Create database applications fast

Documentation

Create database applications fast

Version 1.1 - Release_1_0_4_stable_rc1_branch

© 2000-2015 Lothar Behrens

PDF version $Revision: 1.11.2.10 $
ODT version $Revision: 1.12.2.12 $

Table of contents

Abstract

lbDMF is a software development framework. It's main application is a prototyping software that helps you to faster get database related applications out of a formal definition using UML.
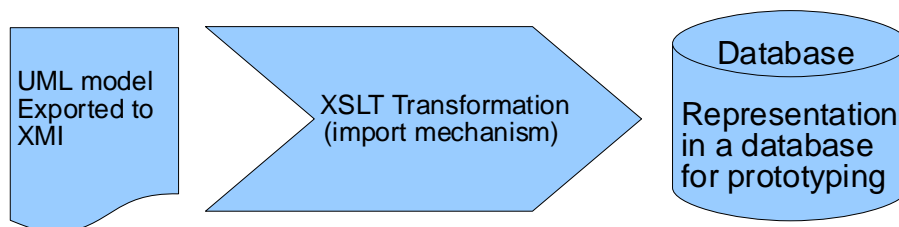
The goal of this software is to help with your ideas in a way that let you try out these ideas without prior coding, without making a decision toward what language is used for the implementation, with the ability to throw away a prototype because it was not expensive to get it running.

Using model driven development methods, it is possible to keep your formal model in a small and precise form. You will have the ability to make your model more precise using features provided by UML.

Introduction

Why is a prototype a good starting point? First, you can express your ideas without starting to code it in a programming language, because you are unsure what programming language should be used. The prototype can be used to demonstrate the ideas and can act as a discussion base for what it should do. Sample data can be entered to demonstrate the use cases or be the base for testing purposes. It is also more than a simple document describing the requirements, it is a tool to help collect requirements.

When you start using this software, it will present a management application for your designs, that you can import in that application. As stated before, the main format for your model entry will be UML, thus there is an import mechanism for UML. Exactly it is based on the XML model interchange format. To illustrate this workflow, here is a picture:

UML model Exported to XMI → XSLT Transformation (import mechanism) → Database Representation in a database for prototyping

You will design your database model for example in a UML tool and then export that model to XMI representation. Then you import the model with a XSLT transformation into a database that can hold the model in a different format.

If you have done this step, you are able to run the application containing forms for your defined tables – classes in UML. Users can enter first sample data. No code (SQL/other) has been written yet and you will have an independent model, that avoids the typical **lock in** effect of many commercial tools. Ok, there may seem a lock in into the prototyping software, but think about the XSLT template – it decouples the model from the application.

UML is a general purpose modeling language that helps you design software or other stuff like hardware (SysML). XML is a generic machine readable format of data. Most UML editors support XML in the form of XMI as an interchange format. XSLT is used to transform XML documents in one format to another format or some other plain text. UML, XML, XMI and XSLT are therefore the core elements to enable the above workflow of application development. The transformation output in the above case is SQL.

Technical overview

As you have seen in the introduction, it is a clear workflow to become a prototype application very fast. The core enabler technology is XSLT transformation that maps an input format into a SQL format compatible with the internals of the prototyping application. When an import is executed, the following happens:

The database schema will be created while the transformation as a first step. This step includes XMI to SQL transformation and execution of the generated SQL script against the configured database. You have the option to skip this step. A second step will create a model description. It is a SQL script with the model representation in the format supported by the system database for the prototyping application as instructions to enable the prototype. That way the prototyping application knows what to do. You also have the option to skip this step.

When you import the system application model (lbDMF Manager), consider to make a **backup of the database** if you have models in it with manually added stuff you do not have in your UML model. Read more in chapter Import an UML model. With that backup you have the option to export each application at a later time if required.

The transformation step enables you to integrate any UML modeling or other tool that supports XML file formats (database modeling tools for example). Those design tools make the development easier because you begin modeling your application in your understanding – with your preferred tool. This is called model driven development for some reasons: Your primary code is the model – the UML model – and not only a tool for documentation. It is a first class citizen, it is part of your code. It is a as much as possible reduced model that fully describes the intention of your ideas for the first development stage. It may not be a complete model and you may have many different parts of models to describe your problem. UML is a general modeling language and not a DSL, but it becomes a DSL when using the extensibility of UML. UML is then not the only DSL available, it is a graphical DSL. Other forms of DSL's are textual representations. The software described here is not yet capable to understand textual DSL, it prefers to use graphical forms with XML as an output.

When your model is entered and exported, the prototype application imports the model and executes it. It will probably not understand all elements – especially the types – you define in the model. There is a mechanism for this case, but not yet described here.

The model is also called a domain model for some reasons: It consists of stuff about a specific domain your solution should be responsible to support. The domain model follows a specific language – the domain specific language (DSL). Here the domain for generic CRUD applications is expressed with UML extensions and thus is a graphical DSL. A domain may be something like health care, transportation, CRM or ERP. Such applications are typically CRUD applications with some enhancements and special elements. Thus the domain language can be put onto a base DSL as that for prototyping. The conversion from the domain model is done by a XSLT template you can support. In the release, UML2 is supported using BoUML. Newly supported is ArgoUML. You will be able to support model elements for your domain to describe a specific issue like credit card numbers found in a CRM. So your model element may be of a type named credit_card_number. This is comparable with a simple type like a string to hold simple text. With this little type definition in the model you put very much information into the model, because much is implicitly defined: A format for the card number, a BLZ or other encoding to identify the bank. Also it implies a verification process to be used when a card number is entered. All this with only one identifier – the type name. The trick is knowledge about the domain you understand. The technical implementation is part of the transformation from your domain model into the target system.

A main goal at the design step is **not** to make any decisions toward any language to be used or what platform to be supported (Linux, Windows or iOS for example). You only model the **what**, not the **how**.

With the prototype you can test the application before any line of code has been written with a real database and real end users. This is demonstrated by the main application that is used to create prototypes – **It is itself really a prototype** based on an UML model (BoUML using XMI 2.1 export format) with handwritten code of course to interpret the internal model representation. To further throw in buzzwords, the prototype application model can be seen as a meta model.

This is because the prototype will show and let users edit data in a generic CRUD functionality. For a first design this first prototype approach is just enough. It is to the later step (when generating code) to make decisions about HOW the implementation looks like on the specific platform, or how the code must look like. This is done by selecting the code generator transformation – also XSLT.

Design tools need to export the design in XML form. If this is possible, there will be a way to use these tools. The chapter Concept will explain this in more detail. After the option to let users try out the application using the prototyping tool, the ability to generate code (using XSLT at the moment) you have the choice to select a third party template for your desired framework or existing application code the generated code should fit into. See in chapter Quickstart. This enables a step further into a real application that later may be deployed to end users.

Even a code generated for an application for any target environment and language may help to present the application for discussion. It is simply cheap to press a button and build for sample a DevExpress application. The chapter Quickstart will guide you directly into the creation of a .NET application using this professional framework. The template supports simple and end user designable, or dynamic forms, grids with master detail support and end user designable reports. Also a new feature with this release is a real WCF based SOA using an STS for the authorization and an XPO (ORM) backend WCF service providing the data. SOA is a big issue in it's own and the application is definitely not following the principles behind SOA - yet. This is a work in progress and may be a failure to try to put SOA principles into a model driven development methodology at my own. But there are books and companies about model driven SOA that promise the possibilities. [MDSOA]
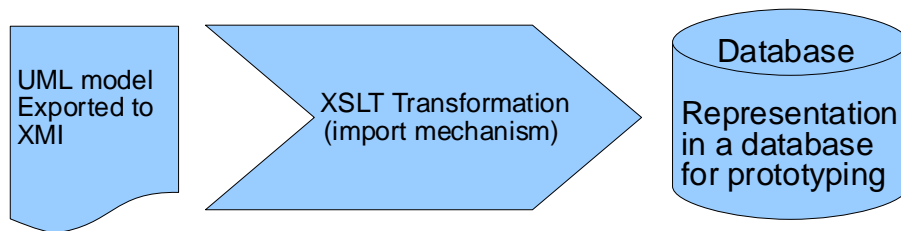
Of course you also could develop your own code generators and probably not only targeting desktop applications. It will be possible for example to generate code for Java, Hibernate, Mac OS X, iOS, Web or what ever.

One of the important point is that you have the choice to export to UML2 using just another XSLT template. This enables **reverse engineering**. That way (including the import) you are free to select your tool of choice and create your own software development method.

Note: Currently, the reverse functionality is deactivated in favor to keep the schema information in the model. This schema information will be fed in by a plugin that will become available as a separate release. The old feature was based on reading the schema while writing the XML output. This solution imposed some problems. Now the schema information is created while the import step out of the UML model. This is a more clean solution and you can define when and with what tool a plain database schema will be reverse engineered into the internal application model (SQL) or UML (XMI).

Concept

Basically there is no silver bullet. Every design tool tries to do its best. Different ideas are implemented, but most times you will find things that does not match your requirements. Therefore the main goal to use XML, XSLT and UML2 seems a good approach for openness. Here is an overview of the development flow (as of the introduction):

UML model Exported to XMI → XSLT Transformation (import mechanism) → Database Representation in a database for prototyping

XML based Design => XSLT => SQL => Database => application prototype

Application prototype => Internal XML => XSLT => XML Design

Application prototype => Internal XML => XSLT => Code.

The main point to identify is to reduce the amount of information to get an idea into an application. This is abstraction. Using the design entry method you are most versed, you only need to create a mapping of the artifacts of the model into the representation in the prototype. If the prototype supports a feature, **you are lucky to only create a mapping**.

If the prototyping application does not support the artifact by any built in feature, **you are lucky to have the option to extend the application**.

In the last development activities of the project, extensibility possibilities have been practiced when needed and proved, there is good support at several levels:

- Adapting XSLT templates for importing different formats (XMI1.2/XMI2.1)
- Adapting XSLT templates for exporting different formats (Internal XML/XMI2.1)
- Adding a plugin to support other serialization format (also possible with XSLT)
- Adding a plugin to get a different representation (like a diagramming plugin)
- Storing new model artifacts in existing domain models if appropriate (for example added ribbon definitions in the formular_parameter table)
- Extending the domain model (by adding entity classes, visitors, integrating containment at various places – hard to do but possible with documentation)
- Writing a XSLT code generator to replace parts of the application for more easy extensibility using models (started, but skipped for the new main branch of version 2.0)
- Writing plugin actions like the master detail or generate code action

Having a reasonable design entry method and the support in the prototype, you can start to get productive, but think:

Using a tool makes you a bit dependable to that tool. It is your decision what to use and optionally ensure more than one tool in your suite. This was a case at a point where BoUML has changed its licensing. A rethinking about alternatives was required. It is a question of time or money to support more than one tool.

A solution with the least effort was required. Using a template to create the application itself into a professional looking CRUD based tool (besides the support for ArgoUML as a different modeling tool). The framework used is DevExpress and additionally a composite application approach is used. By doing so only extensions in the UML model were needed to get a full domain model of the prototype application and let the template do the rest. So plans to support a **tabular design entry** approach for the second choice of tool are made. Indeed changes in the model were possible by the generated application and then the prototype were able to reexport the model into UML2. Tabular design entry worked for that!

If you have a version of BoUML before the commercial version, support is possible. Support for the commercial version is not yet possible.

A third design entry method is to use the prototype application. But it is harder to accomplish, because the tabular design entry version is readonly yet and did not show more than 30 rows. Recommended is the usage of BoUML if you have it, or ArgoUML.

The last methods of interest are domain specific languages and a diagramming plugin. The diagramming plugin has been started but it is still a conceptual preview with no functionality except showing an automatically created view for parts of the model.

To support other tools, extensibility points were listed up. You can read more about extensibility in separate chapters to extend for example with a workflow demonstrated by a TurboVision code generator. See in chapter Using activities for code generation. The new code generation activity features are the most powerful ones that have been implemented. Despite of Open Source BoUML development has stopped, there is an alternative: DoUML. Full support for DoUML is awaiting, because of not yet fully tested XMI support.

If the workflow capabilities are missing some features, then the following chapter Extend the functionality of the core about activities may help. Plugins can be written and called within an UML Activity and that is the important point explained in that chapter. Actions are also used to create business entity validations. An UML Activity action is a plugin in the prototype system that registers an eventHandler who is then callable by the send signal activity UML artifact.

The chapter Extend the functionality of the core also explains how a feature could be integrated into the application to extend the application in general. It explains creating a plain method, and a un marshal method to enable dynamic usage. The dynamic usage can be activated if the function is registered with an unique event name. Then in the UML model of an activity you use the 'Send signal action' artifact and add the event as the value for the signal parameter. Additional parameters are based on the parameters the un marshaller who passes the values to the plain method.

Plugins that register eventHandler, can also register menu entries to fire those events. Depending on the eventHandler it is then possible to start any task the plugin supports. If you register an activity to a menu entry, then you have a problem. Where the activity get it's parameters? It is a design issue and where you plan to use your plugin.

Features

This chapter describes all new features per version and is intended as a short list of new functionality.

Starting with version 1.1.1

- Improved the source code installer for Windows. It should now be able again to compile the code out of the box.

- Made available a wxWidgets code generation compilation package to generate code that integrates in the above built process by manually invoking two make commands after the main code has been built.

- Updated the main model to make initial import of it obsolete, minor bugfixes.

Starting with version 1.1.0

- There is a new plugin that enables using domain specific languages. Initially there is no binary release for this, but get's compiled with an unit test demonstrating the integration of the Antlr3 based language.

- A new code generator compilation has been made available. This new template generates code targeting the Syncfusion Essential Studio. They have a community edition, so just try the new template at no cost. The template is using the WinForms part of their product using the Syncfusion Essential CAB Enabling Kit. Yet it is not much, what I have got within about 20 hours, but get inspired about the possibilities :-)

- Added initial logging mechanism to see access to the services that are generated in the CAB DevExpress Generator template – simply to see where the users come from. The logging mechanism still has to be improved.

- A bugfix has been made regarding handling of datetime fields. They were not handled as strings but required to.

- Trigger generation template code has been fixed by reversing the direction. The trigger complained about referential integrity, but in the wrong direction.

- The few changes and fixes let me made a decision to make a minor release change. Also a completely new model input will be made available using textual DSL's.

