DSLs to fully generate Business Applications

Daniel Stieger, Matthias Farwick, Berthold Agreiter, Wolfgang Messner

Introduction

One of our customers, a major Austrian retailer, approached us with specific requirements. He was looking for a viable way to accompany existing, well-adjusted business processes with an easy-to-use application. With some support from the IT department, process owners should be enabled to design their application on their own. These are pretty standard requirements for individualized business applications involving concepts like domain entities, workflows, views for CRUD operations and transactions. However, our customer already had some good experience with fat-client architectures in the past and therefore demanded to employ a database centric architecture without relying on any application server. In order to ensure long term maintainability, a migration path to a more centralized architecture had to be suggested additionally.

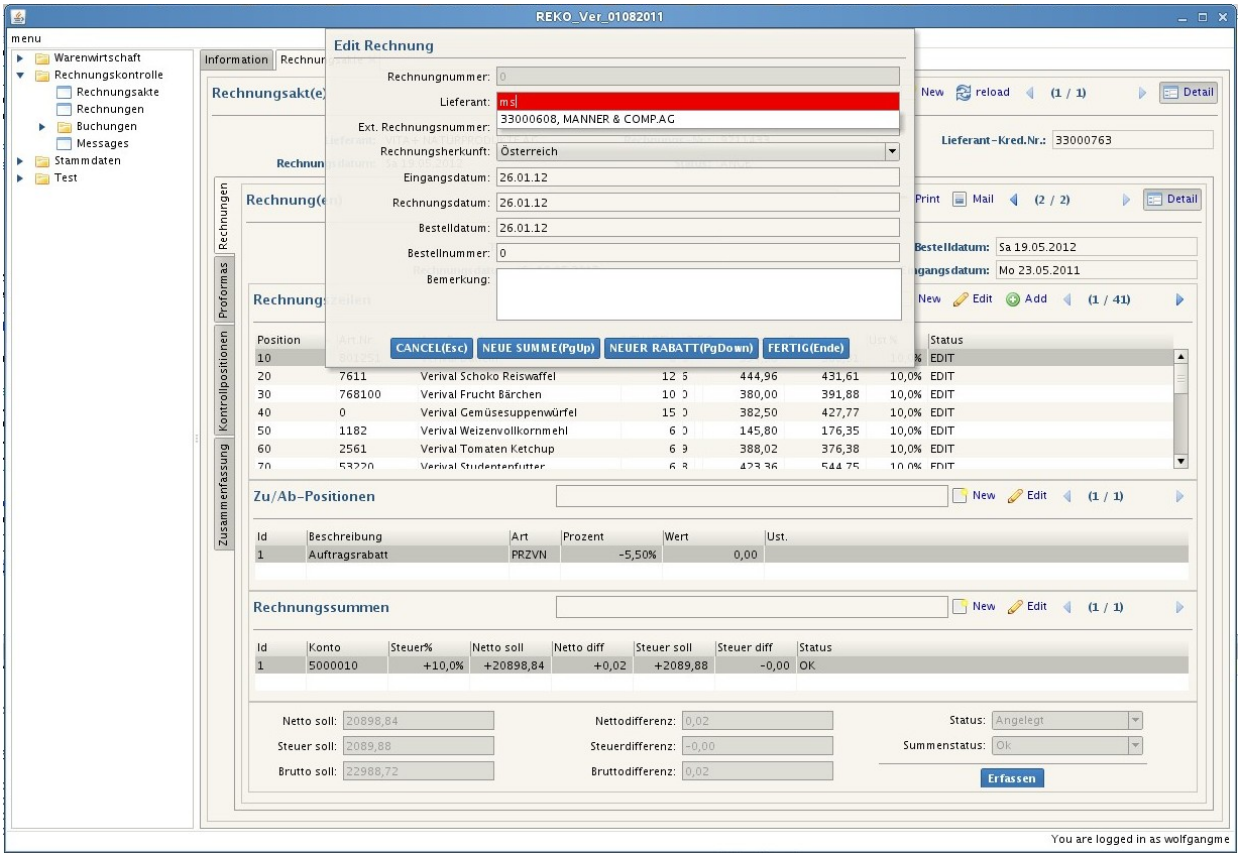

Fig 1. Example of a fully generated business application that was created with our domain specific languages.

