HyperSenses®

Tutorial: Getting Started

MA 21'598.02

DELTA

software technology

Imprint

Title	HyperSenses® Tutorial: Getting Started
Manual-No.	MA 21'598.02
Product	HyperSenses®
Date	December 2011
Disclaimer	Delta Software Technology GmbH reserves the right to make improvements in the product described in this manual at any time without notice.
Copyright	© 2011 Delta Software Technology GmbH This manual contains proprietary information. All rights are reserved. This document may not in whole or in part be copied, photocopied, reproduced, translated or reduced to any electronic medium or machine readable form without prior consent in writing from Delta Software Technology GmbH.
Trademark	Delta, SCORE, ObjectBridge, SCOUT², AMELIO, HyperSenses and the logo of Delta Software Technology are registered trademarks and SCORE Adaptive Bridges, SCORE Data Architecture Integration, Model Driven Legacy Integration, Integration in Motion, SCORE Transformation Factory, AMELIO Modernization Platform, ADSplus, ANGIE and Active Intent are trademarks of Delta Software Technology GmbH in Germany and/or other countries. All other registered trademarks, trademarks, trade names or service marks are the property of their respective owners.
Copies	Re-order copies of this manual from any local Delta distributor.
Homepage	http://www.hypersenses.com

Table of Contents

General

Document Conventions

In this document the following text conventions are used:

Convention	Description
`Syntax element`	Syntax elements, filenames, directories etc are written in this font
Menu points	Menu points are bold and cursive.
Controls	Windownames, texting of controls and attribute names are written in this font.

Software Required

- HyperSenses Version 3.0 or higher

Introduction

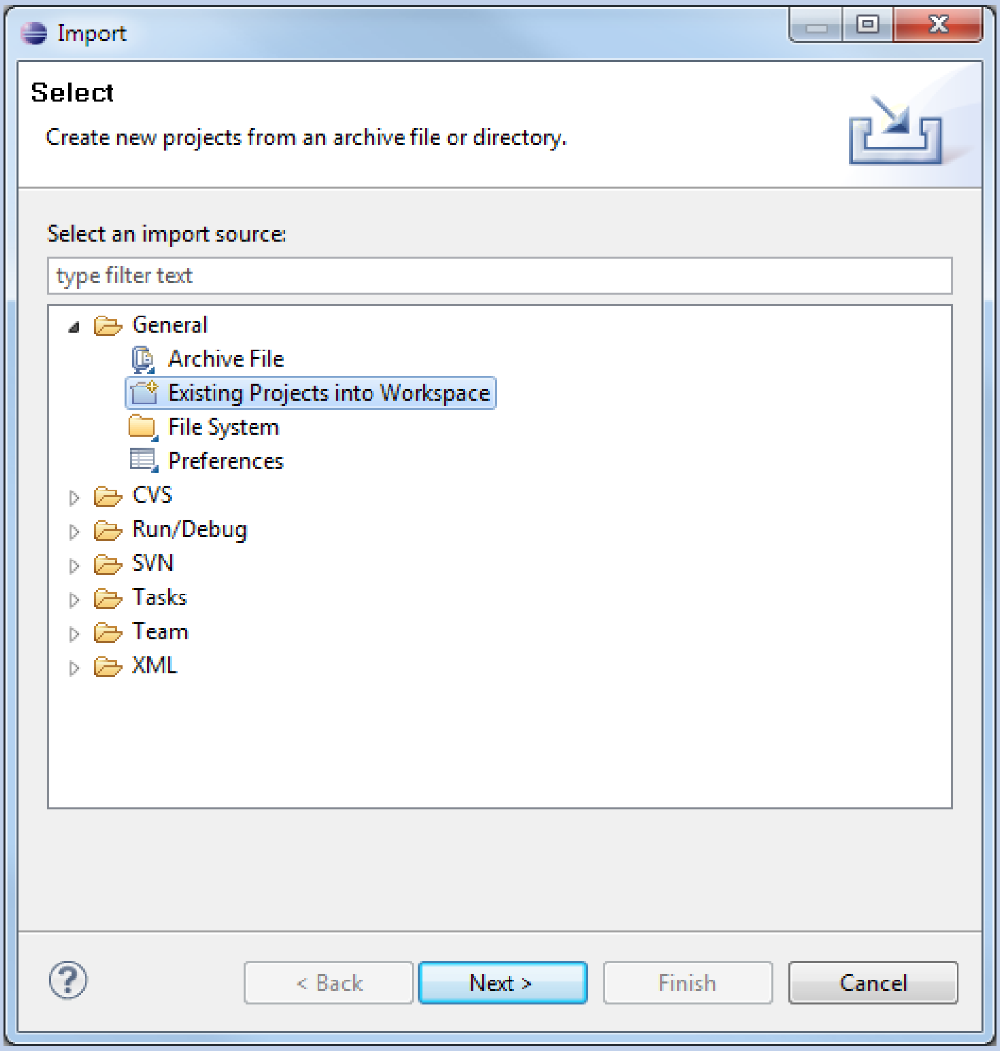
With this small HyperSenses exercise we want to generate PL/SQL scripts that are intended for the creation of tables (`CREATE TABLE`) and foreign key constraints (`ALTER TABLE`) for an Oracle database. We briefly explain every HyperSenses element (Meta model, pattern, configuration, DSL, ...).

