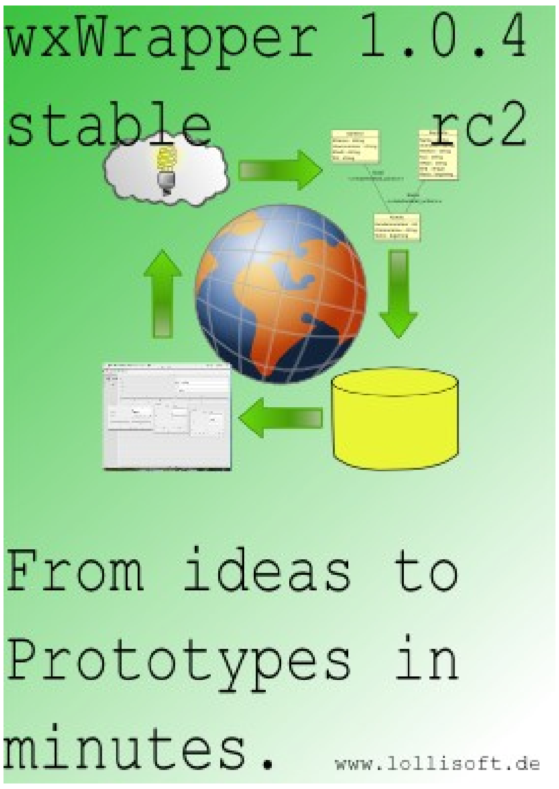


Create database applications fast

Documentation

Create database applications in minutes to hours

$Revision: 1.13 $

Table of contents

Introduction

Creating database applications usually seems to be easy. A simple CD cataloging system for sample does not have very much tables, so you start creating them manually in the database administration tool. Then with some tools like Open Office Base, you probably have no problems to create the first set of masks for such an application.

But what is if you have to develop a complicated CRM system with hundreds of forms and corresponding tables?

When you spot the fact that much of the forms are systematically the same, but with changes only in what they show (the fields and their SQL queries), then you probably ask your self, how can I save time or what a boring job.

With this project you will have a starting point in creating database applications faster. The method to use design tools is a great help. Those design tools eases the development because you begin modeling your database view in combination with some informations for the form design.

Also the system helps you to present a first solution faster what is called the prototype. With that prototype you can discuss with your customer in more detail, because they see a real working application, thus maybe see gaps in the logical design very early.

Also with the prototype you can test the application before any line of code has been written with a real database.

The tool also helps you to make decisions like what framework should be used or what language has to be used done very late at a point where you do the step from the prototype to code generation.

With 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 new code should adapt.

Of corse you also could develop your own code generators and probably not only targeting plain database backends. It will be possible for sample to generate code for Java, Hibernate or what ever.

One of the important point is that you have the choice to export to UML2.

Concept

The prototype application can either created dialog based what is a bit more complex, but it can be attended in more detail settings were may not yet supported by additional modeling tools. Or you use a designer that understands to create XML files that principally could be imported.

This software package is designed to use a UML modeling tool (BoUML) that I support with an import function (template). Note: At the current time this tool is no longer available for free (except forks). Other UML tools are not tested.

There also could be used other tools that supports a XML format. The XML format must correspond to the modeling needs you require and the XSLT template must translate it to the internal representation. If there is a missing feature in the prototype runner you want to have, create a plugin or extend the code and find a way to model it as you like. Then adapt the existing templates to understand the translation from and to the internal representation.

At the same time of exporting to UML, a code generator button is located in the application form of the main sample application (wxWrapper). This button is an action that is modeled in the UML and thus it could be a starting point on how the modeling artifact is implemented (translation).

The action is a core element to let you extend the current features. For sample there is a new feature in the action it self to behave like a workflow, thus you could model a validation workflow to integrate a business logic into a form.

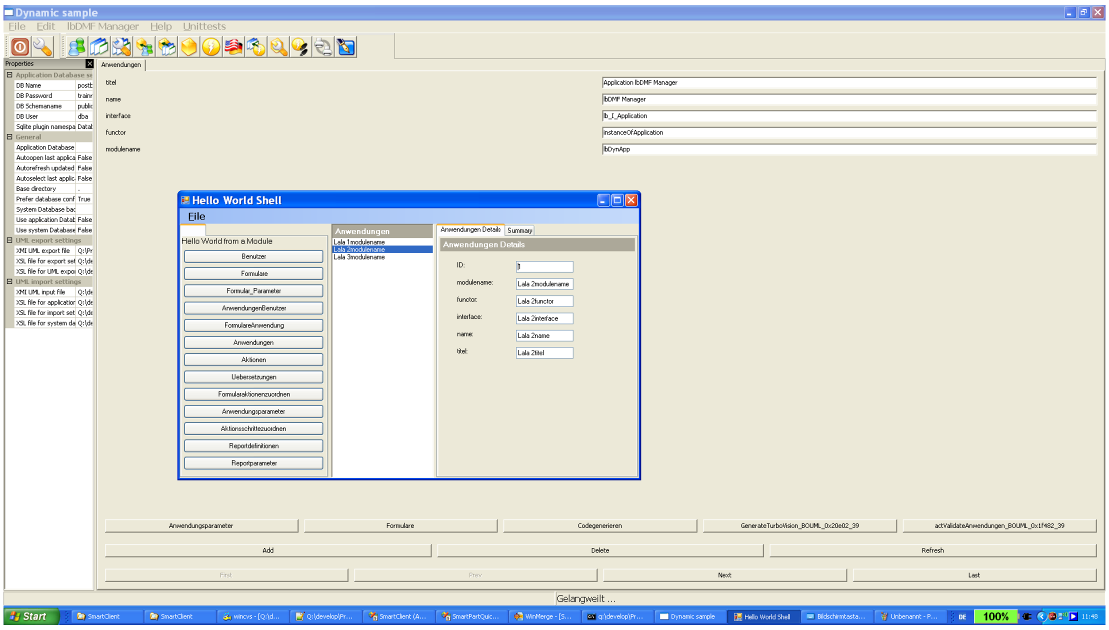
The business logic feature also shows you how to use modeling artifacts in conjunction with plugins or 'callbacks'. It is the most powerful opportunity to extend the features of this project.

The new code generation activity features are the most powerful ones I have implemented for this release. Despite of BoUML development has stopped, it is still downloadable and thus usable, but I think of a new one to support. See: 'Using activities for code generation'

The chapter 'Extend the functionality of the core' explains how a new modeling feature could be integrated into the application before you use it in your UML models. It explains creating a plain method, and a demarshalling 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 demarshaller passes to the plain method.

Quickstart

To get a first grasp of the capabilities the application has I will guide you through a short list of steps to get a result. 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. See here for an article what do you get. Here is a short preview:



Application Database se	
DB Name	postb
DB Password	trainr
DB Schemaname	public
DB User	dba
Sqlite plugin namespa	Datal
General	
Application Database	
Autoopen last applica	False
Autorefresh updated	False
Autoselect last applic	False
Base directory	.
Prefer database conf	True
System Database bac	
Use application Datab	False
Use system Database	False
UML export settings	
XMI UML export file	Q:\Pr
XSL file for export set	Q:\de
XSL file for UML expor	Q:\de
UML import settings	
XMI UML input file	Q:\de
XSL file for applicatior	Q:\de
XSL file for import set	Q:\de
XSL file for system da	Q:\de

titel	Application lbDMF Manager
name	lbDMF Manager
interface	lb_I_Application
functor	instanceOfApplication
modulename	lbDynApp

Lala 1modulename
Lala 2modulename
Lala 3modulename

ID:	1
modulename:	Lala 2modulename
functor:	Lala 2functor
interface:	Lala 2interface
name:	Lala 2name
titel:	Lala 2titel

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.

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 I have 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 I have 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. I cannot choose one database backend because you may like another.