Starting with version 1.0.4.4

- There is a new plugin that looks for updates. If new one are found, a dialog appears per week and a tool bar button is added per application start. So if you encounter any issues with this plugin, simply remove the lbwxUpdateChecker.(so/dll) files found in the plugin directory (or within the application bundle plugin directory respectively).

- DMG packages now contain an extra folder for the templates. This is easier to access when choosing a template.

- Worked on better continuous integration support. My CI automatically builds on Mac (Intel/PPC), Windows and Linux (openSuSE 13.1).

- Fixed a toolbar size issue on Mac platforms. There was an overly sized width that now looks more symmetrically.

Starting with version 1.0.4.3

- There is a new table to store table parameters to store special parameters per table. This feature is used for the new permission capability, development has started.

- New code generator for DevExpress based application capabilities are server components for single sign on. An implementation of an STS server and a management application is available as a UML model.

Quickstart

Assuming you have done a domain model in UML and exported it to XMI using BoUML. This step has been done for you in case you do not have this UML tool. Modeling with ArgoUML is explained as an alternative.

CAB DevExpress prototyping

The first example will create a [DevExpress] based application. It is a Composite based application that let you get the second design entry method beside of UML. If you do not have DevExpress, go to the [vendor] and get an evaluation version. A compiled version of the generated application is provided on request.

The generated application will contain several forms that are shown here:
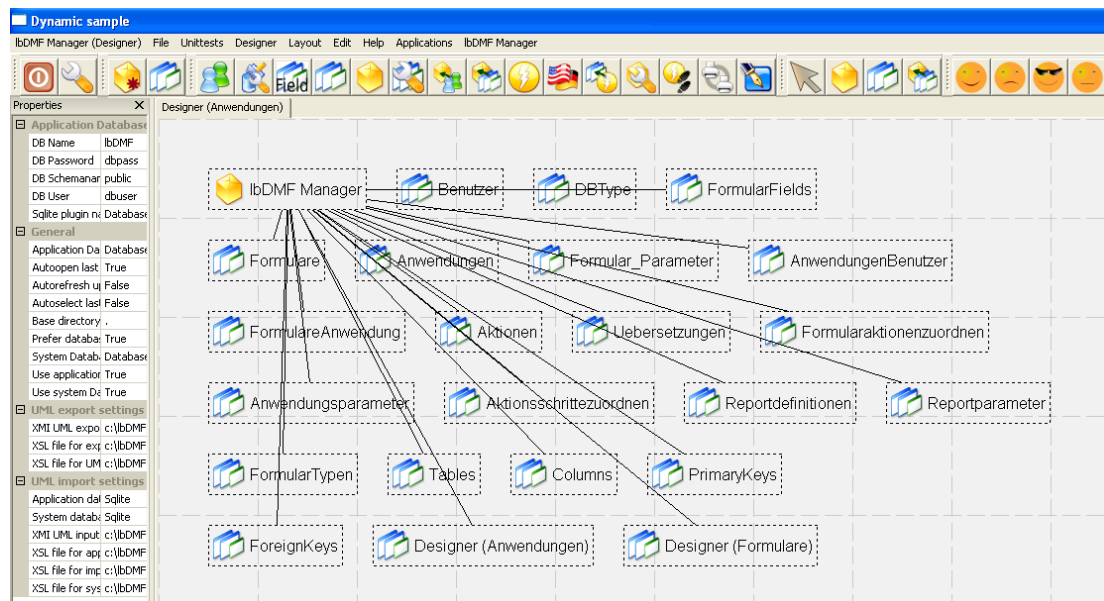
The resulting application will structure the menu by placing them in separate ribbon groups. They are then placed in different ribbon pages. This is designed in the UML model the image above could not display and also the UML model does not display these settings directly. For now, here the resulting application looks like that on the next page.

The generated DevExpress application project:
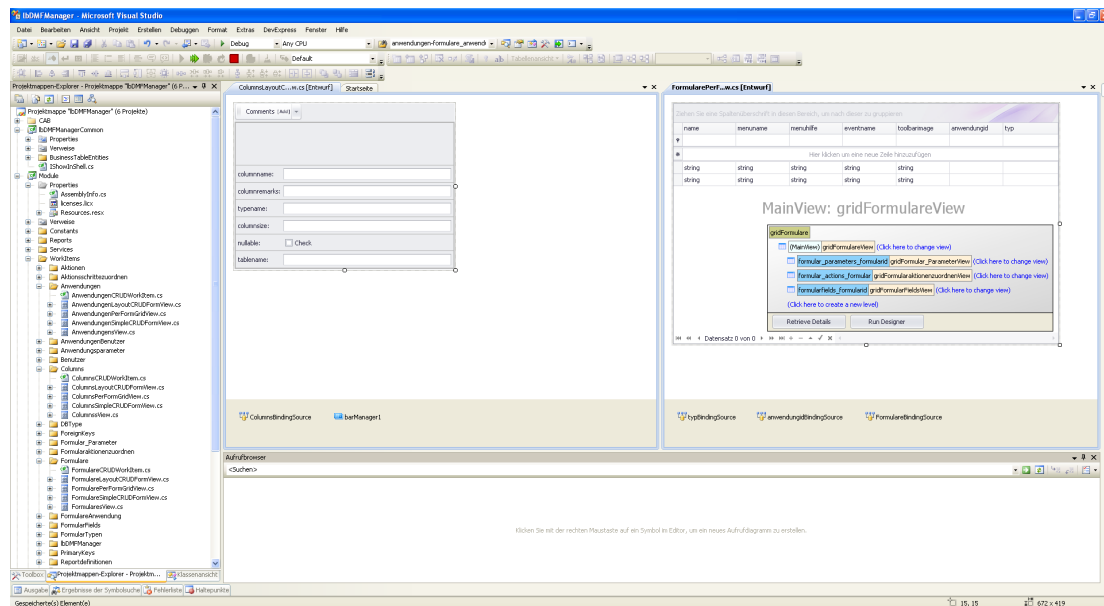
The project consists of nearly 1000 files!

Here is the application showing the application design of lbDMF Manager it self (note the ribbon pages with some icons from the model and others predefined in the template):
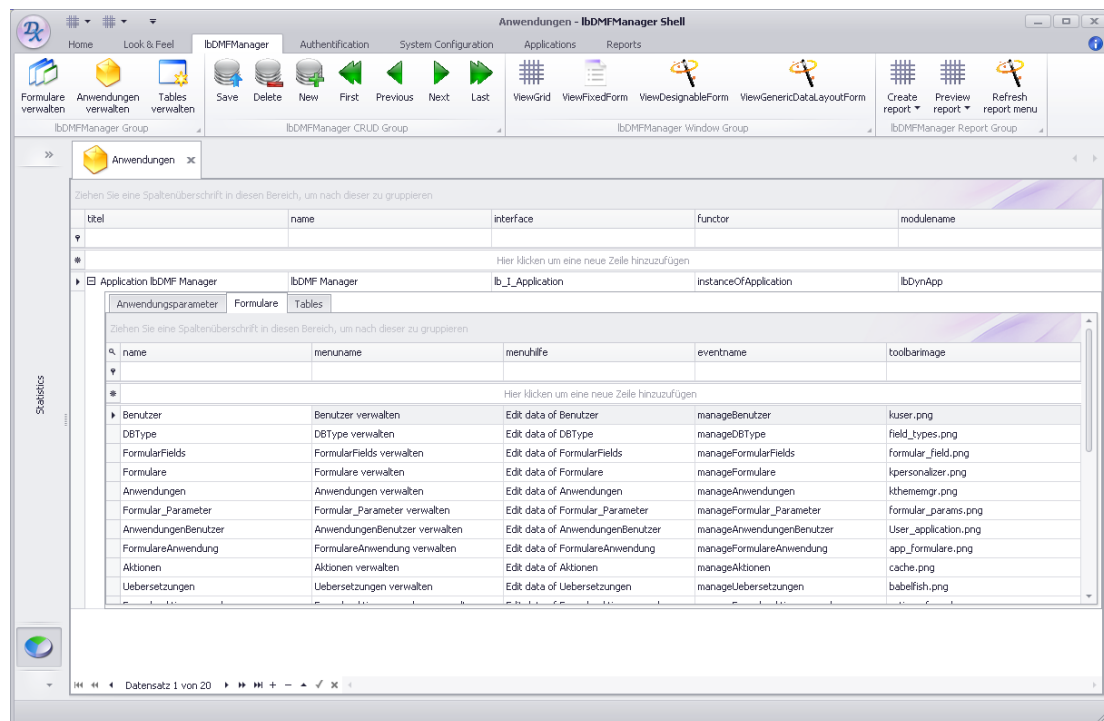
Another screenshot shows another ribbon page:

The steps to reproduce the workflow from design to application are explained now. You could try this with the evaluation version, but must buy after a trial period.

After downloading the binary samples, the CAB DevExpress code generator compilation, and the DevExpress evaluation, you will be able to repeat these steps.

When you install the binary samples and the CAB DevExpress code generator, you need to follow the suggested installation directory, even the second installer will complain about this. It is because this has been tested.

Update lbDMF Manager!	When the installation is ready, click on the icon on your desktop. When you start the application the first time, it creates the application based on provided SQL scripts. You do not need to install any SQL Database. Internally Sqlite is used as default. The SQL scripts have been transformed by XSLT templates that are also provided.
	Import the system application model lbDMF Manager as explained in Import an UML model, but keep the settings in the properties window as predefined. (Pointing to lbDMF Manager.xmi and XMI2.1 import templates)
	After starting the application, press the tool icon to open the applications definition form. Within this form you see the following button to generate code:
	Codegenerieren
	It will start with a path selection dialog when the application model has not yet generated any code. Choose C:\lbDMF to generate code into that directory.
	Then press the Ja button and choose the following template: generate.xsl

This will start creating the .NET CAB DevExpress based application based on the model

information stored in the database (from the SQL scripts). You do the whole steps within a prototype using appropriate plugins that are used in the model! You will notice this with a malfunction if you do omit the initial UML import step to update the prototype application it self.

Here you will see the application with the open application definition form and the focus on the right button to generate code (before initial UML model update):
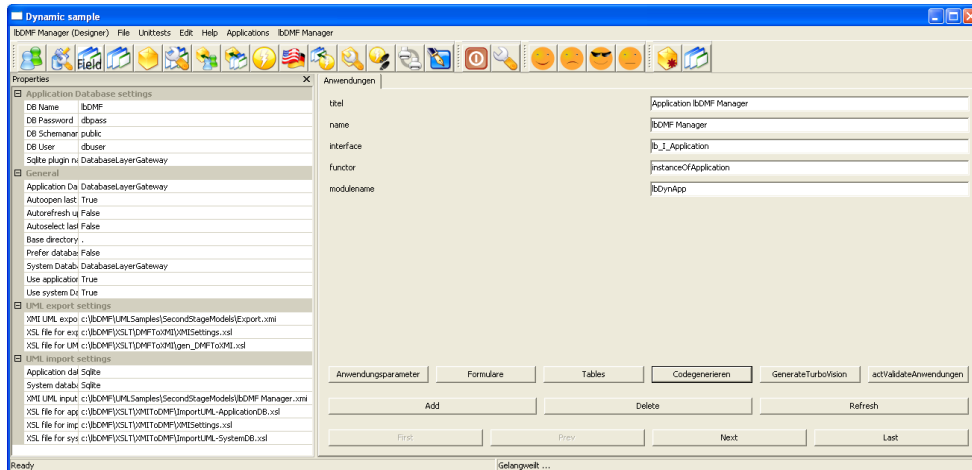
Note: The properties here show the correct settings for the system application update.

Prototyping and Cloud Computing

The new CAB DevExpress code generator is now capable to run distributed and thus the server could be placed into a cloud (IaaS). The application requires a security token service that is responsible for authorization, user creation and password reset. The STS itself will be generated with each project you create. Management requires another UML project to contain the classes / entities the STS is using. All together you can create complete code for the components in the environment.

You can download the current version of this generator as an installation package for Windows only because of the target is a Windows application only. Go here to get the required software. Download the Windows installer behind the Windows logo and the most right link to get the CAB DevExpress generator. Download the DevExpress evaluation version, Visual Studio Express or SharpDevelop.

Note: The new SOA based CAB DevExpress requires to apply application parameters for the setup projects (product id and upgrade code). These are applied in the package of the UML model. Also WCF endpoints have to be defined. They are actually pointing into a cloud you cannot use.

As of the new SOA features implemented in the code generator an article will be available at CodeProject to further describe the architecture behind it. It is too complex to put it here as a full chapter for this release.

Note: The SOA application points into a development cloud, that is hosting the WCF services. They are not always up and running, thus and because accounts are not provided to access the services, you have to deploy and run your own.

Note: The application requires a valid certificate, a thumbprint is configured in the code generator template. This is not yet made configurable and a todo. Replace the thumbprint values with your own. You may simply create a certificate for a trial of one month at any CA.

Please contact me, if you have any issues to get up and running your own cloud.

A version without a cloud environment is the previous release of the software. A current version without a cloud will be made available later as a configurable option. Currently this is only configurable in the template itself.

To disable services and brokered authentication, do set not to use the datastore client (USE_DATASTORECLIENT), disable authentication (USE_AUTH) and disable brokered authentication (SUPPORT_BROKERED_AUTH).

To add additional parameters per application to configure these settings in UML, you need to always setup these parameters. It is an exercise for you to implement this feature if you more often need to configure this.

SCSF Application block

The next sample is from the second old version of the documentation and simply repeated here. You do not need a third party framework but only .NET it self (Visual Studio or SharpDevelop). You simply select another template and repeat the steps above. Please note, that it may function only with the release of my software at that time.