The example is taken from DotNetPro magazine, issue 2/2011 "Programme zeichnen oder beschreiben" ("Designing or describing programs") by Mykola Dobrochynskyy.

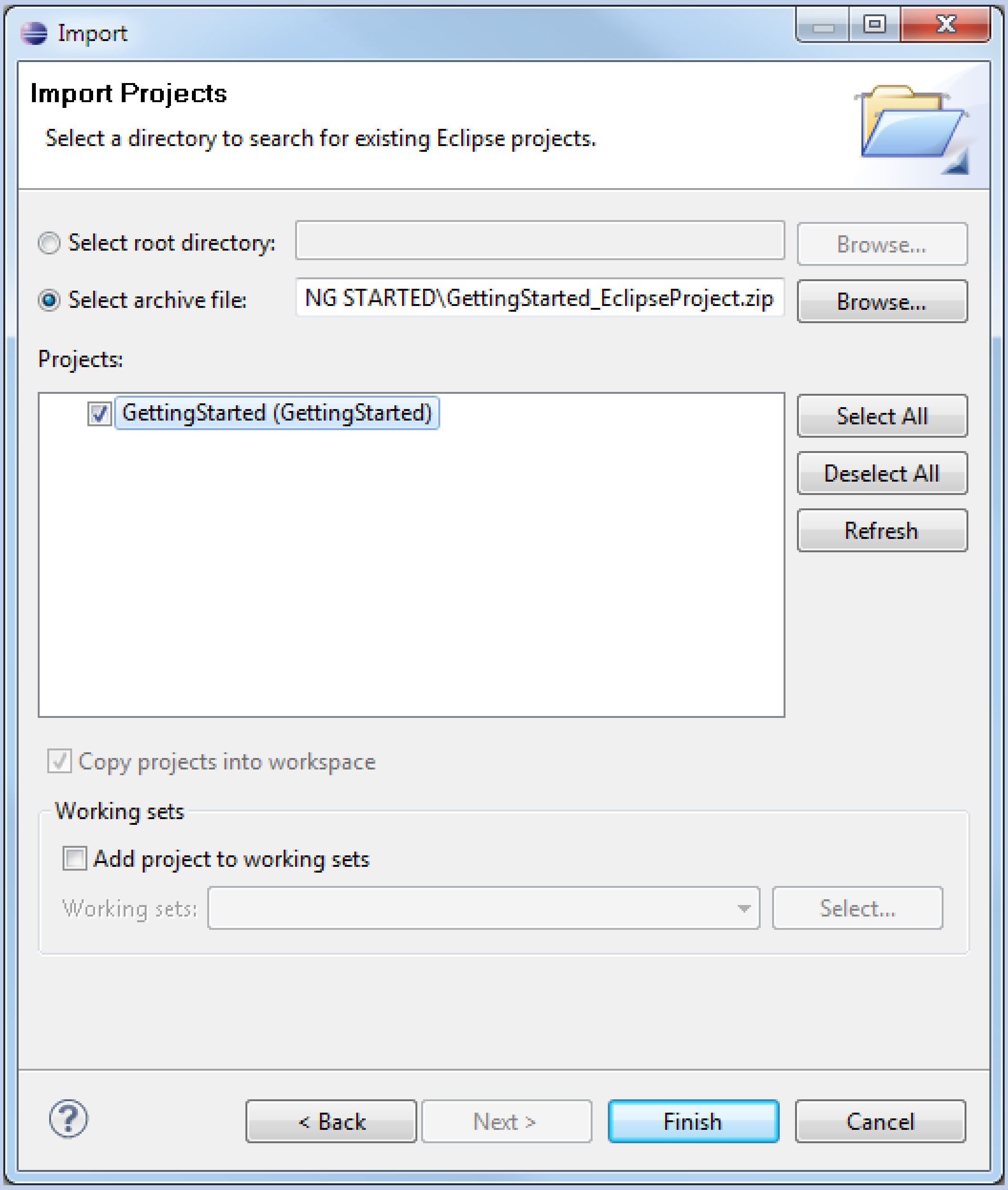
In this tutorial, you can choose between two ways (highlighted in colours as follows):