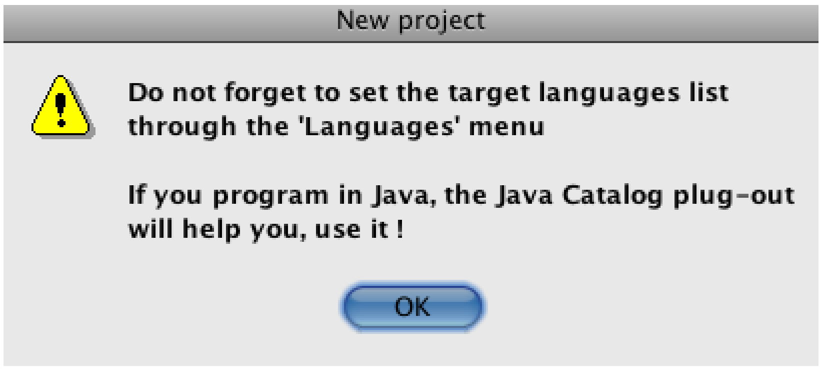
Please follow my blog. I will post about the project. There you will also see other (not yet available) templates and stuff around my project.

Start with Modeling

In this chapter you 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.

Creating an UML model

To create an UML model you start the included and installed software BoUML. In the 'Project' menu you select 'New' and enter the project name. Enter 'BoUML'. You then will see the following warning message:

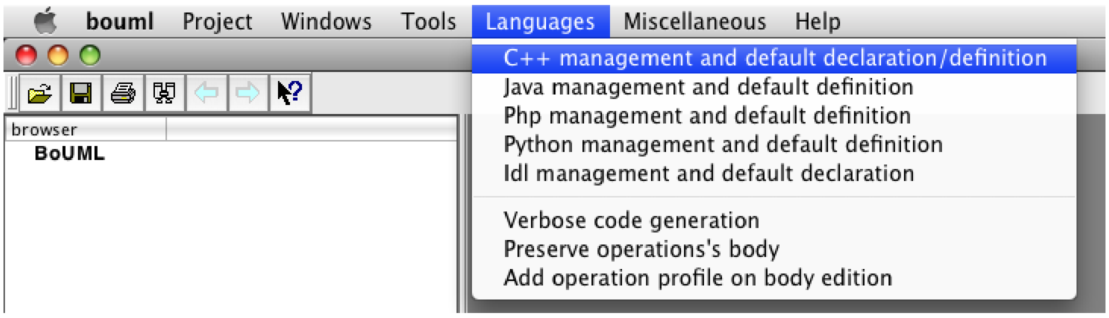


New project

Do not forget to set the target languages list through the 'Languages' menu

If you program in Java, the Java Catalog plug-out will help you, use it !

Close the window and select 'C++' as I prefer always.:


bouml
Project
Windows
Tools
Languages
Miscellaneous
Help
C++ management and default declaration/definition
Java management and default definition
Php management and default definition
Python management and default definition
Idl management and default declaration
Verbose code generation
Preserve operations's body
Add operation profile on body edition
browser
BoUML

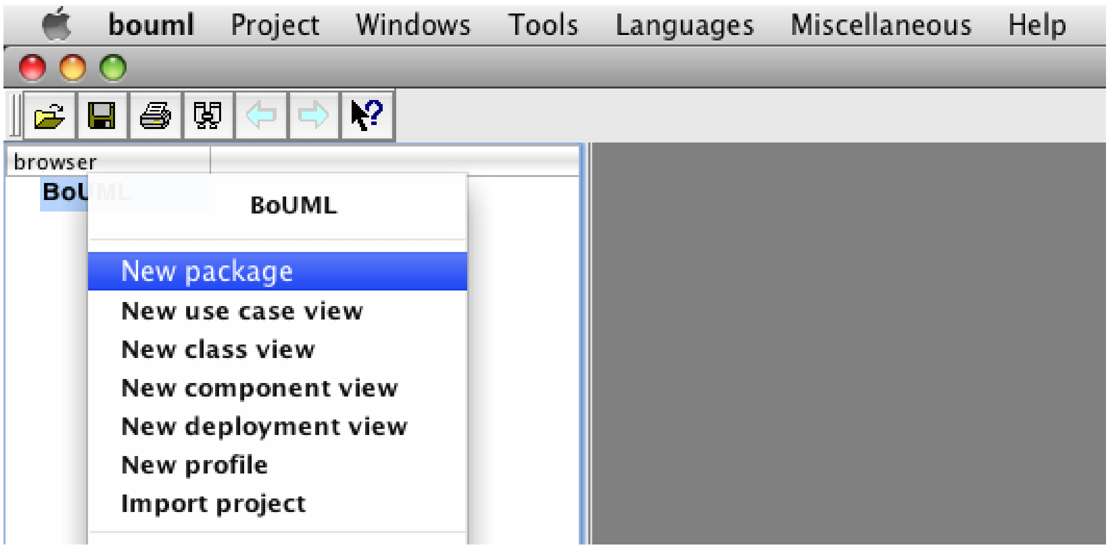
This setting relates to the internal code generation of the UML modeller and doesn't matter here.

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:



Multible 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'

Create a class view

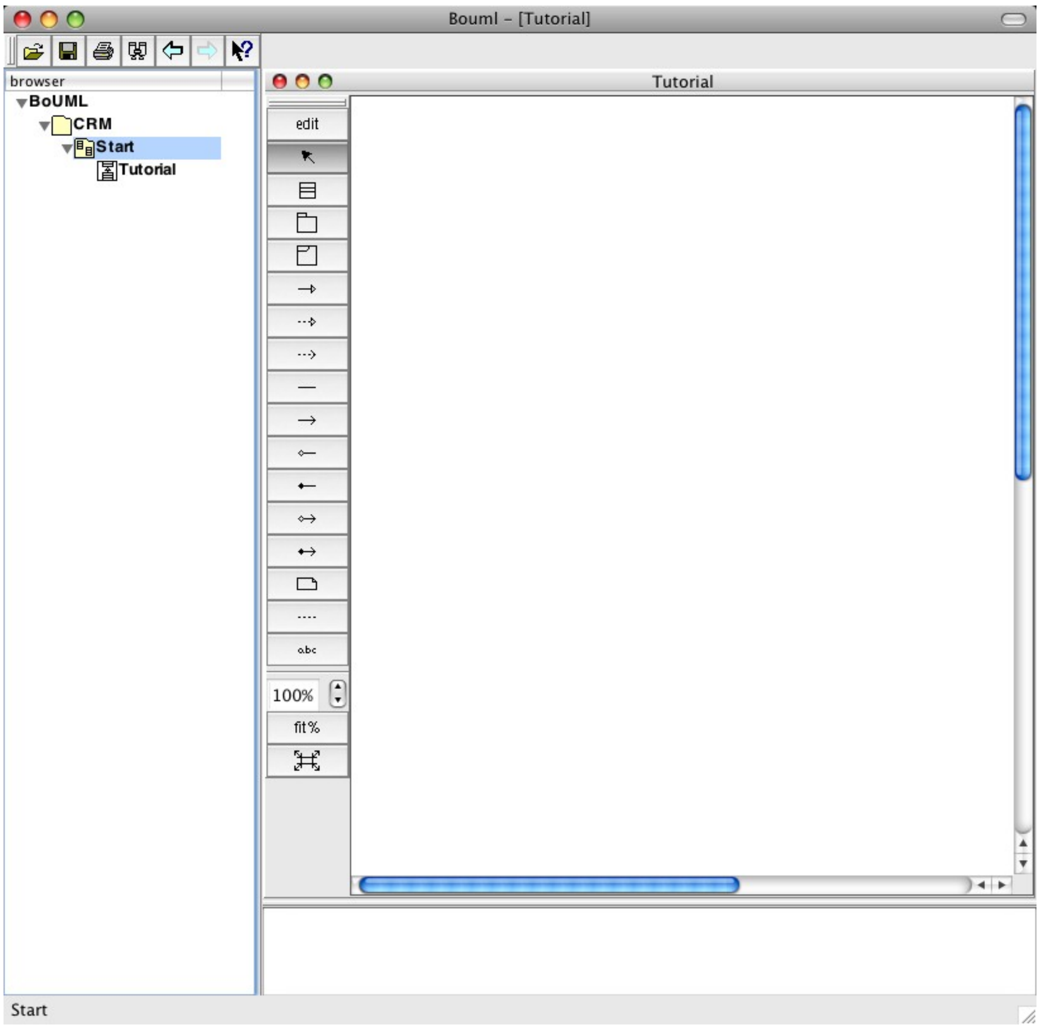
After you have created a package, you create in there a class view. See the picture above to spot the next menu entry for creating class views.