The steps result in an application you could compile with Sharp Develop 4.0 or earlier. The application is based on the SCSF Application block from Microsoft Patterns & Practices. It shows the generated application in front:
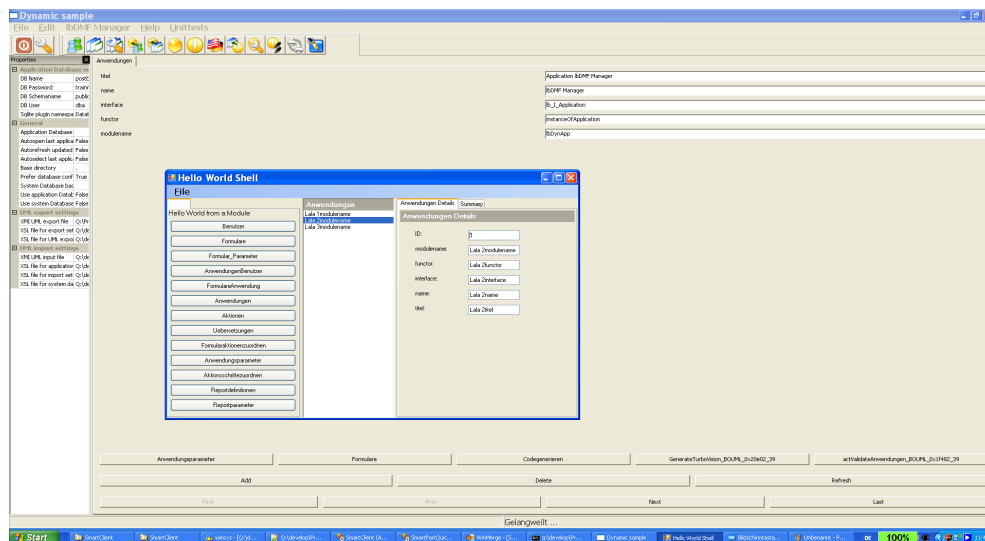
To get the sample application in front compiled, you need to download the application block mentioned in the page above. For reference you can go directly to the download page of the software package from Microsoft you need. It is the 'Smart Client Software Factory – April 2008' for Visual Studio 2008. If you have installed the software, please go to the menu Microsoft patterns & practices/Smart Client Software Factory – April 2008 and execute the Smart Client Software Factory Source Code Install. This will typically install into your user folder and creates a folder 'SCSF-Apr2008'. Check 'Open new working copy folder' to open the source folder. Then copy Lib and Blocks to the installation folder of wxWrapper installation (typically C:\lbDMF). Then you could start generating code.

Note	The Lib and Blocks folder are supplied by the CAB DevExpress code generator.
	Also the images may have to be copied into the generated application, as this was not yet placed well into a common directory. The old template does not yet use the images installed with the CAB DevExpress code generator.

To generate the SCSF based application for the lbDMF Manager application, open wxWrapper, then open the 'Anwendungen' form in the menu lbDMF Manager/Anwendungen verwalten or click to the icon as below:

Then in the form click on the button 'Codegenerieren'. Then click 'Ja' to start selecting the template to use.

Choose C:\lbDMF\XSLT\SmartClientSoftware\generate.xsl. This will create a folder named C:\lbDMF\SmartClient. In that folder you will find the solution. If you have followed the setup instructions and chosen an output base folder that was omitted above, as it should work without that, you could compile and run. Therefore start the solution either with Visual Studio 2008 or with Sharp Develop to compile it and then try running the application. You should get a similar screen as shown in the screenshot above. The data you will see in the application is from a fake, not a real database. This is to be developed by you as an exercise.

A blog for further reading is available. There you will also see other (not yet available) templates and stuff around the project.

If you do missing some images in the project, copy them over from the C:\lbDMF\Images folder after installing CAB DevExpress code generator.

Start with Modeling using BoUML

In this chapter you will learn, how easy it can be to start with this tool. The samples of the UML models are based on version 4.9.3 Mac. The Windows version is identical and should be replaceable. The goal of this tutorial is to show the little modeling artifacts needed for a first design. It is simply a view to the tables that get also transformed into form representations.

Note	The tabular design entry will become similar by simply editing the table schema and then chosing the right export template. These steps need to be made more simple

Creating an UML model

To create an UML model you start BoUML. If you have the last free version, you are luky. In the 'Project' menu you select 'New' and enter the project name. Enter 'BoUML'. You then will see the following warning message:


Close the window and select 'C++':
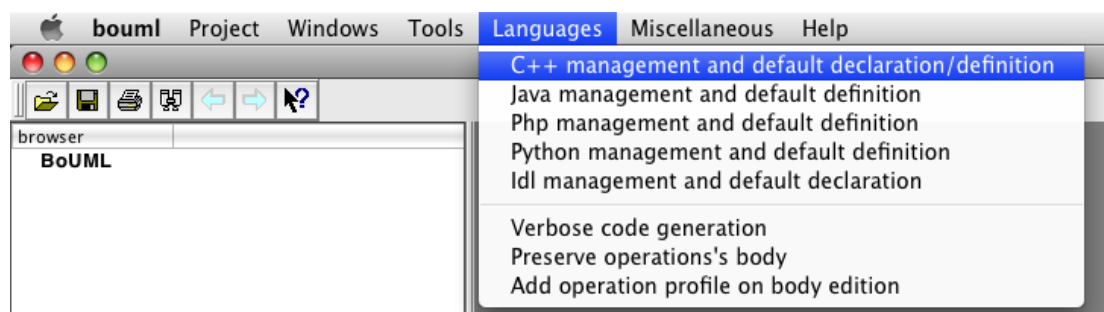
This setting relates to the internal code generation of the UML modeler and doesn't matter here.

Package names

There is an important difference between project name and package name. Until now you have entered a project name that is used at the same time for the package name. Later you have to remember to the package renaming if you rename the project name.

Creating packages

To work with multiple packages, you could create new packages inside an existing package. Here you will see, how:
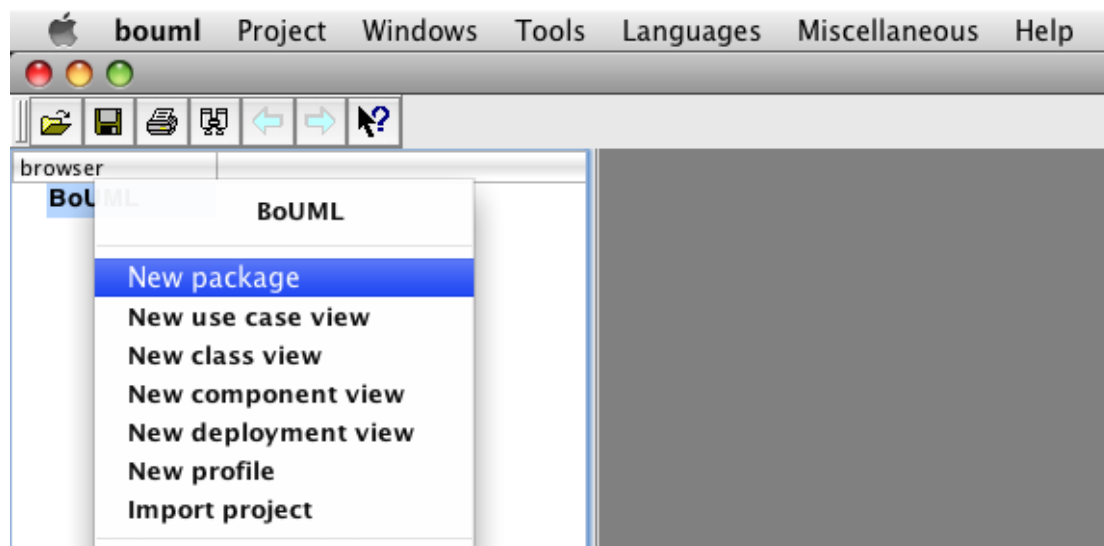
Multiple packages are not supported yet or are not tested at least. If you use a revision control system like CVS, it is recommended to create a package to avoid renaming issues with the project related to the revision control system.

A package name is equal to the name of the application. The inner most package name is used for the application name. We now use a package name 'CRM'

The package is now also used to contain application level parameters. The parameters are a way to configure things like MSI package id's or upgrade codes. Also any parameter used in your templates can be configured here, if it is scoped for the application level. If you add new parameters into the UML model that are new to the template, do not forget to modify the templates to support your new parameter's values while generating code. The same is true for all new UML attribute types, parameters in form definitions, parameters in a table definition and probably other UML artifacts like stereotypes. Using stereotypes you will have a much more powerful modeling feature to express meaning on top of plain types and parameters.

Create a class view

After you have created a package, you create in there a class view. See the picture above to spot the next menu entry for creating class views.

The name of the class view is not relevant for modeling the application and can be named as you like. There is also the possibility to use multiple class views to structure your design to logical pieces. Use 'Start' as the name.

Create a class diagram

To use graphical documentation and modeling you create class diagrams. This is done with 'New class diagram' as depicted in the picture above. Use the name 'Tutorial'.

Now you will see the following:
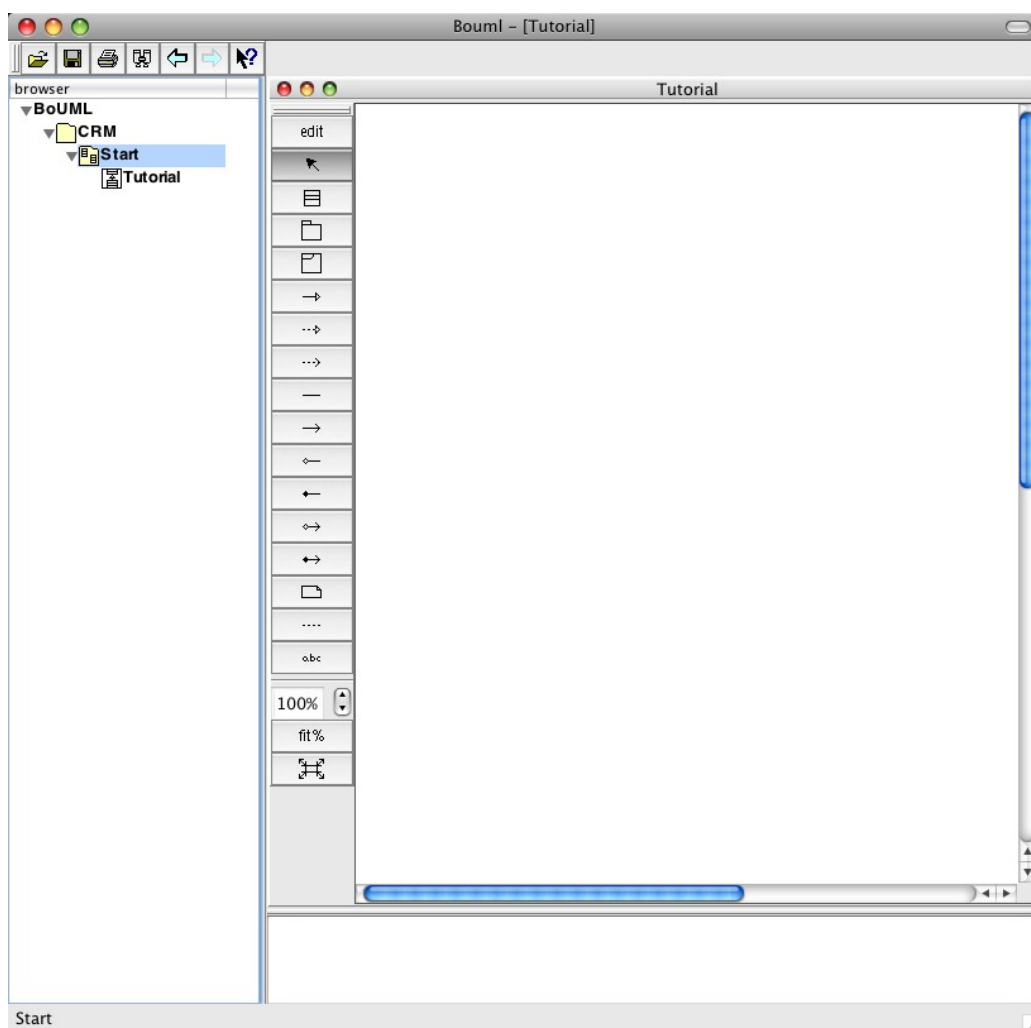
Start modeling

Now you have created a project that contains a class diagram with them you could start modeling. Save the complete project directory as a template for new projects. (If you copy a new project from that template, rename the package name for the most inner package to your new project).

Create some classes

You create new classes with the following symbol:

Thereof you get the following class dialog to enter the new class name.
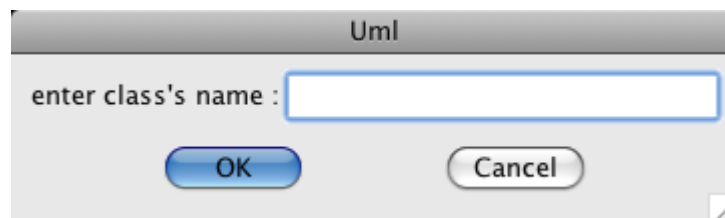
Do not use a classname multiple times. You could use the same class multiple times in a class diagram. This is useful for an overall overview of classes without all the details in each class.

Relate classes

Classes usually have relations to other classes. An important relation in our modeling is the following one:
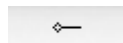
The side with the diamond applies to the referenced class. The referenced class therefore is the primary class and the other is the foreign. In the following sample we will create a class named 'Kunde' (customer) which is referenced from a second class named 'Bestellungen' (orders). It is then recognized as a one to many (1 to N) relation. Here we have orders that are related to customers. Therefore the diamond is on the side of the customer class. There are other presentation or modeling alternatives to achieve the same result, but those are not yet supported in the prototype application import XSLT template.

Therefore you use this notation for database relations.