HyperSenses standalone	HyperSenses in the Eclipse environment

Run Eclipse and install the HyperSenses plugin if necessary. More information can be found in the manual "HyperSenses - Configurator Plugin for Eclipse" (MA 21'597). The 'Import' dialog opens by selecting ***File / Import...*** . Here, please open *General* and select *Existing Projects in Workspace*. We proceed by selecting *Next*.



In the next dialog, please select *Select Archive file:* and navigate to the HyperSenses data directory via *Browse*, for example: `C:\Users\<CurrentUser>\Delta\HE\TUTORIAL\GETTING STARTED\` and open the project archive 'GettingStarted_EclipseProject.zip'.



Select archive file: NG STARTED\GettingStarted_EclipseProject.zip

Projects: [✓] GettingStarted (GettingStarted)

[✓] Copy projects into workspace

[ ] Add project to working sets

We proceed by selecting *Finish*. Now open the Meta Composition 'Tutorial-1_1.META' by a double-click.

Run *HyperSenses Meta Composer* and open the Tutorial 1 *Meta Composition*. For this, select ***File / Open*** and switch to the folder '`TUTORIAL\GETTING STARTED\METACOMPOSITION`' and select 'Tutorial-1_1.META' then.

This *MetaComposition* already contains a prepared meta model and two production patterns. The pattern *CreateTable* contains the SQL code we want to generalize.

Note: You can restore the delivery state of a tutorial at any time by starting the respective self-extracting file in the local user directory.
For example: `C:\Users\<CurrentUser>\Delta\HE\TUTORIAL\GettingStarted.exe`

You can delete and re-import your project in the Eclipse environment.

To gain a short overview, we first have a look at the meta model. Select the 'MetaModel' tab in the *MetaComposition*. When you open 'Classes', you see that the meta model contains three meta classes. By double-clicking on a class and subsequent opening you see its features:

1. *TableScript* with the feature `tables` as composite of class *Table*
   (this class is only be required as top link for the generation)
2. *Table* with the features `name` as string and `columns` as composite of class *Column*
3. *Column* with the features `name` as string and `datatype` as string