Compared to existing applications at our customers site, management aimed at improving the quality of newly designed applications as well as decreasing costs of long term service efforts. Best practice suggests that this could be achieved by establishing design principles and patterns that foster software architecture standardization. Creating applications that consistently follow the principles of a good design (typically defined at the project inception) is known to be a difficult task. The patterns that make up a good design are often not part of the programming language itself. Even if appropriate patterns could be identified and captured in toolkits, participating developers oftentimes follow their own path, ignoring defined guidelines and thus impairing standardization efforts.

In order to meet our customers demands we introduced model driven development (MDD). MDD focuses on creating and exploiting domain models (that is, abstract representations of the knowledge and activities that govern a particular application domain), rather than relying on generic technical programming concepts. All requirements are captured in a logical model of the business application. The logical model can be formulated in a Domain-Specific-Language (DSL). Afterwards, the logical model can be translated into a common programming language by applying appropriate patterns (code generation). MDD maximizes compatibility between systems (via reuse of standardized models), simplifies the process of design (via models of recurring design patterns in the application domain), and promotes communication between individuals and teams working on the system, by using domain concepts as the unit of discussion.

In this article we will show how we applied model driven development with the JetBrains Meta Programming System language workbench in order to fulfill the requirements of our customer.

Extending DSLs - language extension

After deciding on Java technology as the implementation environment, the domain was isolated in detail and relevant patterns were identified. Two DSLs were devised to model business applications, namely Objectflow (data handling + logic) and Forms (user interface). As an environment to implement the DSLs, the JetBrains MPS workbench was chosen. The workbench brings along unique capabilities regarding language extension and language composition. For example, our two DSLs extend standard Java in a way that important Java concepts like an “if statement” or expressions - which are of most importance to formulate logic - can be used within our DSL. Thus, our DSLs cover major patterns and terms used in the domain, whereas Java still adds some flexibility where needed. There is no necessity to write additional Java code in order to generate the final business application. Users applying this “new Java” can focus on implementing the domain logic, rather than spending time on reading guidelines, creating boiler plate code to follow architectural patterns or coding the same user interface controllers again and again.

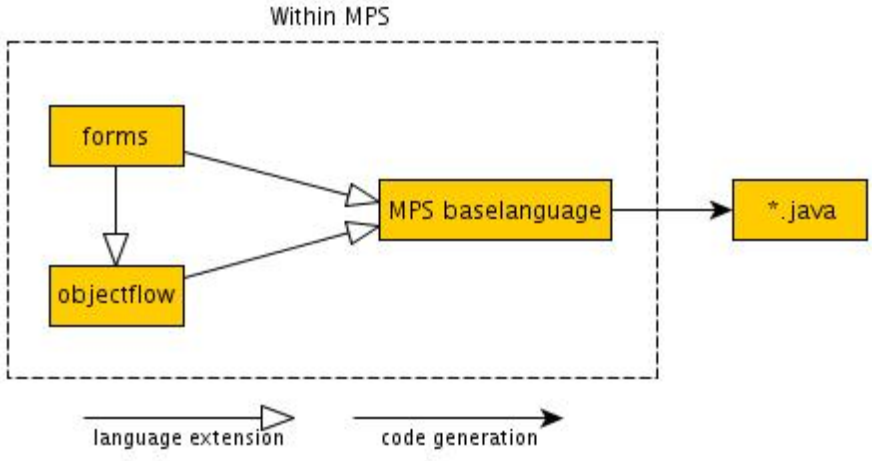

Figure 2: the relation between Forms, Objectflow and the MPS java baselanguage

Figure 2 points out the language extension hierarchy of the employed DSLs described in this article. Forms, the language to capture user interfaces logically, and Objectflow, which introduces services and data objects, extend the MPS java BaseLanguage. Forms, as well as Objectflow, thus receive additional flexibility of java concepts. All high-level concepts will eventually be transformed into pure Java code.