Until now the model looks like as follows:
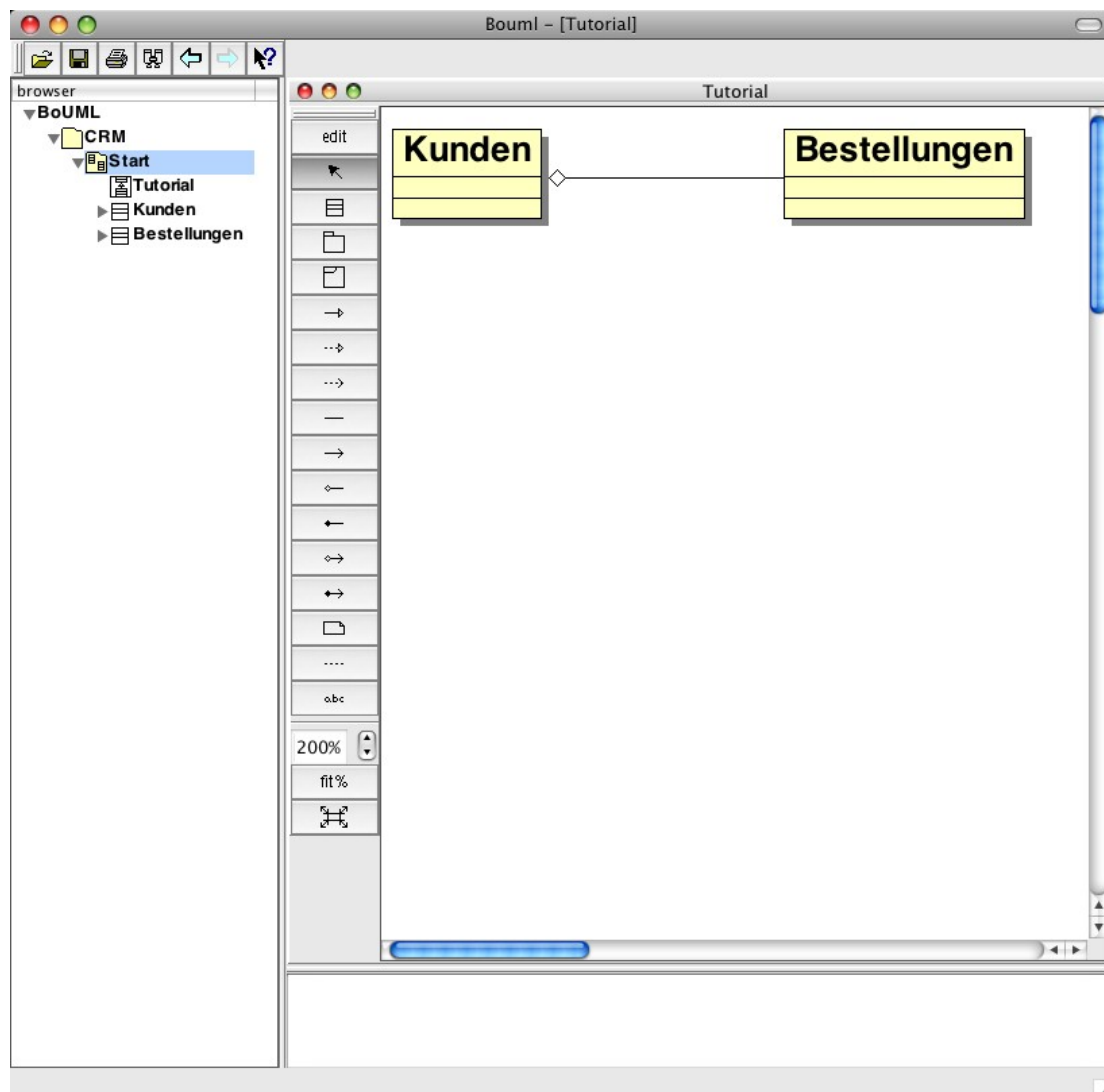
As you see, the tree structure is larger. You now could right click on these classes (tree or diagram) to do the following actions:

- Create attributes
- Create operations

Attributes are columns in the database table. Therefore the class name results into the equivalent table name where the data is stored later. Operations are handled as controls at the form you later will see. Until now there are supported only few operations. One of them is the execution of a stored procedure.

Creating attributes

We now create some attributes we think to be required in the form for now. To do this you right click on the customer class and there click on 'Add attribute'. Then you will see the following dialog:
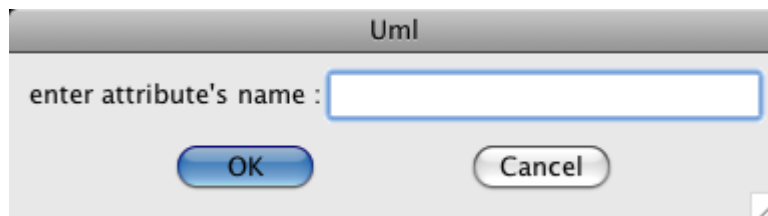
Assign to the customer the following attributes:

Firma (company name): string

Anlage (creation date): date

Kreditlimit (credit limit): float

The class orders becomes the following attributes:

Bestelldatum (order date): date

Bearbeiter (sales rep.): string

Now your UML model looks like this:
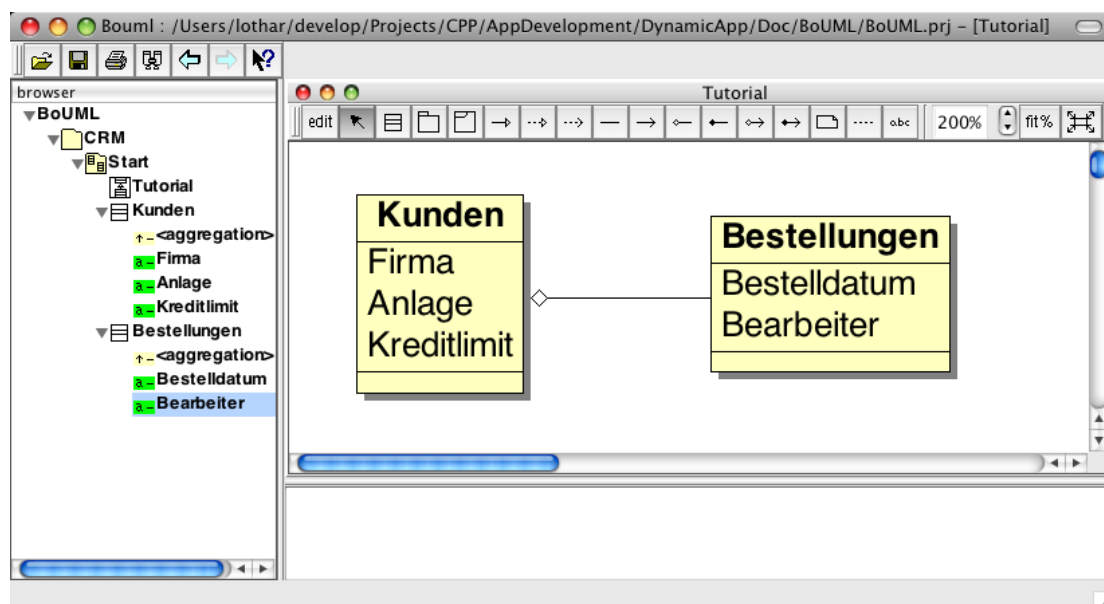
Export the UML model

After you have finished your first model, it is time to test the application. Therefore you export the UML model as follows:
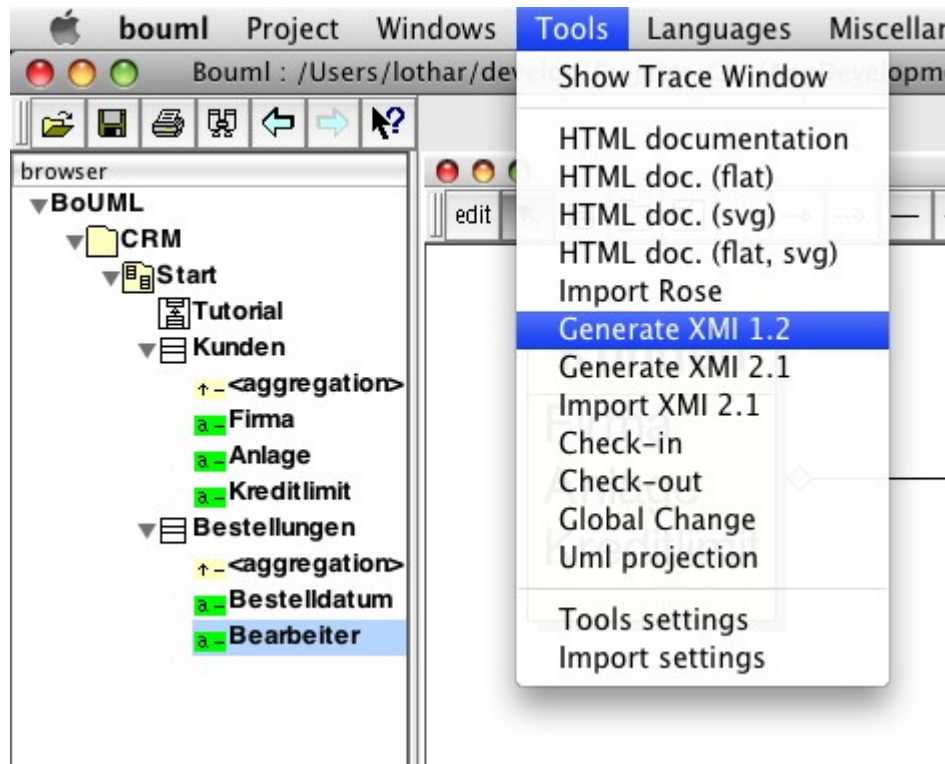
Note: This is an initial UML model. It has no detailed information as of modeling the forms and the database tables separately. Thus it is important to select 'Generate XMI 1.2'

Now there appears a dialog that has been created from a support application of BoUML. It's name is 'gxmi'. The application probably may not be visible. If that is the case please look at the task list. Select an export filename or type in a new one in the field left of 'rows'. (Browse, the layout on Mac is broken):
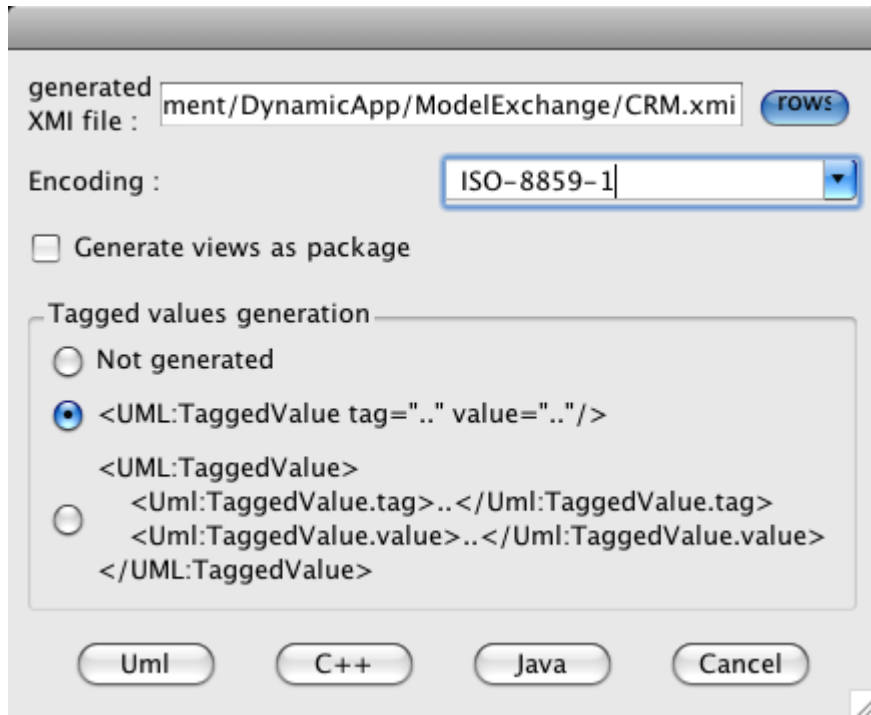
generated XMI file : ment/DynamicApp/ModelExchange/CRM.xmi `rows`

Encoding : ISO-8859-1

☐ Generate views as package

Tagged values generation

◯ Not generated

◉ <UML:TaggedValue tag=".." value=".."/>

◯ <UML:TaggedValue>
    <Uml:TaggedValue.tag>..</Uml:TaggedValue.tag>
    <Uml:TaggedValue.value>..</Uml:TaggedValue.value>
</UML:TaggedValue>

(Uml)   (C++)   (Java)   (Cancel)

Take the same settings as depicted. Important here is the settings of the encoding. It can't be empty. The filename is later used at the import into the prototype.

Start with Modeling using ArgoUML

Here an alternative to BoUML is shown. This is a new feature, as of you may no more be able to get an Open Source version of BoUML. The alternative also demonstrates the ability to change and keep you unlocked from any vendor. Creating a new model includes some basic setup in the tool like in BoUML. These setup includes the types you want to model with and a stereotype. The steps are shown by simple screen shots and some words about it. The documentation will be not as complete as that of BoUML due to missing features compared to BoUML.

Set the Default Package Name and create a new one

The default package name should name the application of your model. Rename it accordingly. This is not yet used, but it may be used if you have more than one package and you need a common application name. The application name that is used is the one of the most inner package.
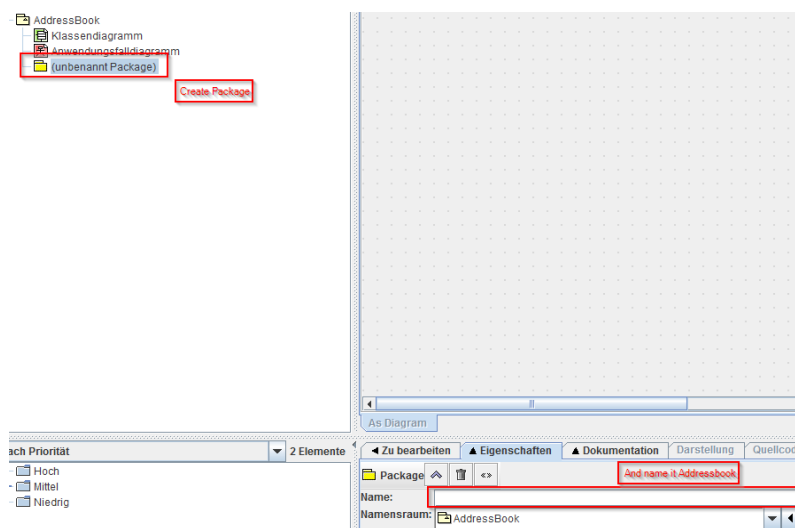

When you have renamed the most outer package name, add a new package within the renamed one and name it with the same application name. You see this in the above image. The name for this (the inner) package will be used for later features to split up model packages into separate projects. This is useful if the model artifacts become more and more.

## *Create basic data types used in the XSLT templates*

To get a functioning model, you first need to create some basic data types, the import XSLT templates understand. To do so right click on the outer AddressBook package and create new data types. When you not do so, ArgoUML **will create new classes automatically** and thus the XSLT template thread this as forms like the classes you would like to model as forms. This is because any new type should be defined before it's usage.

