in verschiedene modellrelevante Sichten und (3) Manipulation und Management von Modellbestandteilen, geführt.

Diverse Fallstudien evaluieren den Einsatz von modellgetriebenen Methoden und Techniken. Eine Laborstudie führten etwa die Autoren in [WSG05] durch, um das Potenzial von modellgetriebenen Methoden bei der Vereinfachung der Entwicklung von autonomen Systemen, die Enterprise Java Beans einsetzen, zu untersuchen. Darüber hinaus evaluieren mehrere Studien das Potential auch im industriellen Kontext [BLW05, Sta06].

Modellgetriebene Methoden kommen, wie auch im Rahmen des von uns beschriebenen Migrationsprojektes, häufig in Verbindung mit DSLs zum Einsatz. Eine grundsätzliche Unterscheidung erfolgt durch die Einordnung in interne und externe DSLs [Fow09], bei *Forms2Java* kamen beide Varianten zum Einsatz. Eine weitergehende Klassifizierung findet sich in [LJJ07]. Die Autoren bilden verschiedene DSL-Varianten anhand von gemeinsamen Merkmalen bezüglich des Einflusses auf die Sprache selbst, die potenziellen Transformationen, die Werkzeuge und die Entwicklungsprozesse. Muster zur Erstellung von DSLs werden in [Spi01] behandelt. Hierbei orientiert sich die Klassifizierung vornehmlich an dem Verhältnis einer DSL zu einer etwaigen Basis- bzw. Wirtsprache. Kriterien, die für die Erstellung von DSLs sprechen, untersucht [MHS05]. Die Autoren liefern darüber hinaus einen Literaturüberblick zu Kriterien, die für die Erstellung von Mustern relevant sind. Hierbei erfolgt eine Unterscheidung von Mustern nach Phasen im Entwicklungsprozess von DSLs. Die Entscheidungsphase liefert die Entscheidung für oder gegen die Erstellung einer eigenen DSL. In der Analysephase wird weitergehendes Wissen über die Problemdomäne erworben. Die Entwurfsphase beschäftigt sich mit dem Entwurf der DSL, die danach in der Implementierungsphase realisiert und in der Deploymentphase auf einer Zielplattform eingesetzt wird. Eine weitere Klassifikation von Ansätzen zur Entwicklung von DSLs findet sich in [Wil01].

Als weitere Grundlage für die modellgetriebene Migration von Softwaresystemen ist die Transformation von Modellen anzusehen, da ein Modell des Altsystems auf ein Modell des Zielsystems abgebildet werden muss. [SK03] klassifiziert Ansätze zur Modelltransformation hinsichtlich ihrer Eigenschaft, Modelle direkt manipulieren, exportieren und die Transformationsregeln explizieren zu können. Des Weiteren werden als wünschenswerte Merkmale von Sprachen zur Transformation von Modellen verschiedene Eigenschaften herausgearbeitet. Darunter beispielsweise die Fähigkeit, eventuell existierende Vorbedingungen, unter denen eine Transformation sinnvoll angewendet werden kann, zu beschreiben. Eine Taxonomie von Sprachen zur Modelltransformation beschreibt [MG06]. Sie soll unter anderem als Entscheidungsgrundlage zur Auswahl geeigneter Verfahren dienen.

Herausforderungen und Probleme bei der Evolution von Software fasst [vDVW07] zusammen. Für das Teilgebiet der Migration liefert [BLW+97] einen Überblick über gängige Vorgehensmodelle und Muster, wie das im Rahmen dieser Arbeit für den speziellen Projektkontext diskutierte Dublo-Migrationsmuster. Der fehleranfälligen Umstellung von einem Altsystem auf das komplett migrierte System in einem Schritt („Big-Bang“, oder auch „Cold Turkey“ Ansatz genannt) stellen die Autoren in [BS95] ihren Ansatz „Chicken Little“ entgegen, wobei eine inkrementelle Migration hervorgehoben wird. Ein Nachteil zeigt sich hierbei jedoch bei der Wartung und dem koordinierten Zugriff auf parallel gepflegte alte und neue Datenbestände. Diesen Aspekt adressiert die Methode „Butterfly“, die die zu der Koordinierung benötigten Gateways entfernt [WLB+97]. Ein weiterer Ansatz wird in [WSV05] beschrieben, der für die Migration unter anderem Techniken der dynamischen Programmanalyse einsetzt. Erfahrungsberichte zu dem Einsatz von modellgetriebenen Techniken im Kontext der Migration finden sich etwa in [FBB+07, ZG04].