DSLs to model business applications

Instead of discussing the idea of language-extension in more detail, we are going to give you a brief glimpse from a practical perspective by pointing out some major aspects of our business-application DSLs: Objectflow and Forms. Both DSLs were built with JetBrains Meta Programming System. We will focus on Objectflow first.

Objectflow

Objectflow builds on the ideas of Eric Evans´ paradigm of domain-driven design (DDD). Any data (entities or value objects in DDD terms) can be modelled as Objectflow business objects with various attributes and references. For example, an invoice with invoice positions is modelled as a business object invoice containing a collection of invoice position business objects. The invoice becomes a so called “root aggregate”, when functionality or operations are added in a dedicated invoice service. Thus, a root aggregate is a cluster of associated objects that we treat as a unit for the purpose of data changes. All necessary functionality has to be put into a service and in turn, the service has to be related to a root aggregate. Thus, although Objectflow services can be written in plain java, the structure and building blocks of services are standardized. Particularly this standardization results in more transparency and higher code quality. As can be seen in Figure 3, the names of operations are standardized. Objectflow also indicates that create, save and update operations are mandatory for an invoice.

```
// objectflow service component
service component InvoiceService for businessobject Invoice {


  <no fields>

  public InvoiceService() {
    <no statements>
  }
```

```
collection support for positions {
  InvoicePosition createPosition_default(Invoice invoice) {
    InvoicePosition pos = new InvoicePosition();
    pos.invoice = invoice;
    pos.amount = 0.0d;
    pos.tax = 0.0d;
    return pos;
  }

  void attachPosition(Invoice invoice, InvoicePosition position) {
    invoice.positions.add(position);
  }

  InvoicePosition detachPosition(Invoice invoice, InvoicePosition position) {
    invoice.positions.remove(position);
    return position;
  }

}
```

```
businessobject support for Invoice {
  Invoice createInvoice_default(<< ... >>) {
    Invoice invoice = new Invoice();
    invoice.createdAt = now in (default timezone);
    return invoice;
  }

  void saveInvoice(Invoice invoice) {
    transaction {
      save invoice
      save invoice.positions
    } finally {
      <no statements>
    }
  }

  void updateInvoice(Invoice invoice) {
    invoice.totalAmount = 0.0d;
    invoice.totalTax = 0.0d;
    invoice.positions.forEach({~it =>
      invoice.totalAmount = invoice.totalAmount + it.amount;
      invoice.totalTax = invoice.totalTax + it.tax;

    });
```

Figure 3. Example of invoice service: The methods createXxx_default(), saveXxx() or updateXxx() are so called operations in Objectflow. They are mandatory for an aggregate business object and can not be renamed or changed in any way.

Objectflow also takes advantage of a very strong collection language from MPS, which can be applied freely in services. Building unions of invoices or summing up a collection of invoices are central operations in business applications and should therefore be supported with a very clear and handy syntax. The collection language employed with Objectflow provides the means to

build expressive queries using closures - in a way similar to Microsoft´s LINQ.

The very same high level collection language is also used to handle database queries. Objectflow allows the developer to create a so called “repository” containing various queries for business objects. Additionally to Java code, SQL queries as well as their mappings (database table fields to business object properties) are generated automatically. Thus the developer is no longer challenged by technical implementation details like mapping or database querying. Figure 4 illustrates a findInvoiceByDate() method, returning a sorted list of Invoices with specified date. The invioce positions objects are also loaded with a join.

```
// objectflow repository
repository  InvoiceRepo{

  repository for Invoice, InvoicePosition, Company


    //
  public list<Invoice> findInvoiceByDate(datetime date);
    query Invoice.where({~it => it.createdAt == date; }).sortBy({~it => it.createdAt; }, desc)

    Technical options for query:

    join collection <InvoicePosition> positions  in bO  Invoice
      <<no queryprop>>


}
```