The basic types needed for this demo model are string, date, bool and int. The types string and bool are really used and tested. The date may work, but does not provide a default value – for example. Another useful type may be float as in the BoUML example above. Here are the types:
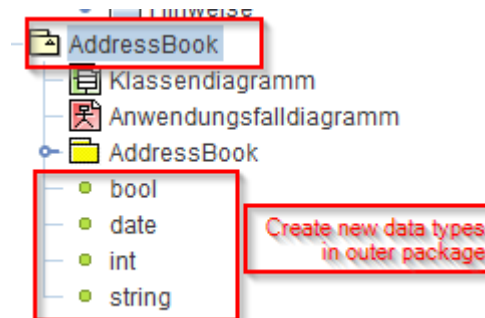
Note that this was done after creating the classes by a mistake. The package is not expanded. The next step will not show the types due to the small change in ordering the modeling steps. The types should be created before any attributes use these types.

Create three classes for the AddressBook
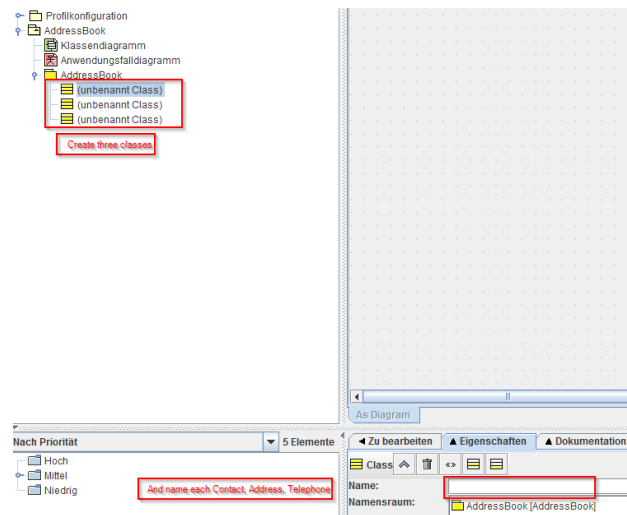
Right click on the package and create three classes. Then name them accordingly Contact, Address and Telephone. The attributes are shown in the next section. Create them as depicted in the screenshot of the next section. Also apply the given types that have to be created beforehand.

Create aggregate associations

The new classes need to have relations to make sense for a simple address book. Create them as shown in the following screen shot:
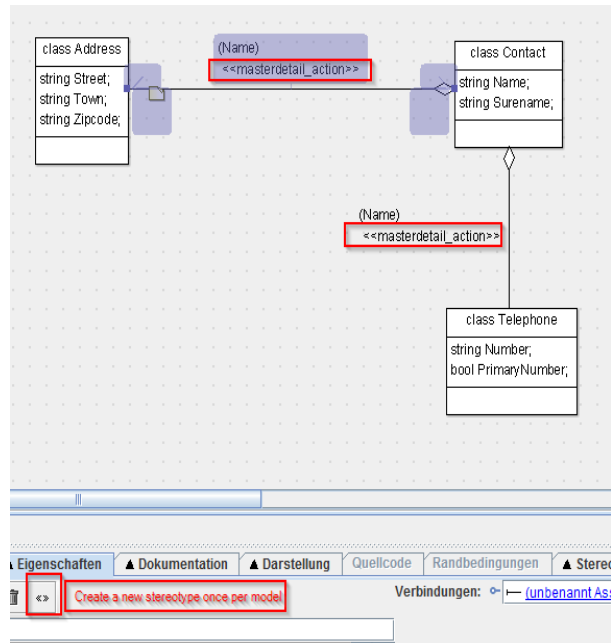
If you have done this, start creating a new stereotype. This will then be visible when applying stereotypes for other aggregates. When you have done the naming of the stereotype (masterdetail_action) then edit that stereotype as follows:
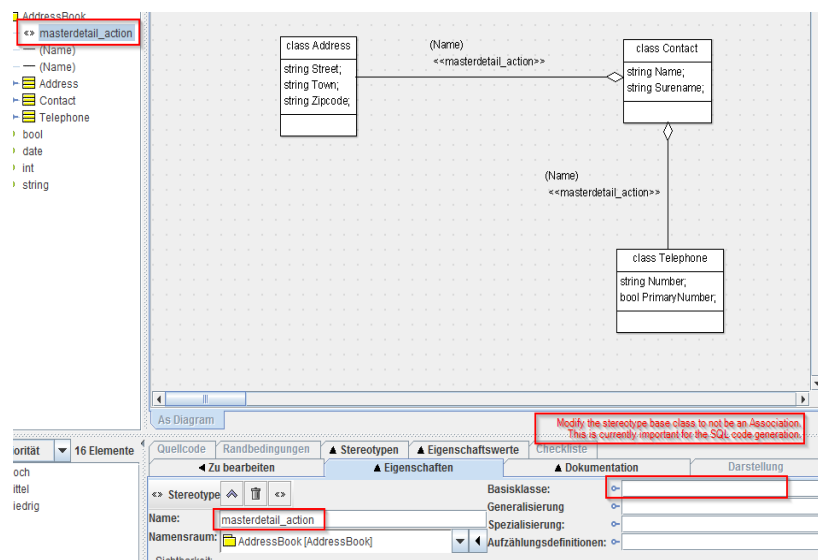
The base class in this case has to be **empty** to not break the XSLT template. This has to be fixed later. Apply the stereotype to the other aggregate shown above. You finished your first model.

Export the UML model

When you have done the design, you need to export the model into a XMI file. This will later be used to import the model into the application. To do this, go to the file menu and choose export XMI. Name the file as your application or package name.

Import an UML model

To use an UML model created in BoUML or ArgoUML, it is required to import it. As it is an initial UML model exported as XMI 1.2 (in BoUML or ArgoUML), you need to select the new XMIImport.xsl and XMISysImport.xsl files as shown here. Other settings that are also important are also highlighted:
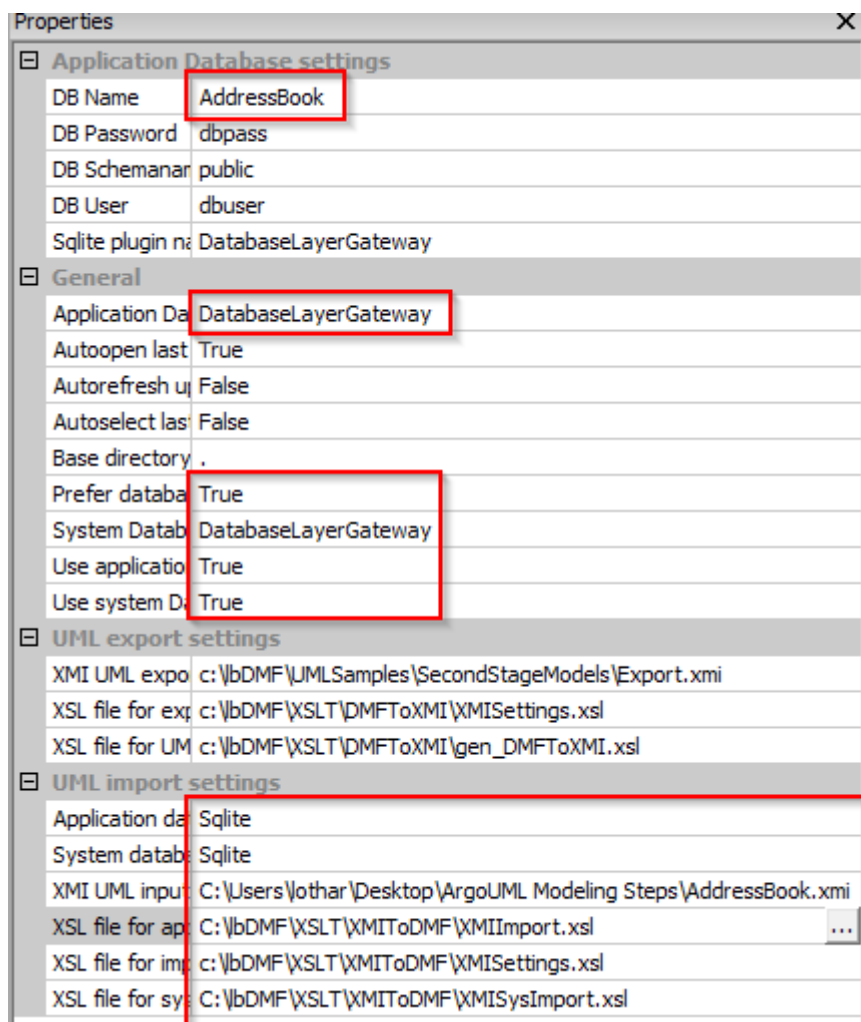
The default import templates are those for XMI 2.1 and BoUML! The additional features in BoUML are currently not available in ArgoUML. Be patient and wait :-) If you have entered

the settings, you are ready for the import. In the following picture you see how to start importing (the XMI file folder from my tests may differ):

Note: If the UML import settings are not done correctly you will be notified. This is when the Application database backend type and System database type fields are empty. A backend database type should be defined and is used in the templates. Another **important** setting has to be activated when you like to overwrite the database schema for the application. In a production database you are **strongly** encouraged to make a backup before overwriting. It **will delete all** data by dropping the schema. It may fail in case of relations not captured in the model, but exist in the database. It may fail if you add new relations, as it is not currently checked if they exist, before dropping the relations or tables. For Sqlite choose to delete the file itself for simplicity :-)
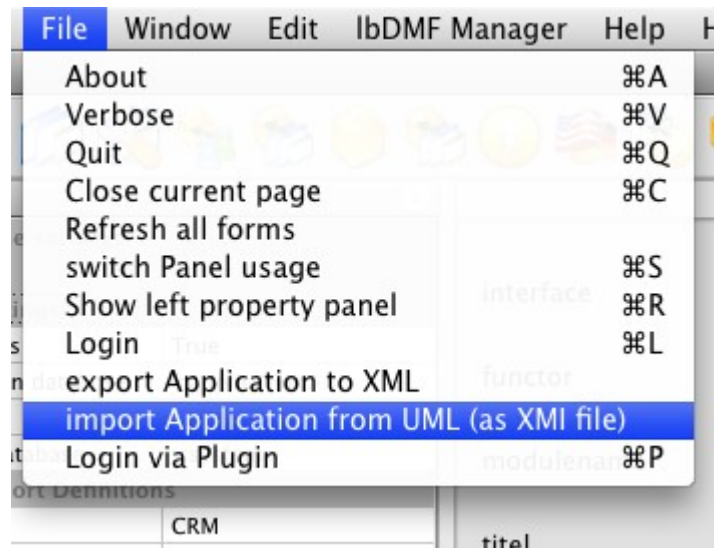
Import steps

The import is done in two steps. The first step creates the application database that represents the physical data model of the UML model. The second step imports the application configuration into the system database. You could skip the first step if you have modeled according to an existing schema (you may be able to do this by taking the same table and column names and their types – for those that are supported).

Note: If you want to create an UML model from an existing database, you create a dummy UML model containing only one class with a name 'Dummy' for sample. Import that UML model and skip the creation of the application database. You don't really need it now. Check in the application settings of the newly created application model the database user and the password. Also you need to deselect the usage of the database backend for the application database as shown below. Then export it when you have it running. This automatically tries to collect the database model and includes it in the exported XMI or XML file.

Note: This step of reverse engineering is removed. The new schema tables in the application model have to be filled correctly, as done with an XMI import. The upcoming reverse engineering plugin will do this in future. In case you really need this, use an older release than the final release. You also need to use BoUML due to the export method of my application writes XMI 2.1 only. Another UML2 tool may be Papyrus, but this is completely untested yet. Contributors are welcome!

Testing the new application

To test the new application, you simply go to the application menu and select lbDMF Manager. Then in that menu a new entry will appear. You don't need this in subsequent imports of the same application. Simply select the application you like.

Note: There are some issues with the prototype not to cope with the forms defined. This does not break the code generation. A fix is provided as soon as possible. So testing is done directly with the application that has been generated. See in chapter CAB DevExpress prototyping for how to generate code. The running lbDMF Manager application as an example that directly runs from the application model if some settings in the corresponding template are set to disable use of the service based backend datastore client (USE_DATASTORECLIENT), disable authentication (USE_AUTH) and to not use brokered authentication (SUPPORT_BROKERED_AUTH).

Reverse engineering existing databases

TBD: This feature has to be moved into a plugin. Currently it is not possible.

With the included capabilities of the prototype runner it is possible to create a prototype for an existing database application. There are many reasons why you want to do this.

- You want to replace an existing application
- You want to provide a separate application with parts only
- You want to create a web interface and therefore start with a prototype

There are truly more reasons, but the steps remain the same.

The sample Postbooks

As a sample here we create a prototype for the Postbooks application. It will show the capabilities of the prototype runner. The following screen shots of extracting the database model from PostgreSQL are taken from the Windows platform and are not shown here. The equivalent Mac screenshots are comparable.

Setup the ODBC connection to the database

To use an existing database for a prototype you need to setup an ODBC connection to it. Use the informations of your database vendor. This description has been tested on a PostgreSQL database. Other databases are possible too, but there are probably required changes in the XSL file to do before you proceed. Developing XSLT templates or changing them is handled in a separate documentation (TBD).

Create a new UML model (Quickstart)

You need an empty UML project in that you add one class. Name the project and the package Postbooks. Have a look in the chapter Start with Modeling using BoUML how this is done. Attend on the class name and do not name it like one of the tables in the database you like to create a prototype for. Name it 'Dummy' for sample.

Export of the model in XMI 1.2 and import

You export the model as of described in chapter Export the UML model and import it in to the prototype runner. Thereafter you check the database access settings in the application settings ('Anwendungsparameter' in 'Anwendungen') in the newly created application model. The important settings are 'DBName', 'DBUser' and 'DBPass'. These settings must reflect your ODBC settings to access the database. **Please note that we still using XMI 1.2 format.**

Prepare for Reverse engineering

To reverse engineer a database model, you need to have an existing ODBC setup. Read more in chapter Setup the ODBC connection to the database. Then you need to run the newly imported application. Click on 'Edit'->'Autoload application' to deactivate this feature. Then you have the ability to start another application. When you start exporting, the database meta information will be retrieved and included in the export.