Auch für die Evolution und Migration von Datenbanken existieren zahlreiche Veröffentlichungen. [BGD97] untersucht die Migration von relationalen Datenbankschemata auf objekt-orientierte Datenbanksysteme. Eine weitere Analyse der Evolution und Integration von Schemata findet sich bei [Cla94]. [MP03] legt einen Fokus auf umkehrbare Schema-Transformationsregeln. Den Einfluss von heterogenen Architekturen bei Datenbanken auf die Transformation von Schemata wird bei [MP06] analysiert. Das Mapping zwischen unterschiedlichen Schemata wird mittels Schema Matching durchgeführt. [DSDR07] präsentiert beispielsweise für diesen Zweck den Ansatz „QuickMig“. Eine Übersicht über bestehende Ansätze für das Schema Matching findet sich in [RB01].

8 Zusammenfassung und Ausblick

Anhand eines nicht-trivialen Migrationprojektes diskutierten wir den Einsatz von DSLs zur Migration von Datenbankanwendungen. Der primäre Vorteil einer externen DSL besteht in der Möglichkeit, eine optimale Abstraktion für eine Anwendungsdomäne bieten zu können. Der primäre Vorteil einer internen DSL besteht darin, dass keine vollständig neue Sprache entworfen und implementiert werden muss. Der primäre Vorteil des GUI-Editors ist in der Akzeptanz einer angemessenen Werkzeugunterstützung durch die Anwendungsprogrammierer zu sehen.

Der grundsätzliche Ansatz, mit domänenspezifischen Sprachen Modelle aus Altsystemen zu extrahieren, um diese zu modernisieren, wird beispielsweise auch in [IM09] vorgeschlagen. Eine Frage, die immer für derartige Projekte gestellt werden muss, besteht darin, ab wann sich der Einsatz von DSLs lohnt. Es stellt sich insbesondere die Frage, ob sich die Entwicklung von spezifischen DSLs schon für ein einzelnes Projekt lohnt, oder ob dies erst sinnvoll ist, wenn beispielsweise eine Produktlinie aufgebaut werden soll. Unsere Antwort dazu ist, dass es gar nicht immer sinnvoll ist zu versuchen, eine DSL zu entwickeln, die über viele Produkte hinweg genutzt werden kann. Der Grund dafür besteht darin, dass eine derartige DSL (üblicherweise) deutlich komplexer werden würde, als dies für ein einzelnes Projekt erforderlich wäre.

Um den Einsatz von DSLs schon für einzelne Projekte wirtschaftlich sinnvoll gestalten zu können, kommt dann der effektiven und effizienten Werkzeugunterstützung zur Implementierung der DSLs eine entscheidende Rolle zu. Mit den diversen Werkzeugen im Umfeld des Eclipse-Projektes stehen inzwischen leistungsfähige Werkzeuge zur Verfügung, um externe DSLs effizient implementieren zu können. Es ist prinzipiell mit diesen oder ähnlichen Werkzeugen möglich, eine DSL für mehrere Projekte zu nutzen, indem diese DSL für jedes Projekt individuell an dessen spezifische Bedürfnisse angepasst werden kann. Das Ziel sollte aus unserer Sicht dabei nicht sein, *eine* DSL für viele Projekte zu entwickeln, die schnell zu komplex werden könnte, sondern eine *Familie* von (kleinen) DSLs zu entwickeln.