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:



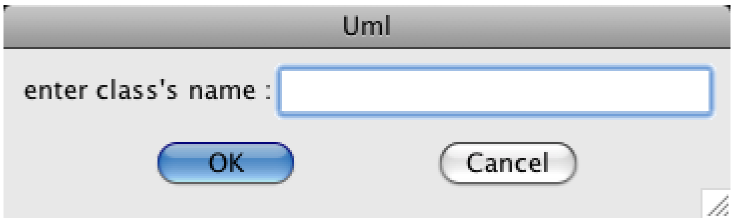
Start modelling

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.

Create some classes

You create new classes with the 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.

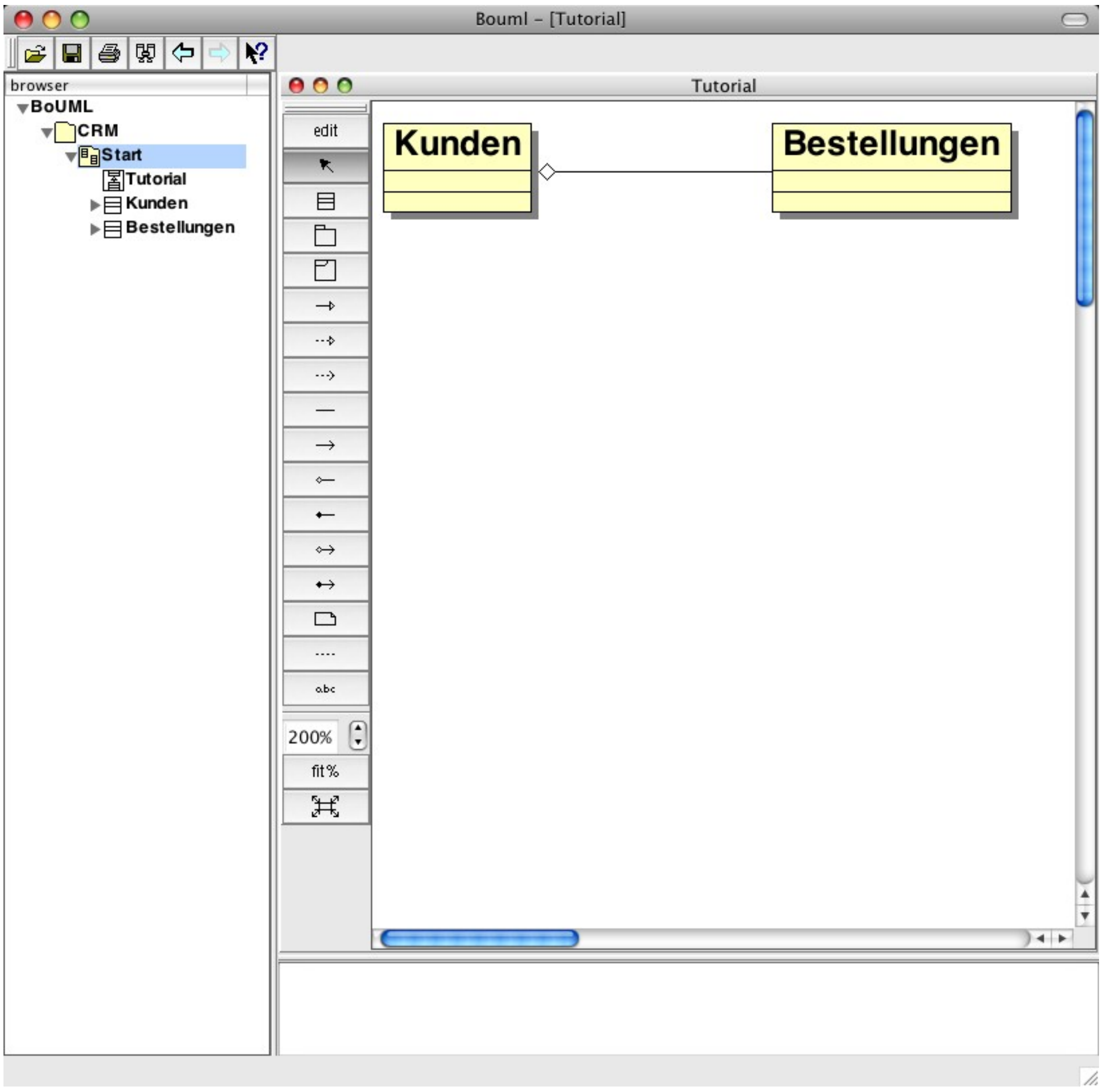
In the following sample we will create a class named 'Kunde' (customer) wich is referenced from a second class named 'Bestellungen' (orders).

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 archive the same result, but those are not yet supported in the prototype application import method.

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:

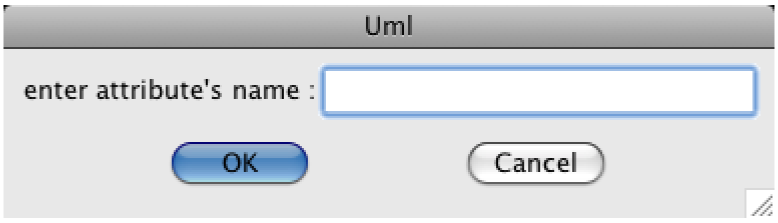
- Create attributes
- Create operations

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

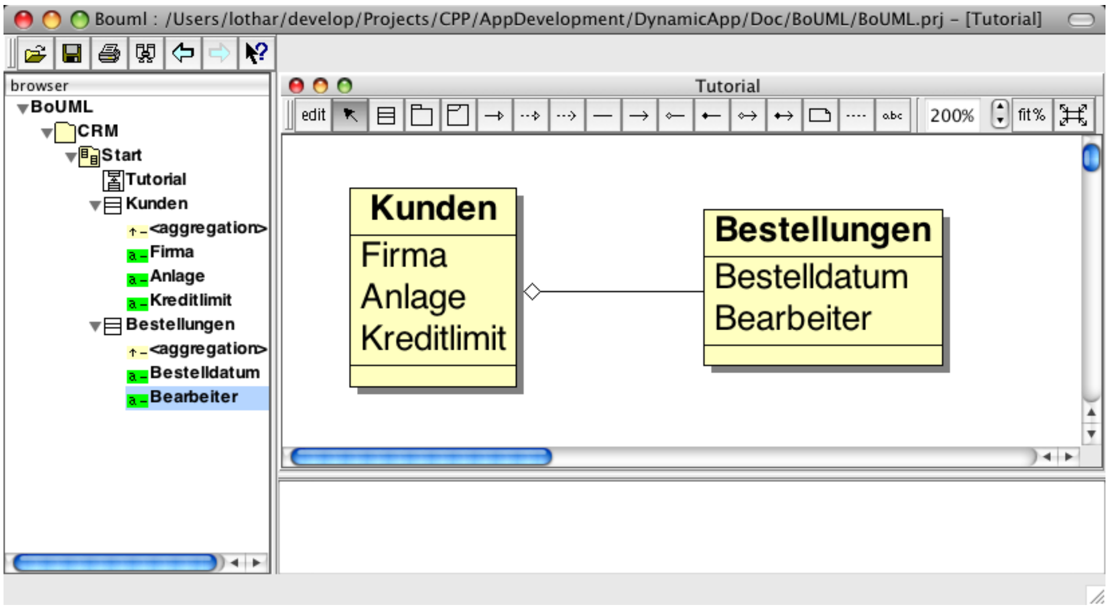
Kreditlimit (credit limit): float

The class orders becomes the following attributes:

Bestelldatum (order date): date

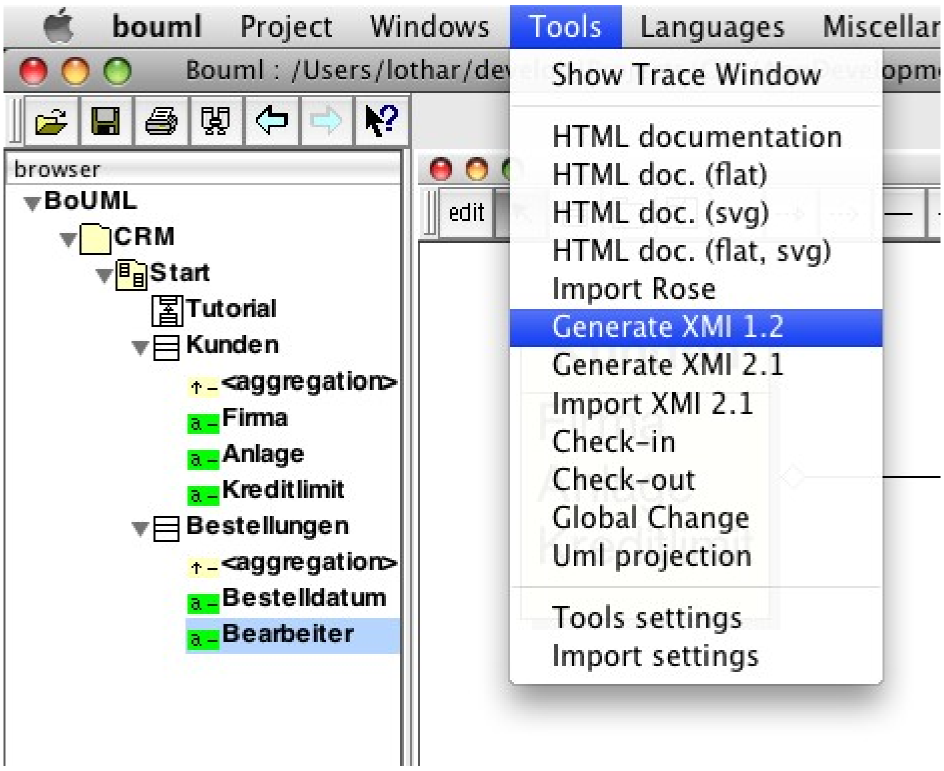
Bearbeiter (sales rep.): string

Now your UML model looks like this:



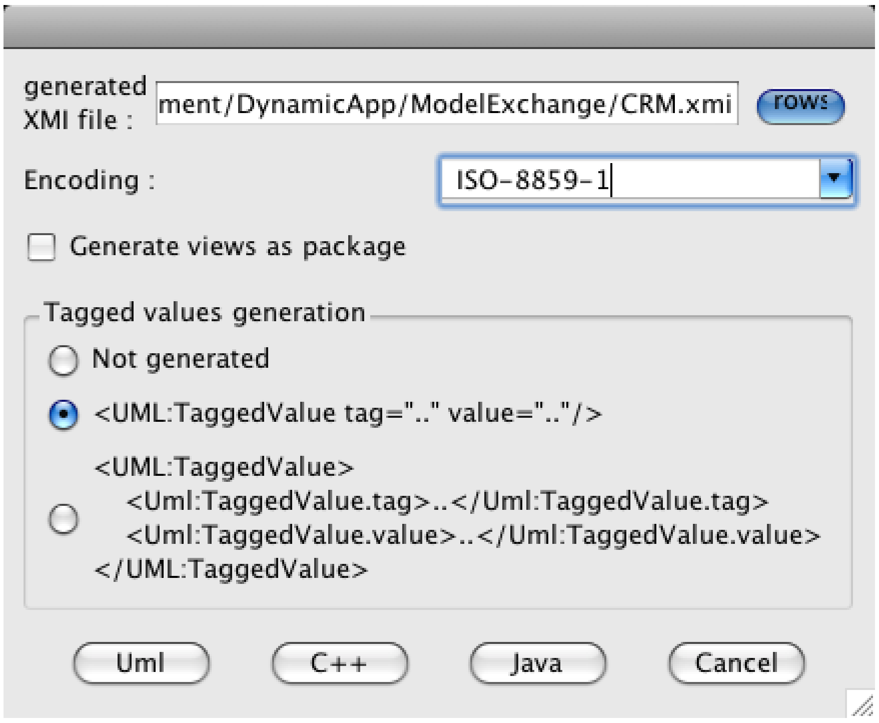
Export the UML model

After you have finished your 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

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>

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.

Import UML model

To use an UML model created in BoUML, it is required to import it. Therefore you start the main application (wxWrapper). On the left side are some settings you need for the import. Fold in the first two groups. You don't need them now. Also note that these properties have been reordered, so I have removed the picture for now. I'll list up the important ones.

In 'Application Database Settings' you define your credentials, schema and name of the database. Also there is a Sqlite plugin namespace that may be relevant. There is no test yet to validate your settings.

The next group is General. In that are important switches to select the database libraries used to talk with the database. For Sqlite, you have to copy the Sqlite plugin namespace in case it is missing in the following settings:

To use Sqlite for the target application (you are designing), set 'Use application Database backend' to true and fill out the 'Application Database Backend' field with the copied value from above.

If you choose to use Sqlite for the system database, you may do the same for the following fields:

Set 'system Database backend' to true and copy the namespace value also to 'System Database backend'.

Important: If you have a fresh install, then the system database points to Sqlite. That is because the ODBC based database settings may be missing, or are wrong. Thus the software chooses to fallback to a Sqlite variant. In the most cases you do not change the system database settings, but if you are working on different machines, you need to change to **not** use system Database backend.

Set the 'Prefer database configuration' to true, because if not, the software chooses to use a locally cached version from a file as it is faster. There is also not yet any check if the database version is more recent to inform the user about a new version.

The next group is 'UML import settings', where you define your source and how to import.

Choose the 'Application database backend type' to be PostgreSQL exactly or Sqlite exactly. These settings configure the XSLT transformation steps to target PostgreSQL or Sqlite. The same is for 'Sytem database backend type'. You do not need to enter the backend namespace from above, as the settings are evaluated in XSLT templates to switch generation of SQL code.

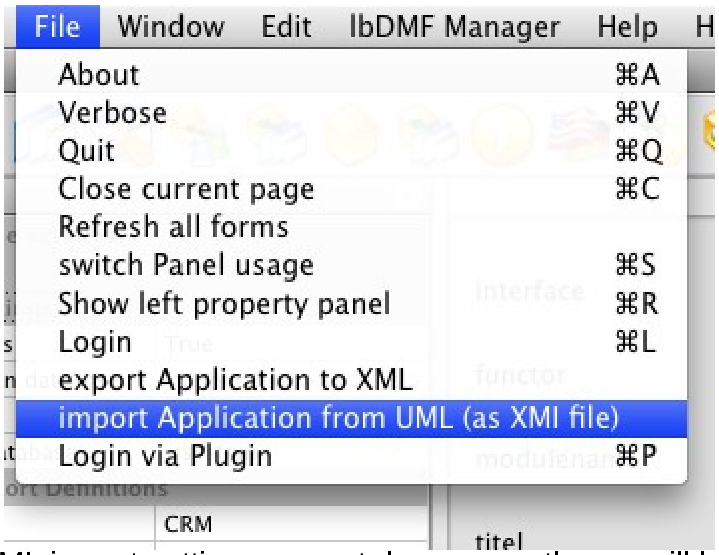
Choose the file you have created with BoUML in <XMI UML input file>. It will be used in the import procedure.

To correctly import the application model you need to know of what format your XMI file is. Shortly we have exported the UML model in XMI 1.2 format, thus the following settings are to be done:

<XSL file for application database>:	xmi1.2_2SQLScript.xsl
<XSL file for import settings>:	XMISettings.xsl
<XSL file for system database>:	xmi1_2_2_lbDMFSQLScript.xsl

These files are in the folder XSLT_Templates/XMIToDMF. The file XMISettings.xsl is to be entered manually in that directory if it doesn't exist. In that file are settings that controls how the import is done as documented above.