Export the application as XMI 2.1

If you have imported the dummy XMI 1.2 model you need to make the following settings in 'Use application Database backend': 'False'. Also setup the XSL files in the group 'UML export settings'.The base directory is XSLT_Templates/DMFToXMI. 'XSL file for UML export' is 'gen_DMFToXMI.xsl' and the file to be generated is 'XMI UML export file'. Name it as you like. Note also the XMISettings file.
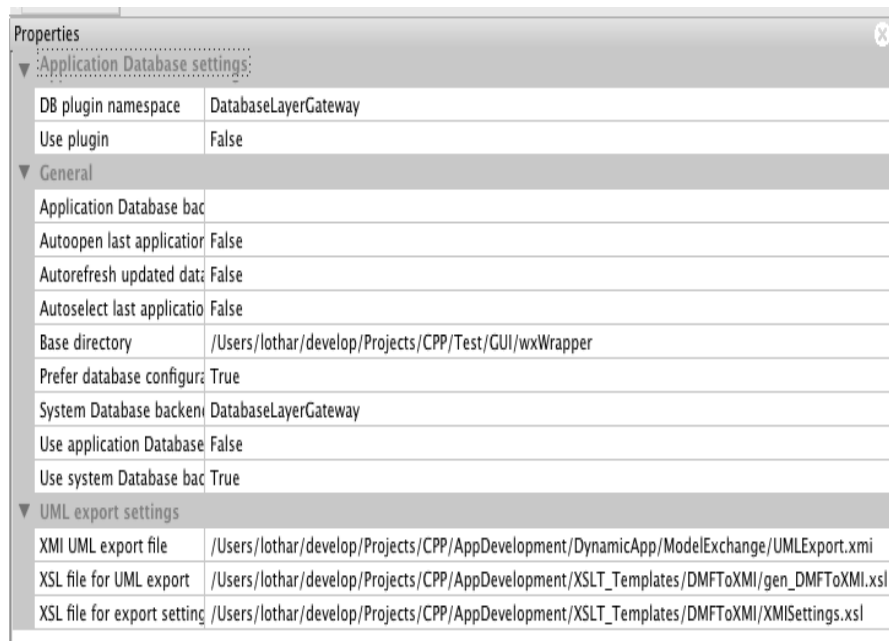
In your environment these files may be at another place. The sample screen shots are made in my Mac OS X development environment. On the Windows standard installation these files are located here: c:\lbDMF\XSLT_Templates\DMFToXMI.

You could only create XMI 2.1 files with this export method. If you like another format, you need to change the template, but better create a new one.

There is currently also a limitation when you have created XMI 2.1 files. There is no way to reexport them as XMI 1.2 files. The XMI 1.2 files are used as a 'first starter' with enables simpler modeling.

Note: The sample database, here Postbooks has two tables with two primary keys each. The templates that will import this model from UML as XMI 2.1 files are not capable of handling multiple primary columns. Remove the second <<key>> stereotype tags in each of the following Entity classes:

The classes: aropenco, payaropen.

Other problems haven't arosed while my tests with Postbooks. But in general arising problems can be narrowed by using manual transformation and check the resulting files using XSLTPROC. In any case you could correct things by modifying the templates. Read more in the upcoming 'Developing XSLT templates'.

Import the XMI 2.1 model in BoUML

If you have exported the model, open BoUML to import it. You could use the dummy project created earlier, but delete the elements as follows:
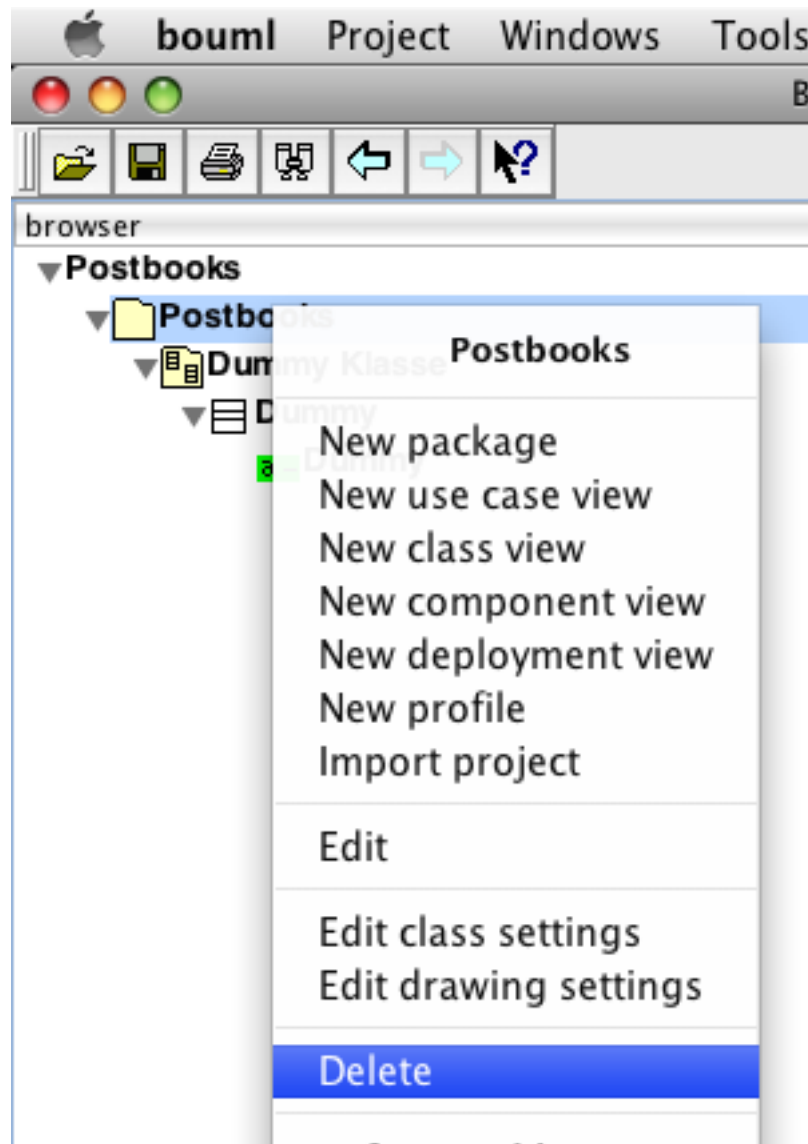
After that the elements are still visible, but marked as deleted. You could revert the deletion, start with the import or reopen the project to remove the deleted elements.

Start the XMI inport into BoUML

After you have created an empty UML model or deleted all stuff therein you could start the import as follows:
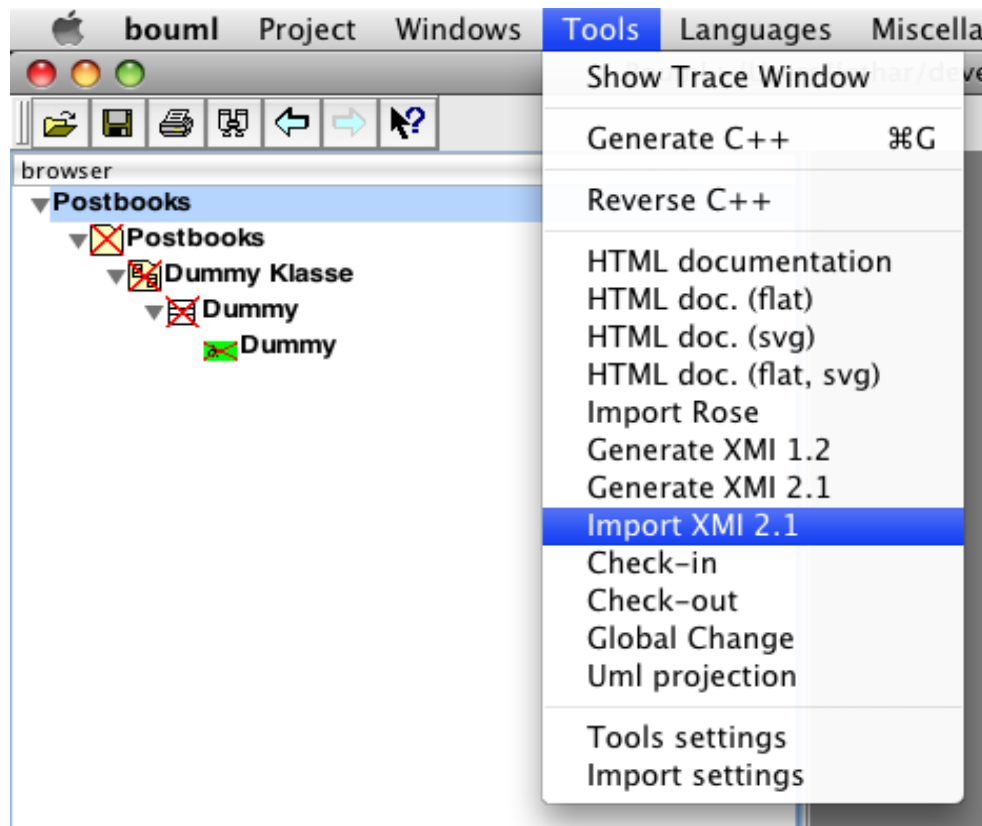
You see here, I haven't reopened the old model, but the import is possible. If you have done the import you will see how it looks like on the following page or like the next when you reimported the export from BoUML :

In that case you could move out the selected node into the outer most element and delete the other to correct that.
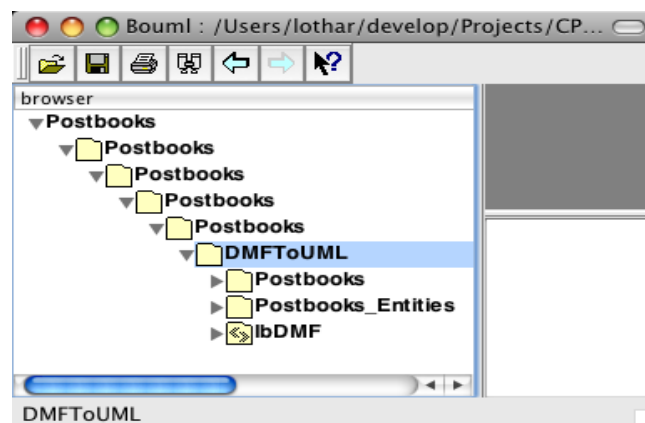
Postbooks application as UML model

Here you see some classes that have been reworked after the import into BoUML and are declared as forms (stereotyp <<form>>). Also you see the relations from forms to their entity classes (stereotyp <<entity>>). Entity classes represents the tables in the database. The forms are created by duplicating them and then unnecessary relations are removed, before starting creating relations. This is a BoUML feature I don't like here, because it is little too much work.

In that way the new application comes up based on an existing database. You could select specific tables only to provide forms in the new application (by duplication and moving them into the correct package).

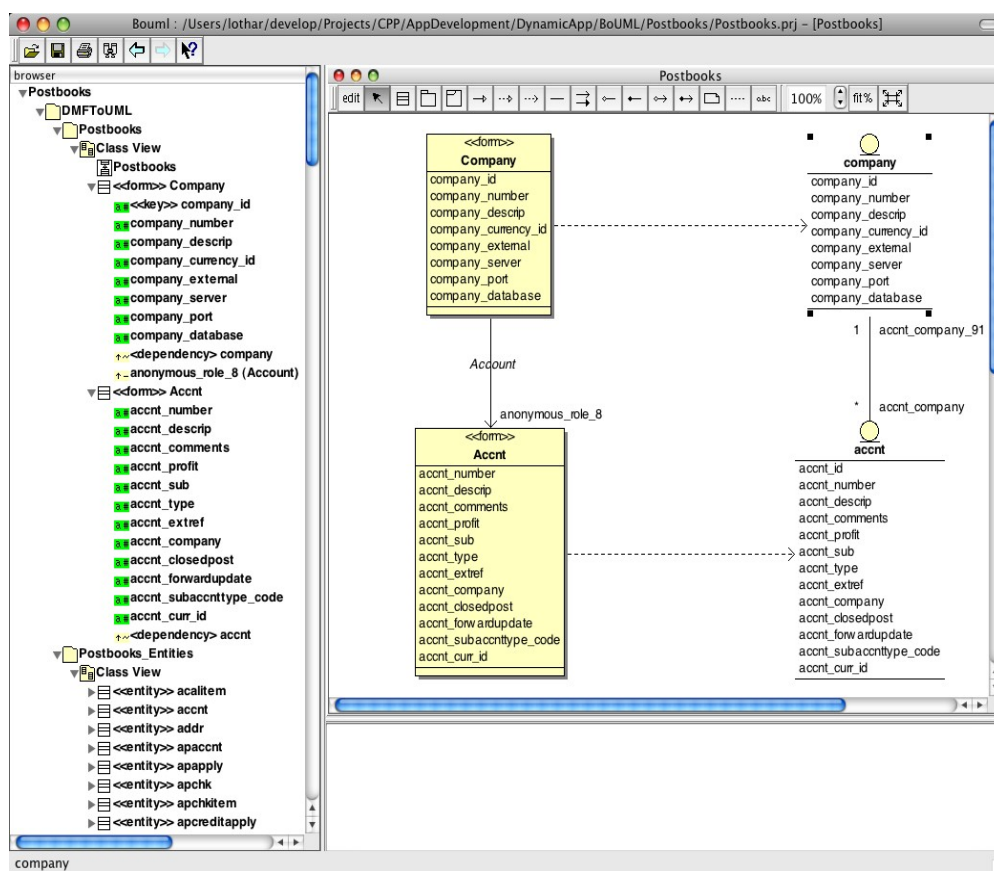That in this way created UML model (firstly the package 'Postbooks' is empty) could be extended by your means. It could be created multiple forms based on the same table (for sample for different WHERE clauses who currently get lost due to missing XML elements where to store the where clauses).

The tables are located in the package 'Postbooks_Entities' and the form classes are located in the package 'Postbooks'. In the diagrams you could show them together. Note that it will be better to create the diagrams in the entity package, as I have noticed there would be drawn all relations. For a more complete information on UML modeling you could read the upcoming documentation about UML modeling.

Postbooks application as prototype

Here you will see two screenshots of company and accnt table in the prototype on Windows. I have currently no Unicode drivers for PostgreSQL on my Mac, thus the resulting screenshots are done using Windows.

The Company form shows some of the columns in the table. Note the detection of different types:

Note: The sample application shown here does not exactly represent the UML model shown above. There is an arrow from the 'Company' to the 'Accnt' form class (<<form>>). This will result into an additional button in the above form to jump from Company to Accnt (account).