Figure 4. Example of database query

Objectflow actively supports test driven development. Developers can create a new executable testsuit - tailored to test business logic- with just one click. The testsuit contains pretest-data and posttest-data, both described in simple Json-notation. Pretest-data contains necessary data saved in business objects, which gets built and initialized before running a test. Posttest-data contains expected outcomes which are compared with actual data from the test by posttest-checker methods. To test a service, one just needs to shovel some pretest-data into a service and check return objects after execution with expected results. In our initial case of invoices, one can test a booking services with a pretest invoice and posttest booking lines.

```
// objectflow testSuit
testsuit  InvoiceServiceTest {

pretest data:
protected Invoice myInvoice = Invoice {
  createdAt: now in (default timezone);
  description: "test invoice";
  totalAmount: 100.0d;
  totalTax: 10.0d;
  company: Company {
    address: "Andreas-Hofer-Strasse 15";
    city: "A6020 Innsbruck";
    name: "Modellwerkstatt";
    };
  positions: InvoicePosition {
    amount: 50.0d;
    tax: 5.0d;
    articleNo: 4712;
    };
  positions: InvoicePosition {
    amount: 50.0d;
    tax: 5.0d;
    articleNo: 4713;
    };
  };

posttest data:
protected AccountingEntry entry = AccountingEntry {
  amount: 100.0d;
  tax: 50.0d;
  };

posttest checkers:
public static boolean AccountingEntryOK(AccountingEntry entry, AccountingEntry entryExpected) {
  if (entry.amount == entryExpected.amount && entry.tax == entryExpected.tax) { return true; }
  false;
}


services under test:
<< ... >>


usecases:  (report always with System.err. )
  usecase CheckInvoiceService {
    <no preTestManipulation>

    sequence CreateInvoice {
      // test code here
    }
  }
}
```

Figure 5. Example of test suit

Eventually, Objectflow generates POJOs for business objects and transForms services, repositories and tests to usual Java classes, handled by a small service middleware written with the Google guice injection framework. Additionally, XML descriptions for the java MyBatis database mapper are generated automatically. Generated code is only dependent on very basic frameworks and thus conveys a sense of simplicity.

Forms

Developing good UIs for commercial business applications is a non-trivial task. It requires effort to build and maintain them in order to assure ease of use and high enduser-productivity. However, business applications often have similar interfaces, mainly based on Forms. Reasoning about best practice UI-patterns for information systems and combining them with architectural considerations led us to the Forms-DSL.

The Forms-DSL is built around a logical Model for (1) views and (2) actions. Views are concepts to visualize data - in our case Objectflow's business-objects. Views also capture elementary UI specifications such as formatting of numbers (e.g. money or amounts) and ordering of data. Requirements are specified in a clear and neat declarative style, without the necessity to be aware of any technical details. For example: The MultiLineEdit is capable of visualizing a list of business-objects (e.g. invoices) in a tabular manner. Beside this so called "list view definition" the MultiLineEdit also declares a detail-view for a single business-object, the "form view definition". As pointed out in the picture, detail-views can in turn contain further MultiLineEdit specifications.

```
MultiLineEdit RechnungView bindsto Rechnung
label: Rechnung(en)  minWidth: 500  selectFirstRow: true

list view definition
  id with label: ID numberFormat: <no numformat> width 50,
  art.<no pathRef> with label: Typ numberFormat: <no numformat> width 100,
  belegNummer with label: Beleg Nr. numberFormat: 0 width 100,
  extBelegNummer with label: Ext. Beleg Nr. numberFormat: <no numformat> width 100,
  valutaDatum with label: Eing. Datum numberFormat: <no numformat> width 100,
  rechnunsgDatum with label: Rech. Datum numberFormat: <no numformat> width 100,
  bruttoWertSoll with label: BruttoSoll numberFormat: 0.00;-0.00 width 100,
  diffNettoWert with label: Diff. Netto numberFormat: 0.00;-0.00 width 100,
  status.<no pathRef> with label: Status numberFormat: <no numformat> width 100,

form view definition
  HEAD
  2 column layout
      art.<no pathRef> with label: Typ numberFormat: <no numformat> width 200,
      bestellDatum with label: Bestelldatum numberFormat: <no numformat> width 200,
      rechnunsgDatum with label: Rechnungsdatum numberFormat: <no numformat> width 200,
      valutaDatum with label: Eingangsdatum numberFormat: <no numformat> width 200,


  BODY
  LinearLayout:
    "-": RechZeilenRechnungView load with rechZeilen  (boundObject: RechZeile)  height 3* (show form view true )
    "-": ZuAbZeileView load with zuAbZeilen  (boundObject: ZuAbZeile)  height 110 (show form view true )
    "-": RechnungSummenView load with steuerZeilen  (boundObject: SteuerZeile)  height 110 (show form view true )
    "-": RechnungWorkflow load with <no propertyPath>  (boundObject: Rechnung)  height 90



  FOOTER
  << ... >>
```