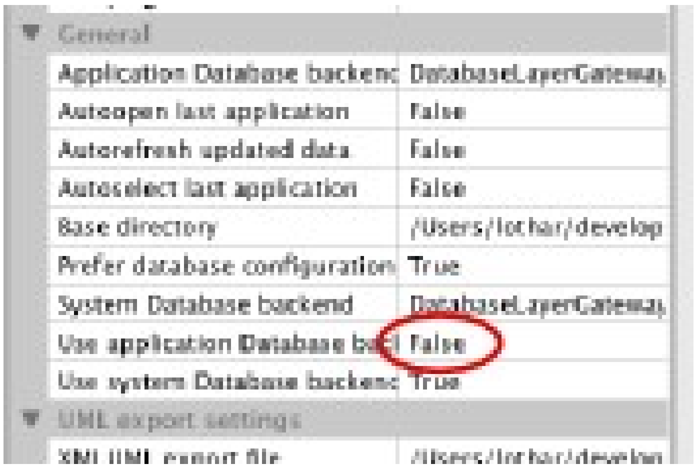
If you have entered the settings, you are ready for the import. In the following picture sou see how to start the import:



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. If these fields are set, you get another warning about not telling the software to **overwrite** the XMISettings.xsl file. This is because you may have added manually some other settings in that file. In that case you know what you do and click to no. 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, or even may fail in case of relations not captured in the model, but exist in the database. If you like to use a production database you may read chapter You have an existing database.

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 reverse engineered that database.

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.



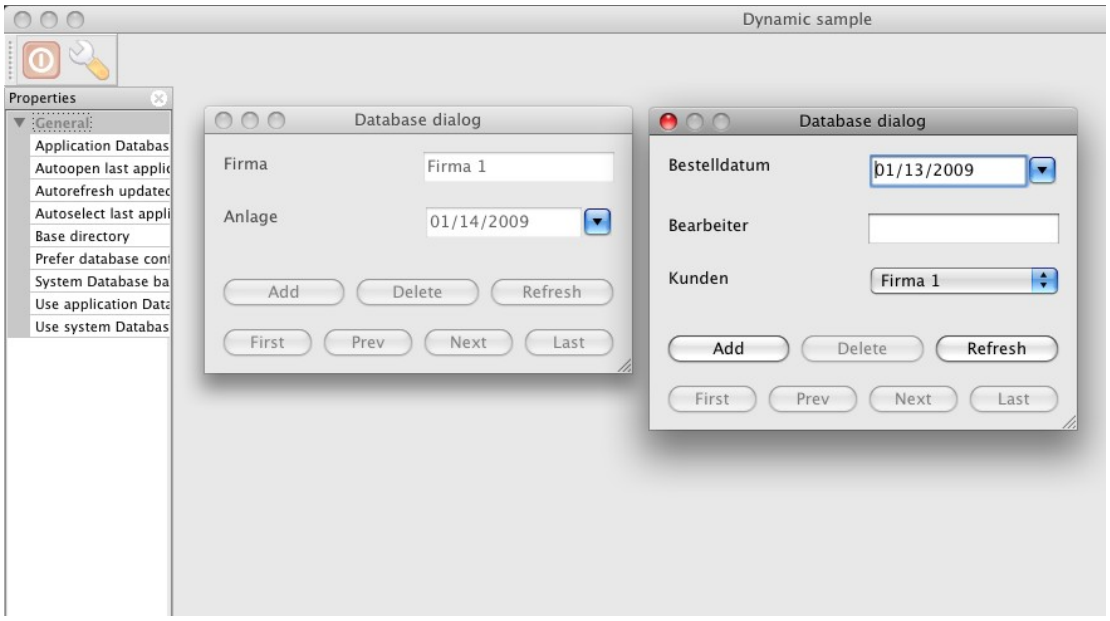
General	
Application Database backenc	DatabaseLayerGatewa
Autoopen last application	False
Autorefresh updated data	False
Autoselect last application	False
Base directory	/Users/lothar/develop
Prefer database configuration	True
System Database backend	DatabaseLayerGatewa
Use application Database ba	False
Use system Database backenc	True
UML export settings	
XML UML export file	/Users/lothar/develop

But using an existing database is explained later in more detail.

After you have imported the UML model ensure you have unchecked the menu entry 'Autoload application in the 'Edit' menu. Quit the application, start it again and login to the newly created prototype. In our sample the user is 'user' and the password will be 'TestUser'. The application you start will be 'CRM'.

The first prototype

The first prototype looks like the following picture if you click once in 'File' 'switch Panel usage' and have entered some data. The application does not contain a second toolbar because the UML modeler can't directly model this.



Firma: Firma 1
Anlage: 01/14/2009

Bestelldatum: 01/13/2009
Bearbeiter:
Kunden: Firma 1

You have an existing database

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 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

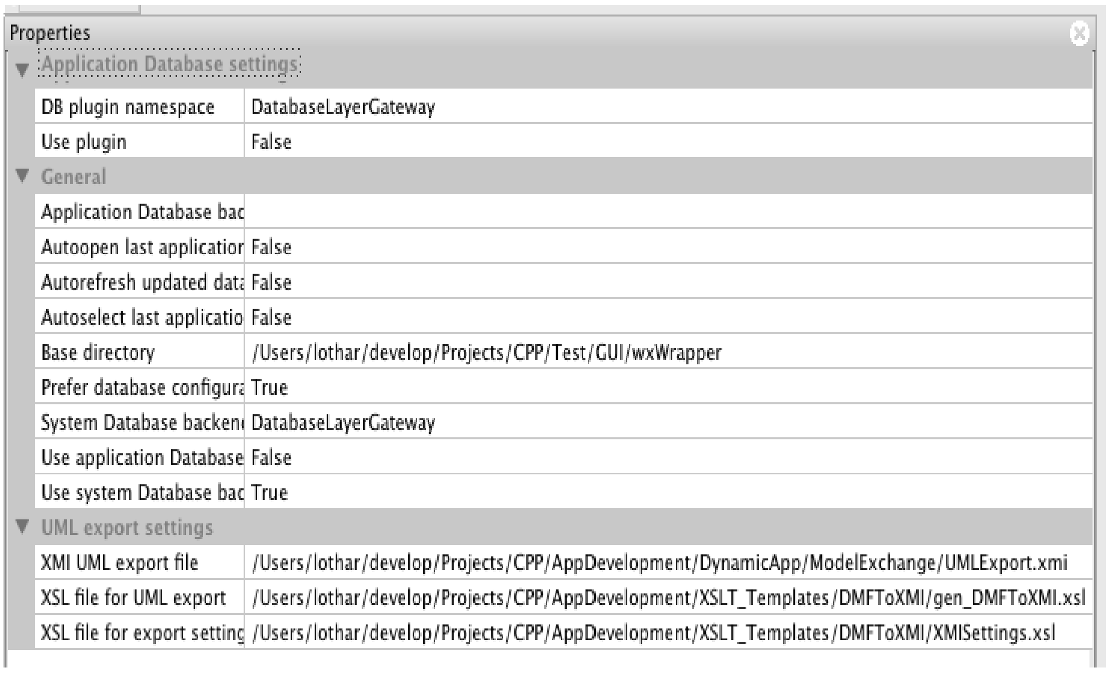
You export the model as of described here: Export the UML model and import it in to the prototype runner. Thereafter you check the database access settings in the application settings ('Anwendungsparameter' in 'Anwendungen') in the newly created application model. The important settings are 'DBName', '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. Look here: 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.


Properties

▼ Application Database settings	
DB plugin namespace	DatabaseLayerGateway
Use plugin	False
▼ General	
Application Database bac	
Autoopen last applicatior	False
Autorefresh updated data	False
Autoselect last applicatio	False
Base directory	/Users/lothar/develop/Projects/CPP/Test/GUI/wxWrapper
Prefer database configura	True
System Database backen	DatabaseLayerGateway
Use application Database	False
Use system Database bac	True
▼ UML export settings	
XMI UML export file	/Users/lothar/develop/Projects/CPP/AppDevelopment/DynamicApp/ModelExchange/UMLExport.xmi
XSL file for UML export	/Users/lothar/develop/Projects/CPP/AppDevelopment/XSLT_Templates/DMFToXMI/gen_DMFToXMI.xsl
XSL file for export setting	/Users/lothar/develop/Projects/CPP/AppDevelopment/XSLT_Templates/DMFToXMI/XMISettings.xsl

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.