Here is the Account (accnt table) showing also some drop down boxes for the foreign keys that are shown:

Here I have not shown, that the application will ask you what field should be shown in the drop down boxes for each foreign key. This happens once when you don't model this before in UML. If you reexport the prototype to UML (XMI) you will get an updated model from the gathered data for columns to be shown instead the foreign key itself.

Applying bussiness rules

In the new version as of 1.0rc4, you have the opportunity to add business rules to your application. (This feature is not yet possible with ArgoUML)

What are business rules?

With business rules you have a mechanism to ensure correct data entry, that is beyond validating values in forms. A business rule for sample is a validation over more than one field to ensure that several values met the criteria for the current business action you perform with the change of the actual data.

Technical implementation of business rules

The technical implementation of business rules is done with an extension of the existing action functionality you could use to model your application. The actions firstly have been described in the document 'DynamicApp.pdf' on page 10. You should find this document in the source code distribution or at the linked document.

An action can be understood as a button click. Some buttons on a form are predefined actions such as 'First', 'Previous', 'Next' and 'Last'. These actions and other buttons are 'CRUD' actions you could perform in a simple 'CRUD' application.

But a business rule could also be understood as an action. It is a validation action to ensure correctness and can include proper error messages if the rule doesn't match.

From the technical view, the action and therefore the business rule too, is a configuration of what should happen on certain tasks or should be possible on a form. The form could have several actions. The first and more simple actions are buttons to open a detail form or a master form. These actions are of a special type that will create a button on the form. In the document mentioned above I explained how the actions are modeled in UML. In this document I follow the type of documentation. It will not contain internals of how the configuration is stored in the database or the XML file. I rather let you get started with modeling the business rules of your application in UML.

A first business rule

I will describe the new features with the CRM sample. There exists an initial UML model without the new modeling artifacts and a new UML model with the new business rule I have added. The pictures at the beginning to start modeling in UML are a little bit different, but that should not bother you.

I'll start to explain the business rule in words as you probably capture when you are in a requirements engineering meeting with your customer. Another source may be the requirements specification. I choose a more simple form for smaller projects – paper and a pen or the famous notepad application.

The meeting is about to discuss the address entity and related business rules.

„Address:“

    „Fields:“
        „Country, town, street, zip code, house number“

    „Conditions:“
        „The fields except country must not be empty“
        „The zip code must match formatting for the current country“

These notes discover a missing entity as the country may not be a free text field due to the second condition. Here is the note:

„Need entity country with additional formatting rules for zip codes“

I just had a look at wikipedia and there are plenty difficult rules you could enforce per country. See here. A more simple rule may be ignoring this condition for now, as the modeling doesn't yet support it and the application can't handle such rules. I assume much more complexity for this.

We begin with the first condition and let the second open for later versions. To best explain the condition I'll show the ready made UML activity diagram. It contains a black circle as a starting point of the 'action' that is performed to evaluate the condition for this business rule.

The black circle is followed by a flow to the first diamond shaped symbol. The decision is a diamond shape. The activity starts with a decision about 'Plz'. This is the zip code and the rule for it is as follows: The zip code must not be empty. If it is not empty the flow to the next diamond is chosen in the flow of the activity. If the zip code is empty the rule matches at the flow with additional text and squared brackets that define the matching rule. In this case the matching rule is on condition failure. So at the end all rules look the same and will flow to right on failure. When the last diamond succeeds, the values match the condition and the activity is ended with a 'success'.

The activity diagram used for the validation as a business rule:
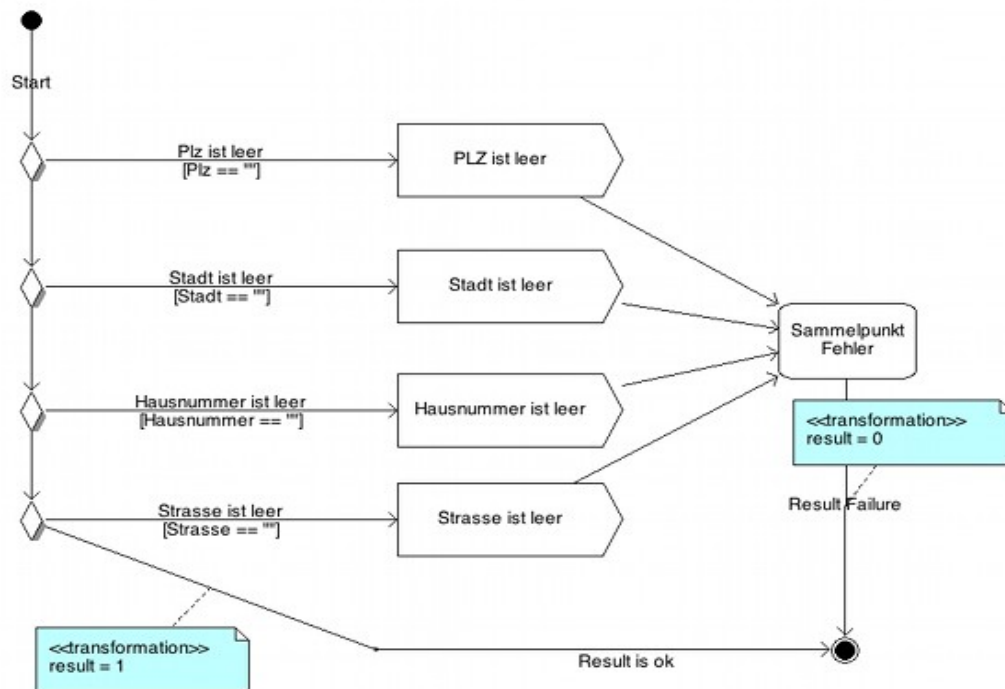
The result of a success is indicated by a 'return' value of result = 1, where as a failure has a value of 0 in 'result'.

So if any condition in the flow to the right matches, the activity follows that flow that first matched and enters one of these boxes with an arrow at the right. The box with this arrow indicates an UML send signal action. This kind of action is set by the 'kind' field in UML tab of the activity action dialog when you double clicking on the action symbol. This is the first thing you need to setup when you like to show a message box with a proper error message. Then you need to add properties in the properties tab to fully setup the action to be a message box with the proper error message. The following properties are required:

signal = showMsgBox, title = Error and msg = Your error message.

Without these properties nothing will happen or the application will behave wrongly. The name of the action with is used to show a message box id for documentation only. The real message is taken from the msg property. (This should be simplified by taking the name of the action. Also using stereotypes would further simplify the modeling work)

All message boxes are followed by one action as a next step. This action is a UML opaque action that currently does nothing, but directing the flow to one followup flow or a final flow on failure.

The final flow after an error sets the value in 'result' to 0 to indicate an error. The setting of a value is done by adding a transition rule as an assignment. As above the succeeding flow uses result = 1, the failure flow uses result = 0 as a transition.

These final flow transitions must be configured to let the application function properly. Without that you will get an error message about wrong UML modeling.

You see, that validating several values could be done with moderate amount of UML modeling. There is no restriction in amount of decisions and error messages and therefore much more complex rules could be modeled.

The UML model for this documentation is included in the source code distribution and in the binary samples distribution. For a reference consult that UML model.

Note: The modeling of simple rules may be done in that way, but more complex rules may be done in different ways as I have figured out that this is better done otherwise. The following could be done:

Create stereotypes for repeatedly used signal actions and modify the XSLT templates to support them. Or start creating a model to model transformation as an intermediate step. But this requires to define a UML import activity where you can model the workflow. See next chapter Using activities for code generation.

Using activities for code generation

In the release 1.0.2, you will have a new feature that let you do more sophisticated code generation tasks. The flexibility with the existing code generator was reduced to use only one XSLT template file that you had to choose every times. The new UML activity based code generator uses signal actions to execute code modules that the system provides. In the validator sample this is the message box. But the code generator activity required new functionality callable via the send signal action. See in chapter Extend the functionality of the core.

Available activity steps

The following new features have been implemented and provided:

- Write the XML representation of the application model into a parameter
- Write a parameter (string) into a file
- Read a file into a parameter (string)
- Make a XSLT transformation by given parameters for the stylesheet and the input file
- Make a XSLT transformation by given parameters for the stylesheet and the input as a parameter (string)
- Detect if a file is present
- Detect if a path is present
- Detect the running OS type

With these features I was able to create a code generator for a specific XSLT template and do different actions on the detected platform the activity runs. Currently the code generator activity can only be used when running against a PostgreSQL database. Using Sqlite currently fails at least on Debian (PPC) so I state Sqlite based code generation as buggy.

> Now the code generation action is included, but also a new version is modeled including new model validation features that **must** be imported as the templates using it.

The new features include a bigstring (UML) field to contain the validation messages and a checkbox to indicate model completeness. The XMI import templates will fill these fields to enable a review of modeling errors. This is useful if no OCL constraints are used in UML. I do not yet use any OCL.

The sample code generation workflow

The implemented sample workflow uses the Turbo Vision template to create a console based TUI application. It decides based on the running platform what the path to the XSLT template is. The path must probably adjusted as I have it only tested on my development machines and on a Windows XP virtual machine using the setup routine. Bear with me, if there are some remaining bugs, but I liked to release now. Here is an image how the workflow looks like:
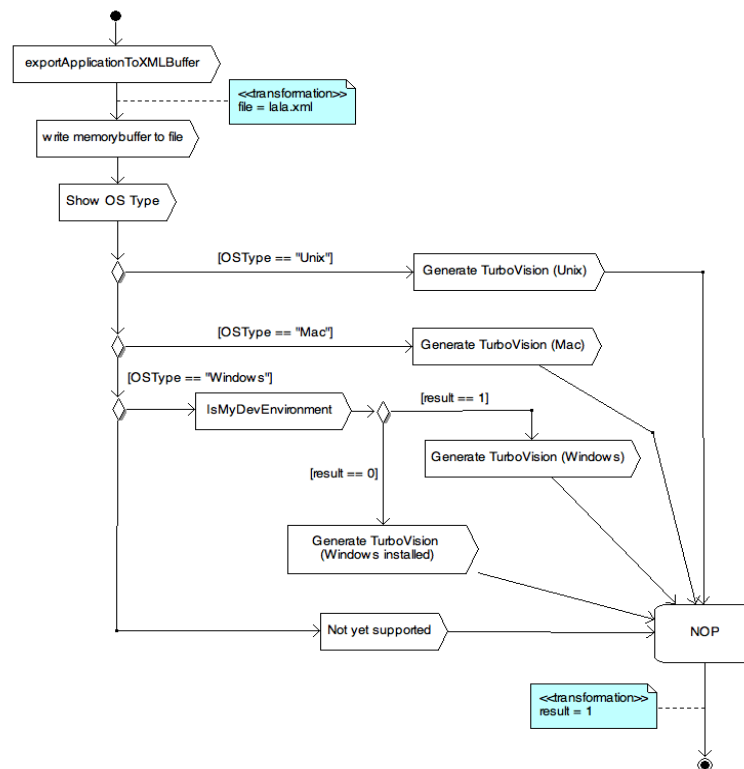
As a side note: I feel that creating such an activity takes a bit longer than using a scripting language or a domain specific language. Using a language requires more development. I currently don't implement such a language. Also note that there are some properties set in the model you don't see on the diagram. Take a look into the UML model in the second stage folder.

Printing

This chapter discusses the feature of printing. A business application usually requires to print some reports, thus it is an important feature yet missed.

Earlier, printing was an attempt using wxRepWrt, but that had one big downside – there is no designer available as I know. The new printing implementation benefits from a license change of the OpenRPT project and thus I decided to use it with very little effort. The implementation uses the lbExecuteAction action. That technically starts an external application.

For you the report designer is an easy way to define reports. To further speed up designing applications I plan to automatically create some sort of reports to be modified later, but that is a todo and may be an enhancement in the XSLT templates that are responsible to fill the configuration tables.

Preparing for printing

Before you could print anything, you need to add a table in the database from where you want to print reports. The following steps explain what you need to do to enable printing.

Creating the table report:

```
CREATE TABLE report
(
report_id integer NOT NULL DEFAULT  nextval(('report_report_id_seq'::text)::regclass),
report_name text,
report_sys boolean,
report_source text,
report_descrip text,
report_grade integer NOT NULL,
report_loaddate timestamp without time zone,
CONSTRAINT report_pkey PRIMARY KEY (report_id)
)
WITH (OIDS=TRUE);
ALTER TABLE report OWNER TO dba;
GRANT ALL ON TABLE report TO dba;
COMMENT ON TABLE report IS 'Report definition information';

-- Index: report_name_grade_idx

-- DROP INDEX report_name_grade_idx;

CREATE UNIQUE INDEX report_name_grade_idx
ON report
USING btree
(report_name, report_grade);
```

Then you need to fill some configuration tables with data per report that should be invoked by the user. First you must ensure having an action of the following type in action_types table:

The 'bezeichnung' (name of the type) does not matter, but will be helpful when configuring actions in the application 'lbDMF Manager'.

The field 'action_handler' must be 'instanceOflbExecuteAction' as this is the functor that creates the instance of the class responsible to execute external applications.

The field 'module' must be 'lbDMFBasicActionSteps' as it implements the action. Save the generated id for the next step.

The document uses as a sample the report to print a list of forms. Thus the table actions should get a record containing the following relevant data:

- The field 'name' may be 'Print forms of the actual application'.
- The field 'source' must be the foreign key pointing to anwendungen and thus must be 'anwendungid'.
- The field 'typ' must be '1' pointing to the 'Buttonpress' action in the field 'bezeichnung'.

The table 'action_steps' need the following entry:

- 'bezeichnung' may be 'Print'
- 'a_order_number' should be '1'
- 'what' must contain a call rptrender with some parameters as explained in the following

Sample for Windows:

C:\Programme\openrpt-3.1.0\RPTrender.exe

-databaseURL=pgsql://<database server>:5432/lbdmf

-username=<database user>

-passwd=<password>

-loadfromdb={ReportName}

-printerName={PrinterName}

-printpreview -close

-param=filter:int={anwendungid}

The parameters are as follows:

- -databaseURL specifies the type of database and where it is installed. Additionally it specifies the port and database name.
- -username and -passwd not commented here :-)
- -loadfromdb={ReportName} is a dynamic parameter and gets replaced.
- -printerName={PrinterName} is a dynamic parameter and gets replaced.
- -printpreview -close will directly start the preview mode and automatically closes the reporting application after printing or cancelation.
- -param=filter:int={anwendungid} is a dynamic parameter and gets replaced.