Ein noch offenes Problem besteht dann darin, wie eine gute Wiederverwendung für die Implementierung der DSLs einer Familie ermöglicht werden kann. Dieser Frage widmen wir uns gegenwärtig im BMBF-geförderten Projekt Xbase.[12] Ziel des Projektes Xbase ist es, den anfänglichen Aufwand zur Implementierung der Infrastruktur für eine DSL erheblich zu reduzieren. Durch diese Maßnahme soll die modellbasierte Softwareentwicklung auch schon bei kleineren Projekten kosteneffizient eingesetzt werden können. Dabei werden immer wiederkehrende Aspekte der DSLs in Xbase allgemeingültig, anpassbar und einfach wiederverwendbar implementiert. Der Einsatz von Xbase verspricht hier weiteres Kosteneinsparungspotenzial, gegenüber der jetzigen Situation mit dem Einsatz modellgetriebener Entwicklungsmethoden.

Literaturverzeichnis

[BG05] Sami Beydeda und Volker Gruhn. *Model-Driven Software Development*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[BGD97] Andreas Behm, Andreas Geppert und Klaus R. Dittrich. On the Migration of Relational Schemas and Data to Object-OrientedDatabase Systems. Bericht, University of Zurich, 1997.

[BLW+97] J. Bisbal, D. Lawless, Bing Wu, J. Grimson, V. Wade, R. Richardson und D. O'Sullivan. An overview of legacy information system migration. Seiten 529 –530, dec. 1997.

[BLW05] Paul Baker, Shiou Loh und Frank Weil. Model-Driven Engineering in a Large Industrial Context - Motorola Case Study. In Lionel C. Briand und Clay Williams, Hrsg., *MoDELS*, Jgg. 3713 of *Lecture Notes in Computer Science*, Seiten 476–491. Springer, 2005.

[Boo09] The programming language Boo. `http://boo.codehaus.org/`, 2009.

[BS95] M.L. Brodie und M. Stonebraker. *Migrating Legacy Systems – Gateways, Interfaces and The Incremental Approach*. Morgan Kaufmann, San Francisco, CA, USA, 1995.

[Cla94] Stewart M. Clamen. Schema evolution and integration. *Distributed and Parallel Databases*, 2:101–126, 1994.

[DSDR07] Christian Drumm, Matthias Schmitt, Hong-Hai Do und Erhard Rahm. Quickmig: automatic schema matching for data migration projects. In *CIKM '07: Proceedings of the*

[12] `https://kosse-sh.de/projekte/xbase/`

sixteenth ACM conference on Conference on information and knowledge management, Seiten 107–116, New York, NY, USA, 2007. ACM.

[FBB+07] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas und Jean-Marc Jézéquel. Model-Driven Engineering for Software Migration in a Large Industrial Context. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt und Frank Weil, Hrsg., *MoDELS*, Jgg. 4735 of *Lecture Notes in Computer Science*, Seiten 482–497. Springer, 2007.

[Fow09] M. Fowler. A Pedagogical Framework for Domain-Specific Languages. *Software, IEEE*, 26(4):13 –14, jul. 2009.

[FR07] Robert France und Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *FOSE '07: 2007 Future of Software Engineering*, Seiten 37–54, Washington, DC, USA, 2007. IEEE Computer Society.

[HBG+08] Wilhelm Hasselbring, Achim Büdenbender, Stefan Grasmann, Stefan Krieghoff und Joachim Marz. Muster zur Migration betrieblicher Informationssysteme. In K. Herrmann und B. Brügge, Hrsg., *Tagungsband Software Engineering 2008*, Jgg. 121 of *Lecture Notes in Informatics*, Seiten 80–84, München, Februar 2008. Köllen Druck+Verlag.

[HKRS08] Wilhelm Hasselbring, Stefan Krieghoff, Ralf Reussner und Niels Streekmann. Migration der Architektur von Altsystemen. In *Handbuch der Software-Architektur*, Seiten 213–222. dpunkt.verlag, 2. Auflage, 2008.

[HRJ+04] Wilhelm Hasselbring, Ralf Reussner, Holger Jaekel, Jürgen Schlegelmilch, Thorsten Teschke und Stefan Krieghoff. The Dublo Architecture Pattern for Smooth Migration of Business Information Systems. In *Proc. 26th International Conference on Software Engeneering (ICSE 2004)*, Seiten 117–126, Edinburgh, Scotland, UK, Mai 2004.