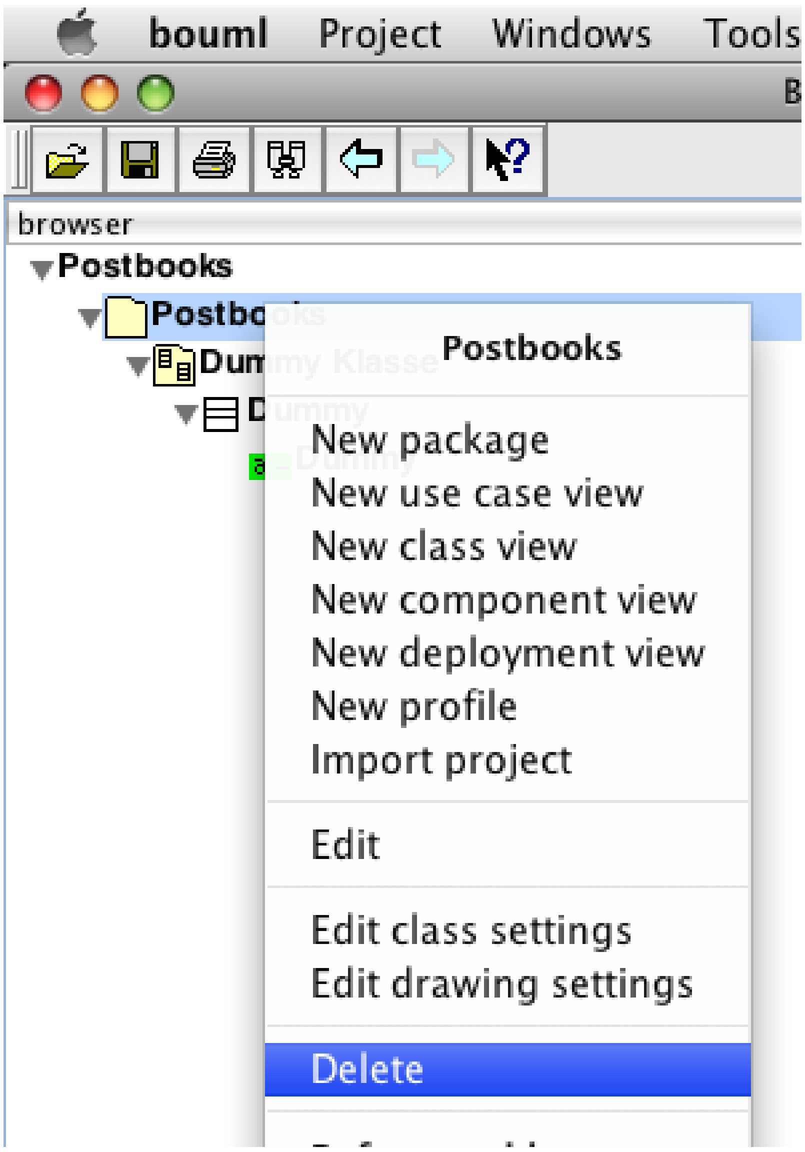
Other problems haven't arise while my tests with Postbooks. But in general arising problems can be narrowed by using manual transformation and check the resulting files using XSLTPROC.

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

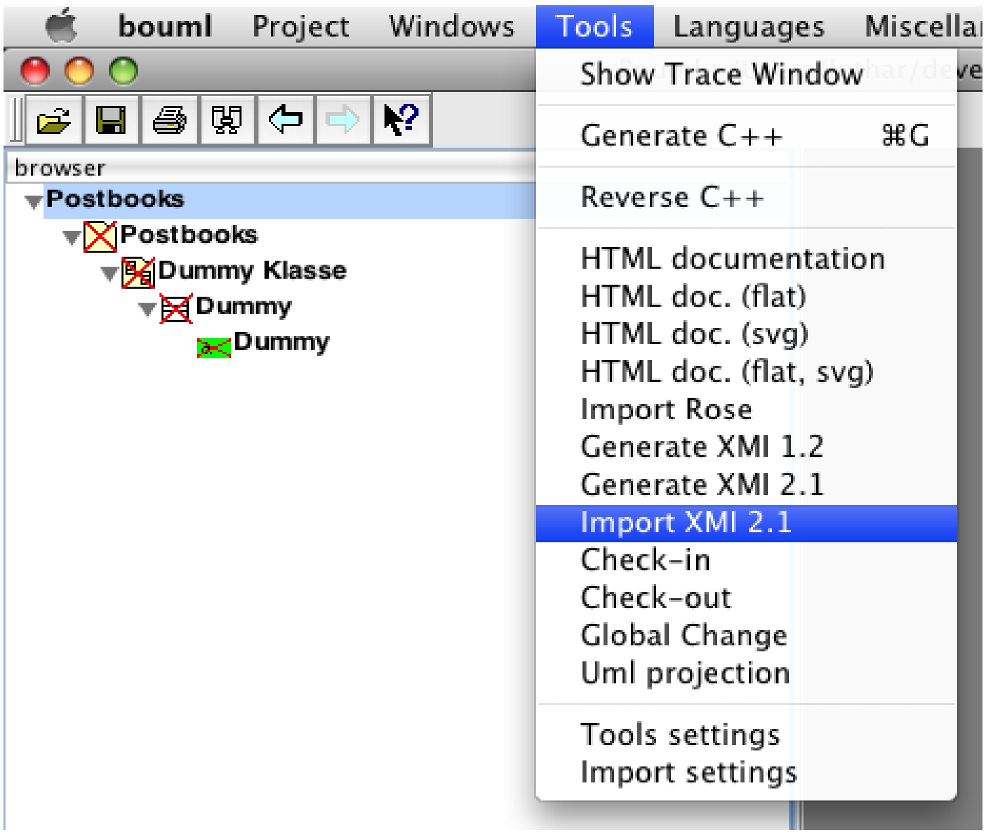
If you have exported the model, open BoUML to import it. You could use the dummy project created earlier, but delete the elements as follows:



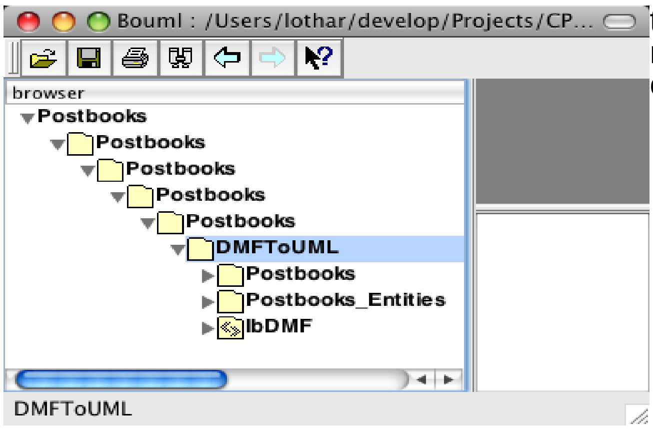
After that the elements are still visible, but marked as deleted. You could revert the deletion, start with the import or reopen the project to remove the deleted elements.

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 :