Dynamic parameters

The dynamic parameters for this action are retrieved from their values in the action_step_parameters table or from values in the field of the form containing the action button. This enables settings to be passed per application and values to be really depend on the data the form displays. So be aware to setup the ReportName and PrinterName name / value pair in the table 'action_step_parameter' pointing to the right entry for your action to be defined.

Also be aware to select the correct field when you refer to data from your application. The field must be in the select cause, but may not shown in case of hiding the field when the form has been opened from a master form.

Note: The application to be executed is not called in platform neutral manner. To enable platform neutral calling, one solution would be changing the action to become a workflow. This is a todo.

Extend the functionality of the core

To get more functionality, you need to read the upcoming document 'Developing XSLT templates'. It will explain how to get the internal representation from the UML XMI representation. Technically, this is a model to model transformation from UML to relational schema, where the schema itself is also an UML model (in that case the meta model). Also I plan a document about 'Extending functionality while keeping it dynamic and modular'.

There is really a need for this third document, as of the flexibility you will gain. For sample the message box I use in the UML sample is implemented as follows:

```
void LB_STDCALL lb_MetaApplication::msgBox(char* title, char* msg) { // ...
```

I can't call such methods dynamically. C++ doesn't has a reflection mechanism built in.

You see the title and the msg parameter above. It is an API functionality the core provides. They are related directly to the way it has to be modeled. So you either have to look into code or wait for the document to read how to extend the functionality. Be warned, this is not all you need to know. It is much more. For sample this function uses a dispatcher and a marshall mechanism to pass parameters. This is because of decoupling the GUI implementation from the application logic that uses it. A dynamic marshaling is used to send a **signal** in terms of the UML signal action.

The real implementation is a normal method, that could be used directly, as it has an interface definition, but it is additionally wrapped by a de-marshaling function to enable dynamic invocation with a marshaling function (the API function above). This was designed prior to those activity diagram features and thought to also enable marshaling a function call over the network. (In my predecessor project before year 2000 whose parts are in the code)

Of course, the networked version with my own TCP stack is somewhat slow – very slow and is no option for production code. It is therefore also not yet used in the main sample application. Other libraries such as ACE or it's CORBA implementation may more likely an option.

This is a good sample of what to think about when enhancing the function of the application. Think twice before hacking, it may be good when having a new feature also usable in UML :-) To be backward compatible, only the de-marshaler and the real implementation should be in a plugin not modifying existing interfaces. That way, older implementations would benefit from it. In the current release, I have not followed this advice while implementing a user feedback mechanism. So you need to update completely.

The code:

```
lbErrCodes LB_STDCALL lb_wxGUI::msgBox(char* windowTitle, char* msg) {
    if (!splashOpened) {
        wxMessageDialog dialog(NULL, msg, windowTitle, wxOK);

        dialog.ShowModal();
    } else {
        if (pendingMessages == NULL) {
            REQUEST(getModuleInstance(), lb_I_String, pendingMessages)
            *pendingMessages = "";
        }

        *pendingMessages += "\n";
        *pendingMessages += windowTitle;
        *pendingMessages += "\n";
        *pendingMessages += msg;
    }
    return ERR_NONE;
}
```

This function is part of a collection in a platform neutral wrapper class having a plain C++ interface in mind. You will see the usage of wxWidgets related code, but the interface doesn't do so, thus it is platform neutral. The wrapping method is implemented in the lb_wxFrame class that gets the lb_I_wxGUI based instance when it is created within the lb_I_wxGUI implementation. This is a point of refactoring :-)

The frame therefore implements many wrapper functions to enable dynamic calls, but it should forward these calls to the lb_I_wxGUI based instance. Thus neither the lb_I_wxFrame implementation should implement wrapper functions when it implements API functions.

The wrapper functions may be factored out into plugins that are **using** the API's, but don't mix wrapper and API. It is a big refactoring task. The wrapper then only need to implement the lb_I_EventHandler interface to provide dynamic calls.

The de-marshaling method:

```
lbErrCodes LB_STDCALL lb_wxFrame::showMsgBox(lb_I_Unknown* uk) {
    lbErrCodes err = ERR_NONE;

    UAP(lb_I_Parameter, param)
    UAP_REQUEST(manager.getPtr(), lb_I_String, parameter)
    UAP_REQUEST(manager.getPtr(), lb_I_String, msg)
    UAP_REQUEST(manager.getPtr(), lb_I_String, title)
    QI(uk, lb_I_Parameter, param)

    parameter->setData("msg");
    param->getUAPString(*&parameter, *&msg);
    parameter->setData("title");
    param->getUAPString(*&parameter, *&title);

    gui->msgBox(title->charrep(), msg->charrep());

    return err;
}
```

This function is responsible for de-marshaling the parameters passed in a parameter container. The first code sample lb_MetaApplication::msgBox is the counterpart to this – the marshaling function. Note: See the refactoring nodes above.

Why do I have chosen such a scheme?

The method as follows is only for the encapsulation of the underlying GUI framework. The user of the independent GUI API shouldn't need to know, what GUI framework is used.


```
lbErrCodes LB_STDCALL lb_wxGUI::msgBox(char* windowTitle, char* msg)
```


The method as follows enables marshaled calls like COM. But if this function is not registered to be used, nothing happens. This is useful if no message box should be shown, or there is simply no UI nor a GUI. A simple console application may the case for this. A log message could be written instead.


```
lbErrCodes LB_STDCALL lb_wxFrame::showMsgBox(lb_I_Unknown* uk)
```


When you extend the functionality and like to enable calling it by designing it in UML, then you need a de-marshaling function to what ever you implement. To use your function in an optimized application beyond a prototype, then write the real function as a usual method and call it in the de-marshal method and / or use it later directly. Don't forget to register the de-marshaling method.

Extend the the core with plugins

The following chapter describes extension of the core by using plugins that are completely isolated (not modifying existing interfaces) and thus may be usable in older versions.

One sample of such an extension is the login functionality provided by a plugin. It is just a demonstration. The second sample is the new graphical designer that is depicted in the chapter CAB DevExpress prototyping. This shows that the software can be extended by complete modules of independent functionality such as a different design entry.

There are the following questions to be answered:

How can existing code benefit from new plugins?

When do they get activated?

These questions can be answered by samples as follows:

The login handler extends the application by adding another menu entry into the file menu. To do this, the plugin must execute some code automatically. This is done in the plugin helper class that is registered in the plugin module. The plugin module class will be found by the plugin manager.

Let start with the handler class definition:

```
class lbLoginHandler :
        public lb_I_Unknown,
        public lb_I_LogonHandler, // marker interface
        public lb_I_EventHandler {
public:
                lbLoginHandler();
                virtual ~lbLoginHandler();

                DECLARE_LB_UNKNOWN()

                lbErrCodes LB_STDCALL registerEventHandler(lb_I_Dispatcher* disp);
                lb_I_Unknown* LB_STDCALL getUnknown();
                lbErrCodes LB_STDCALL runLogin(lb_I_Unknown* uk);

                wxWizard *wizard;
                wxWizardPageSimple *page1;
};
```

The important function in that class is the event registration that makes functions available to the dispatcher (registerEventHandler). The implementation of the class knows what functions to be made available, thus it registers these functions. The second important function here is the getUnknown method. It is there, because the interface doesn't inherit from lb_I_Unknown. It is a mix in interface to enable any class to provide event handlers. In this sample the lb_I_Unknown interface is also implemented.

The cast method to return the lb_I_Unknown interface is required if the instance is passed around with the lb_I_EventHandler interface. I do not yet have any better solution. The issues have to do with multiple inheritance of the same base interface if I would enable the features from lb_I_Unknown in the lb_I_EventHandler interface.

```
lb_I_Unknown* LB_STDCALL lbLoginHandler::getUnknown() {
      UAP(lb_I_Unknown, uk)
      queryInterface("lb_I_Unknown", (void**) &uk, __FILE__, __LINE__);
      uk++;
      return uk.getPtr();
}
```

The event handler registration method:

```
lbErrCodes LB_STDCALL lbLoginHandler::registerEventHandler(lb_I_Dispatcher* disp) {
      disp->addEventHandlerFn(this,
      (lbEvHandler) &lbLoginHandler::runLogin, "RunLogin");

      return ERR_NONE;
}
```

The implementation runLogin:

```
lbErrCodes LB_STDCALL lbLoginHandler::runLogin(lb_I_Unknown* uk) {
      wizard = new wxWizard(NULL, -1, _T("Anmeldung via Plugin"));
      page1 = new wxWizardPageSimple(wizard);
      wxStaticText *text = new wxStaticText(page1, -1, _T("Melden Sie sich nun
an.\n"));
      wxSize size = text->GetBestSize();
      wxLogonPage *page2 = new wxLogonPage(wizard);

      page2->init(NULL);

      wxAppSelectPage *page3 = new wxAppSelectPage(wizard);
      page2->setAppSelectPage(page3);
      page1->SetNext(page2);
      page2->SetPrev(page1);
      page2->SetNext(page3);
      page3->SetPrev(page2);

      wizard->SetPageSize(size);

      if ( !wizard->RunWizard(page1) )
      {
        wizard->Destroy();
        return ERR_NONE;
          }

      wizard->Destroy();

      return ERR_NONE;
}
```

The registration and implementation alone is not enough to get the code functioning. It must be hooked into a menu. Therefore the plugin helper is used to make the hook in it's auto run method. The first part tells that it does something in auto run. This can later be used to allow auto run selectively by configuration to avoid running all auto runnable plugins.

Activating auto run:

```
bool LB_STDCALL lbPluginLoginWizard::canAutorun() {
      return true;
}
```

The auto run method looks like this:

```
lbErrCodes LB_STDCALL lbPluginLoginWizard::autorun() {
    lbErrCodes err = ERR_NONE;
    UAP_REQUEST(getModuleInstance(), lb_I_MetaApplication, meta)
    UAP_REQUEST(getModuleInstance(), lb_I_EventManager, ev)
    UAP_REQUEST(getModuleInstance(), lb_I_Dispatcher, disp)

    int lEvent;
    ev->registerEvent("RunLogin", lEvent);

    lbLoginHandler* hdl = new lbLoginHandler();
    QI(hdl, lb_I_Unknown, loginHandler)
    hdl->registerEventHandler(*&disp);

    char* file = strdup(_trans("&File"));
    char* entry = strdup(_trans("Login via &Plugin\tCtrl-P"));
    meta->addMenuEntry(file, entry, "RunLogin", "");

    free(file);
    free(entry);
    return err;
}
```

The function is called once in the initialization phase of the plugin manager. Thus it gets it's menu at the right place. Writing a complete plugin is beyond this documentation, please look at existing plugins like the one presented here. It is located in the Test/GUI/wxPlugins directory.

The auto run mechanism can be used and is used to bootstrap the main sample application by setting up the database in a clean installation. Extending the application for UML modeling purposes can be done as above, but simply omitting the menu creation steps. Only the instance of the event handler must get alive and the registerEventHandler method must be called. Good samples for that are the lbDMFBasicActionSteps handlers in the Plugins directory of the AppDevelomnetDemo/DynamicApp directory.

Note: The code above doesn't use platform specific code to create the menu entries. It is therefore possible to create handlers that interact with the GUI not knowing what GUI is used. In this case the wxWizardPage and similar stuff used is why the plugin is located in the wxPlugins directory. The main Plugins directory even contains wx stuff (database forms) but this is a refactoring issue. Also users of the wxPlugin login handler don't need to rely on wxWidgets, they only rely on the dispatcher yet.

Provide installation mechanism for plugins

Another issue with plugins is the problem of the first or initial run on a fresh system. The plugin may need some initialization or database schema setup.

For this purpose, an implementation of a plugin module (the SO/DLL) can provide installation instructions in the install() method. For example, the graphical designer registers the forms and the menu configuration in that method on an optional means.

The plugin module class looks like the following:

```
class lbPluginModulewxSFDesignerBase : public lb_I_PluginModule {
public:

        lbPluginModulewxSFDesignerBase();
        virtual ~lbPluginModulewxSFDesignerBase();

        DECLARE_LB_UNKNOWN()

        void LB_STDCALL initialize();
        void LB_STDCALL install();

        DECLARE_PLUGINS()
};
```

The implementation of the install() method may look like this code (some log messages are removed):

```
void LB_STDCALL lbPluginModulewxSFDesignerBase::install() {
        lbErrCodes err = ERR_NONE;

        UAP(lb_I_Database, database)
        UAP_REQUEST(getModuleInstance(), lb_I_MetaApplication, meta)

        char* dbbackend = meta->getSystemDatabaseBackend();
        if (dbbackend != NULL && strcmp(dbbackend, "") != 0) {
          UAP_REQUEST(getModuleInstance(), lb_I_PluginManager, PM)
          AQUIRE_PLUGIN_NAMESPACE_BYSTRING(lb_I_Database, dbbackend,
                              database, "'database plugin'")
        } else {
          REQUEST(getModuleInstance(), lb_I_Database, database)
          if (database == NULL) return;
        }

        const char* lbDMFPasswd = getenv("lbDMFPasswd");
        const char* lbDMFUser   = getenv("lbDMFUser");

        if (!lbDMFUser) lbDMFUser = "dba";
        if (!lbDMFPasswd) lbDMFPasswd = "trainres";

        database->init();
        if (database->connect("lbDMF", "lbDMF", lbDMFUser, lbDMFPasswd) != ERR_NONE)
          return;

        { // Scope for UAP's
          UAP(lb_I_Query, q)
          UAP_REQUEST(getModuleInstance(), lb_I_String, sql)
          q = database->getQuery("lbDMF", 0);
          *sql = "select id, handlerinterface, namespace, beschreibung from
                 formulartypen where namespace = '";
          *sql += "FixedForm_SFDesigner_Anwendungen'";

          q->skipFKCollecting();
          err = q->query(sql->charrep());

          if (err == ERR_DB_NODATA) {
                  err = q->first();
                  if ((err == ERR_NONE) || (err == WARN_DB_NODATA)) {
                          _LOG << "No modification to dynamic formular"
                                  << "'Anwendungsdesigner' definition needed." LOG_
                  } else {
          ...
```