[IM09] Javier Luis Canovas Izquierdo und Jesus Garcia Molina. A Domain Specific Language for Extracting Models in Software Modernization. In *ECMDA-FA 2009*, Jgg. 5562 of *LNCS*, Seiten 82–97. Springer-Verlag, 2009.

[LJJ07] B. Langlois, C.E. Jitia und E. Jouenne. DSL classification. In *DSM '07: Proceedings of the Seventh OOPSLA Workshop on Domain-Specific Modeling*, Jgg. DSM '07, Seiten 28–38, 2007.

[MG06] Tom Mens und Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125 – 142, 2006. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005).

[MHS05] Marjan Mernik, Jan Heering und Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[MP03] P. McBrien und A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, Seiten 227 – 238, mar. 2003.

[MP06] P. McBrien und A. Poulovassilis. Schema Evolution in Heterogeneous Database Architectures, A Schema Transformation Approach. In Anne Pidduck, M. Ozsu, John Mylopoulos und Carson Woo, Hrsg., *Advanced Information Systems Engineering*, Jgg. 2348 of *Lecture Notes in Computer Science*, Seiten 484–499. Springer Berlin / Heidelberg, 2006.

[RB01] Erhard Rahm und Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10:334–350, 2001.

[RSB+04] Dr. G. Rackl, Dr. U. Sommer, Dr. K. Beschorner, Heinz Koßler und Adam Bien. Komponentenbasierte Entwicklung auf Basis der Model Driven Architecture. *OBJEKTspektrum*, (5):43–48, September / Oktober 2004.

[Sch06] Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer Society: Computer*, 39:25–31, 2006.

[SK03] Shane Sendall und Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Softw.*, 20(5):42–45, 2003.

[SPC09] Ben Shneiderman, Catherine Plaisant und Maxine Cohen. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Pearson Higher Education, 5. Auflage, 2009.

[Spi01] Diomidis Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99, 2001.

[Sta06] Miroslaw Staron. Adopting Model Driven Software Development in Industry - A Case Study at Two Companies. In Oscar Nierstrasz, Jon Whittle, David Harel und Gianna Reggio, Hrsg., *Model Driven Engineering Languages and Systems*, Jgg. 4199 of *Lecture Notes in Computer Science*, Seiten 57–72. Springer Berlin / Heidelberg, 2006.

[SVE07] Thomas Stahl, Markus Völter und Sven Efftinge. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt.verlag, 2. Auflage, Mai 2007.

[vDVW07] Arie van Deursen, Eelco Visser und Jos Warmer. Model-Driven Software Evolution: A Research Agenda. In D. Tamzalit, Hrsg., *CSMR Workshop on Model-Driven Software Evolution (MoDSE 2007)*, Seiten 41–49, Amsterdam, The Netherlands, March 2007.

[Wil01] D.S. Wile. Supporting the DSL spectrum. *Journal of Computing and Information Technology*, 9(4):263–287, 2001.

[WLB+97] Bing Wu, D. Lawless, J. Bisbal, R. Richardson, J. Grimson, V. Wade und D. O'Sullivan. The Butterfly Methodology: a gateway-free approach for migrating legacy information systems. Seiten 200 –205, sep. 1997.

[WSG05] Jules White, Douglas Schmidt und Aniruddha Gokhale. Simplifying Autonomic Enterprise Java Bean Applications Via Model-Driven Development: A Case Study. In Lionel Briand und Clay Williams, Hrsg., *Model Driven Engineering Languages and Systems*, Jgg. 3713 of *Lecture Notes in Computer Science*, Seiten 601–615. Springer Berlin / Heidelberg, 2005.

[WSV05] Lei Wu, H. Sahraoui und P. Valtchev. Coping with legacy system migration complexity. In *Proceedings. 10th IEEE International Conference on Engineering of Complex Computer Systems, 2005. ICECCS 2005.*, Seiten 600 – 609, jun. 2005.

[ZG04] J. Zhang und J. Gray. Legacy System Evolution through Model-Driven Program Transformation. In *EDOC workshop on Model-Driven Evolution of Legacy Systems (MELS)*, Monterey, USA, September 2004.