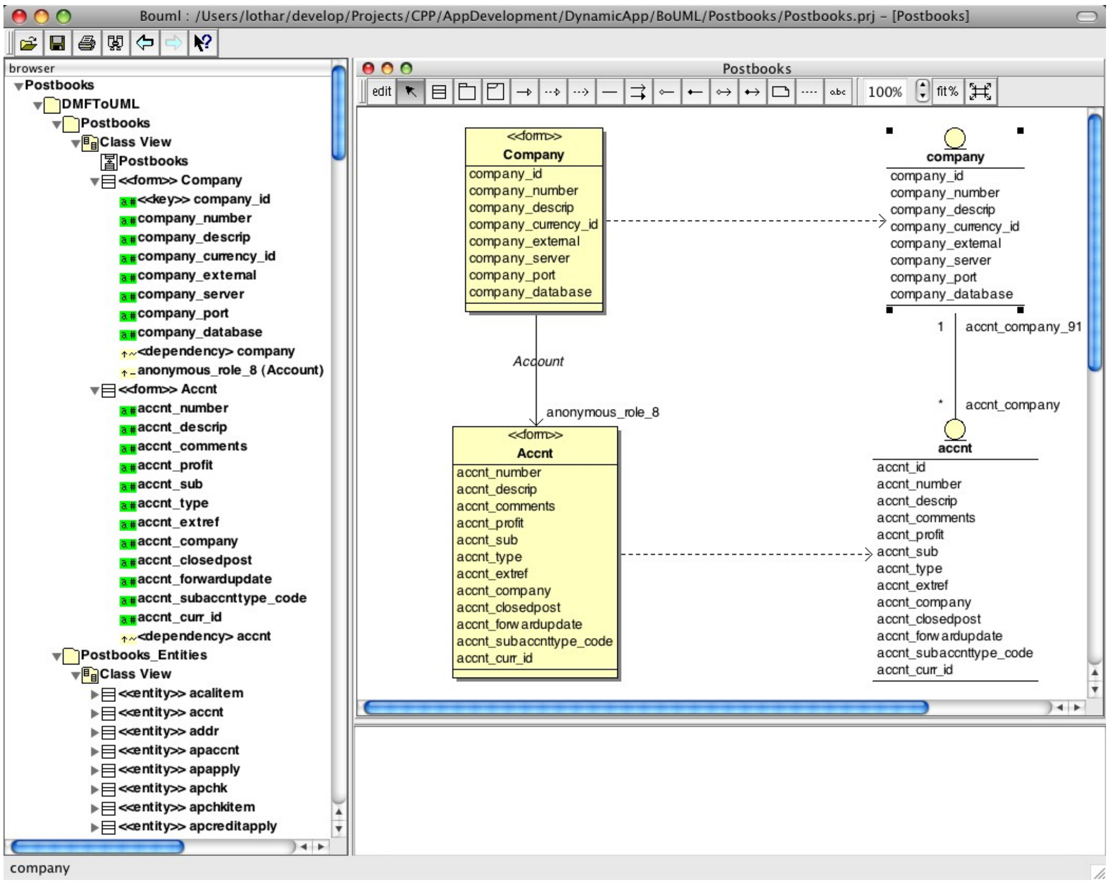


In that case you could move out the selected node into the outer most element and delete the other to correct that.

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).

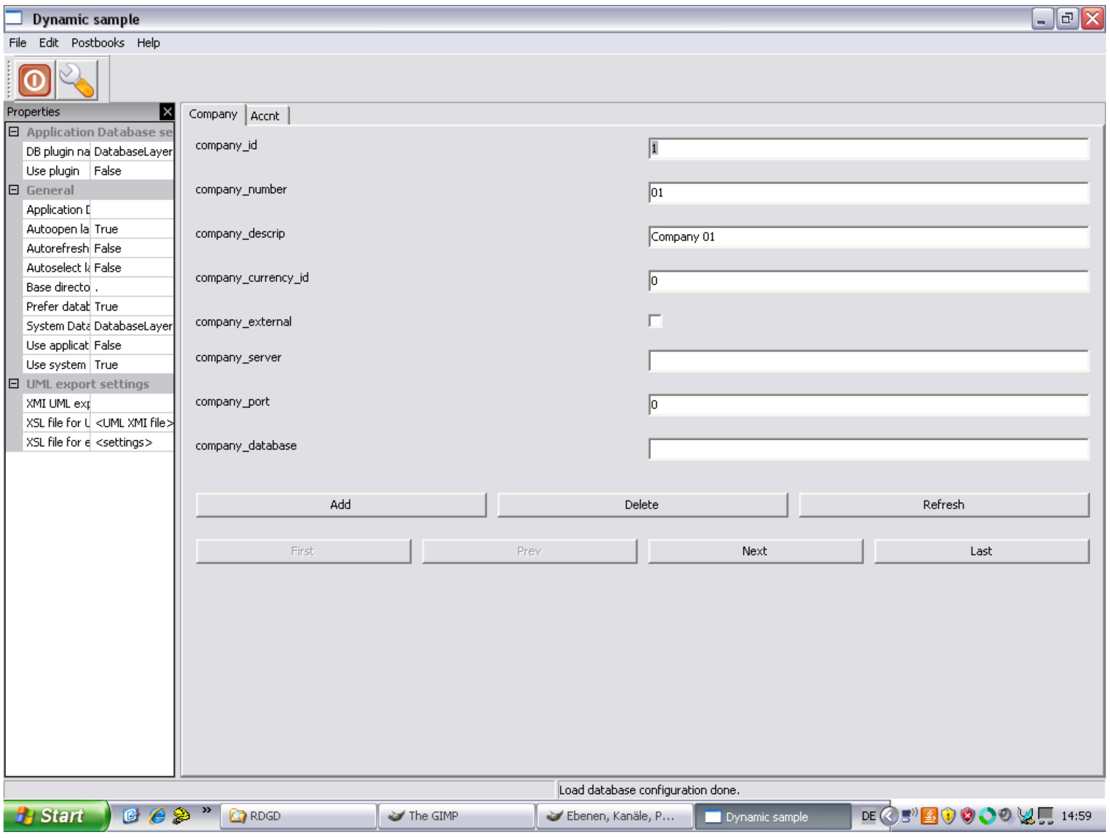
The tables are located in the package 'Postbooks_Entities' and the form classes are located in the package 'Postbooks'. In the diagrams you could show them together. Note that it will be better to create the diagrams in the entity package, as I have noticed there would be drawn all relations.

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:



Properties

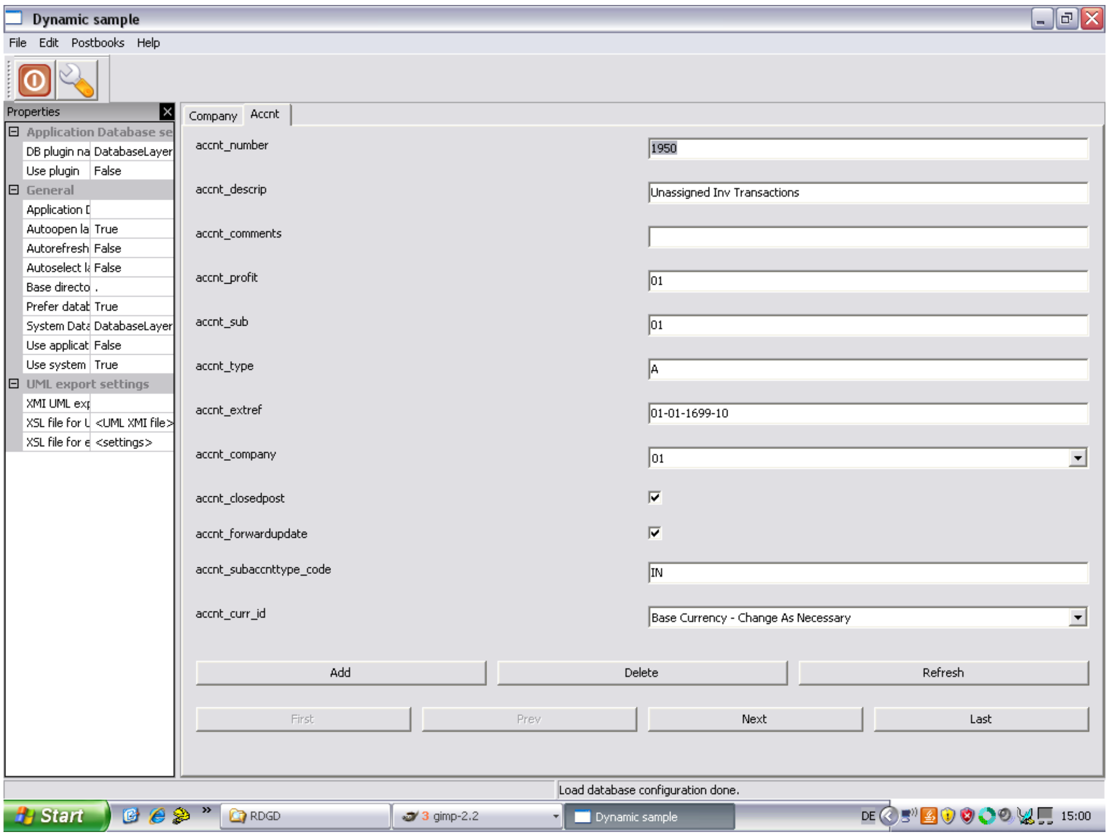
⊟ Application Database se	
DB plugin na	DatabaseLayer
Use plugin	False
⊟ General	
Application D	
Autoopen la	True
Autorefresh	False
Autoselect la	False
Base directo	.
Prefer datab	True
System Data	DatabaseLayer
Use applicat	False
Use system	True
⊟ UML export settings	
XMI UML exp	
XSL file for U	<UML XMI file>
XSL file for e	<settings>

company_id	1
company_number	01
company_descrip	Company 01
company_currency_id	0
company_external	☐
company_server	
company_port	0
company_database	

Load database configuration done.