Figure 6. Definition of a MultiLineEdit with the Forms-DSL.

While views like the MultiLineEdit, the InputWindow or the Wizzard state the visualization possibilities, Actions declare user interactions within the application. E.g., if we want the user to be able to add an invoice-position when viewing the invoice, we have to attach a “new invoice-position” action the the corresponding MultiLineEdit. An appropriate action would look like follows:

```
ACTION NEW_INVOICEPOSITION
shortDescription: “New InvoicePosition”
longDescription: “Add new InvoicePosition to Invoice”
image: new.png                              // in case it s displayed as btn.
hotkey: CRTL-L                              // global hotkey
enabled when: AuthService.hasRole(REKO_NEW_INVOICEPOS) &&
              SELECTED(Invoice).canAddNewInvoicePos()
preconditions:
action implementation:
  SimpleViewAction
  view: InvoicePositionForm form
  on entry: {~form =>
            form.setArticleScope(InvoiceService.availableArticles(SELECTED(Invoice))); }
  load with:{ => InvoiceService.createPosition_default(SELECTED(Invoice)); }
  save: {~position => InvoiceService.attach(position)); }
  cancel:
```

The action specifies long and short labels, an image and a global hotkey. It is interesting to note that the action definition is not related to its representation in the application. The action can be represented as button somewhere or as an entry in an action list in the top right corner of an application. Furthermore, the “enabled when” clause states that a “new invoice-position” action can be executed anytime when a particular invoice is selected, the invoice is in a state to add positions and the user has sufficient rights. The clause itself is formulated as standard Java expression. Thus it is clearly indicated to a developer that a clause is mandatory in an action concept by additionally exploiting the expressive power of Java. The very same expressive power of Java is allowed when specifying the implementation of the action in a SimpleViewAction. Initialization as well as the concluding save operation can be formulated inside a closure where the outer context (in this case the form and the position) can be accessed by parameters.

Eventually, the views and actions need to be rendered at some point. Currently we generate code for the open source rich internet application framework “Apache Pivot” (see figure 1) Apache Pivot applications can run inside a browser as applet or as stand-alone application on desktops.

Conclusion

Business applications are oftentimes very similar in their basic structure. In principle, the potential for standardization is very high. We found the Java environment as a profound choice to build business applications in our domain, since numerous tools and libraries are already

available. However, the Java programming language is very technical in nature and thus not easily readable for non-specialists. Common patterns and concepts used in the domain cannot easily be integrated in libraries or tools, impeding standardization efforts. Especially good architecture and the application of domain patterns are the key enablers of superior software development and should therefore be standardized. Domain-Specific-Languages (DSLs) and Model Driven Development can be seen as new paths offering solutions to these shortcomings of the Java programming language. As we have pointed out in our article, functions of business applications can be captured in logical models, hiding technical implementation details of Java. Jetbrains MPS provides ways to build these logical models in a timely manner by allowing concepts of Java to be used directly in the models, and in turn, enhancing the expressiveness of the models. The logical models also lead to additional independence from the Java technology used. Decisions on libraries or on the architecture (e.g. three-tier, fat-client, or web) are isolated in code-generators, which can be exchanged with limited efforts.

Any feedback or questions are welcome. You can contact Daniel Stieger by e-mail (daniel.stieger@modellwerkstatt.org).