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:



Application Database se	
DB plugin na	DatabaseLayer
Use plugin	False
General	
Application D	
Autoopen la	True
Autorefresh	False
Autoselect l	False
Base directo	.
Prefer datab	True
System Data	DatabaseLayer
Use applicat	False
Use system	True
UML export settings	
XMI UML exp	
XSL file for U	<UML XMI file>
XSL file for e	<settings>

accnt_number	1950
accnt_descrip	Unassigned Inv Transactions
accnt_comments	
accnt_profit	01
accnt_sub	01
accnt_type	A
accnt_extref	01-01-1699-10
accnt_company	01
accnt_closedpost	✓
accnt_forwardupdate	✓
accnt_subaccnttype_code	IN
accnt_curr_id	Base Currency - Change As Necessary

Load database configuration done.

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.

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:

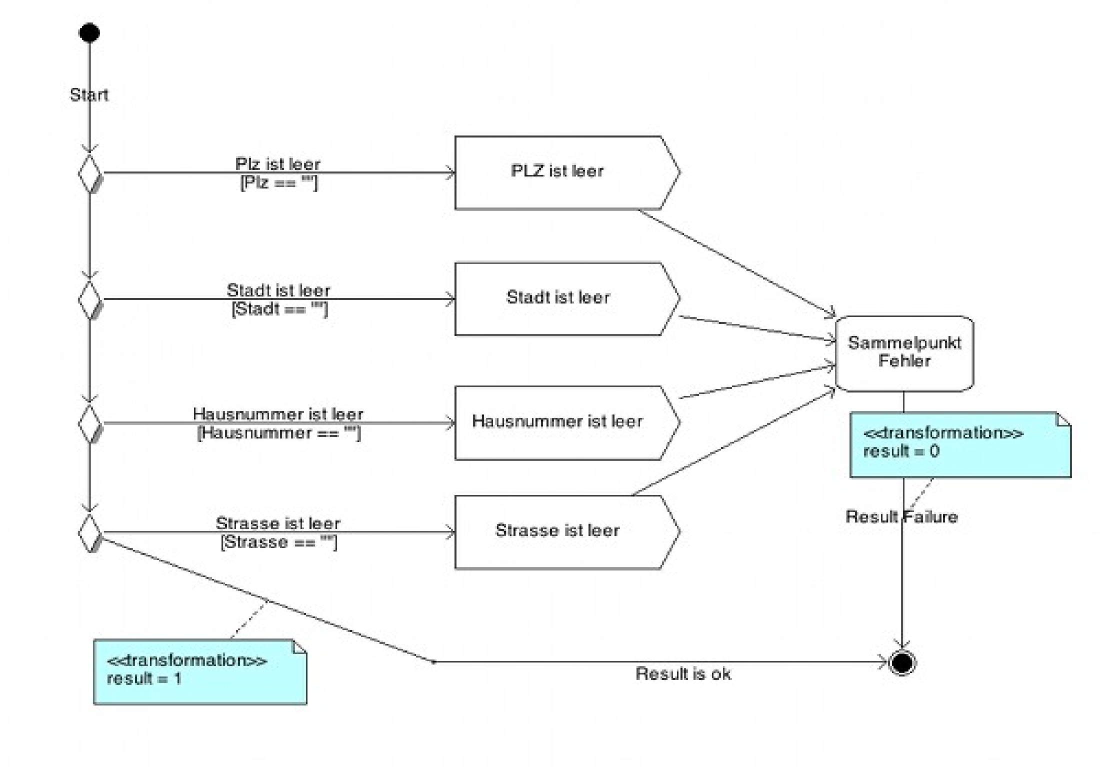
„Need entity country with additional formatting rules for zip codes“

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 is based on the fact that signal actions are possible 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: 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.

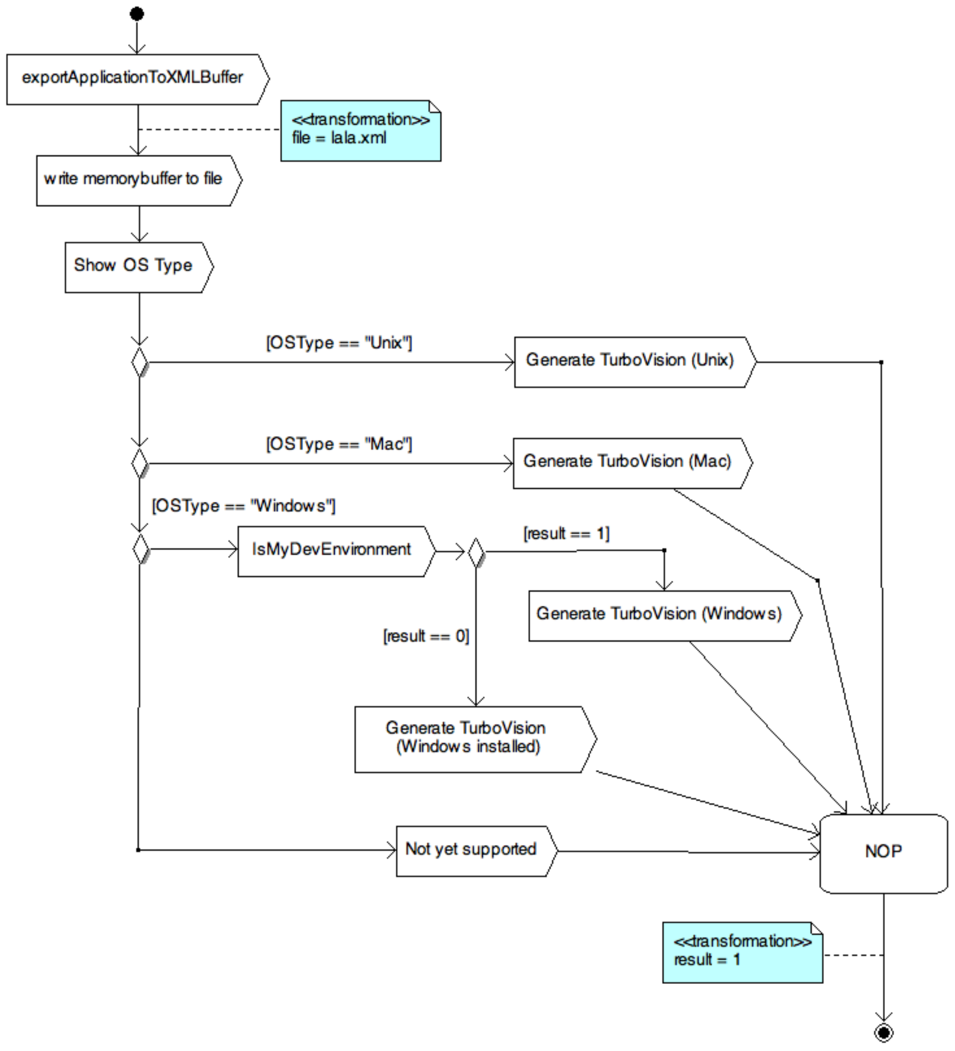
The code generation activity is **not** in the application model if you install or use the application the first time. You need to import the UML model.

As a sample the second stage UML model of lbDMF Manager contains the Activity to be imported into the application (do not overwrite the system database schema, only overwrite the application model (the second step).

When you have imported the model, restart the application to activate the changes (but set the 'Prefer database configuration' property to true). Then you will find a new button in the form 'Anwendungen' (use dialog, not table layout).

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. 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 marschalls the parameter. 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 a interface definition, but it is additionally wrapped by a unmarshaling 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)

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 unmarshaler 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 unmarshalling 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 unmarshaling 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 unmarshaling 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 unmarshal method and / or use it later directly. Don't forget to register the unmarshaling 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.

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.

Lets 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 mad 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 get become 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.

```
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 enhough 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.

```
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.