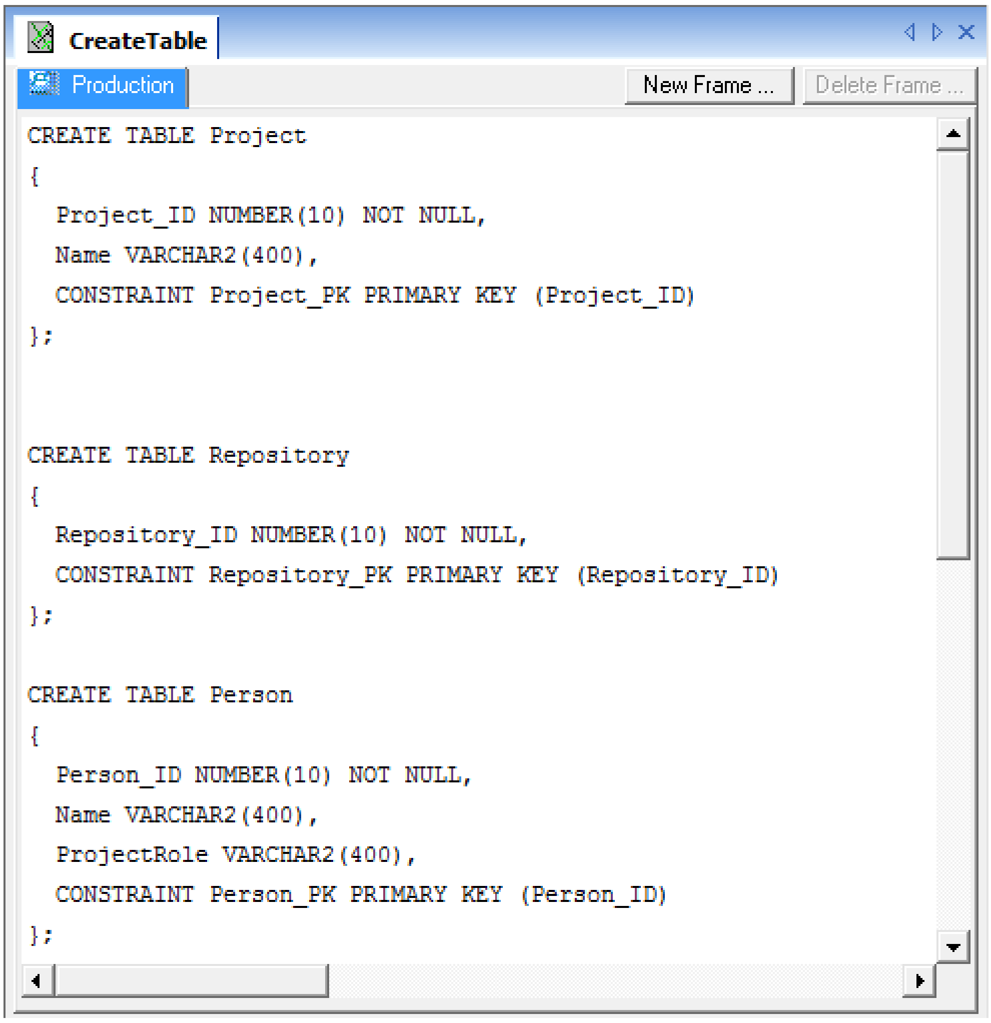
The meta model solely consists of the variable parts of a system family and is independent of the language to be generated. We describe the invariant parts in code patterns (also referred to as production patterns) which are language-dependent. We have previously modeled the meta model in HyperSenses and additionally visualised it as UML model. Close the Meta model.

If a meta class is later bound to a pattern, it is its *Context Class*.

Exercise 1

Opening the First Pattern

Switch to the 'Pattern' tab which contains two patterns: *CreateTable* and *TableScript*. First, open the *CreateTable* pattern with a double-click. Here you see the already prepared SQL code that we want to generalize by identifying the variable parts.



```
CREATE TABLE Project
{
  Project_ID NUMBER(10) NOT NULL,
  Name VARCHAR2(400),
  CONSTRAINT Project_PK PRIMARY KEY (Project_ID)
};


CREATE TABLE Repository
{
  Repository_ID NUMBER(10) NOT NULL,
  CONSTRAINT Repository_PK PRIMARY KEY (Repository_ID)
};

CREATE TABLE Person
{
  Person_ID NUMBER(10) NOT NULL,
  Name VARCHAR2(400),
  ProjectRole VARCHAR2(400),
  CONSTRAINT Person_PK PRIMARY KEY (Person_ID)
};
```

We have already bound the pattern directly to the context class *Table*. You can check this within the *Pattern Properties* where you find 'Context Class' as entry of the 'Interface' tab. You could change the context class there, but this is currently not necessary.

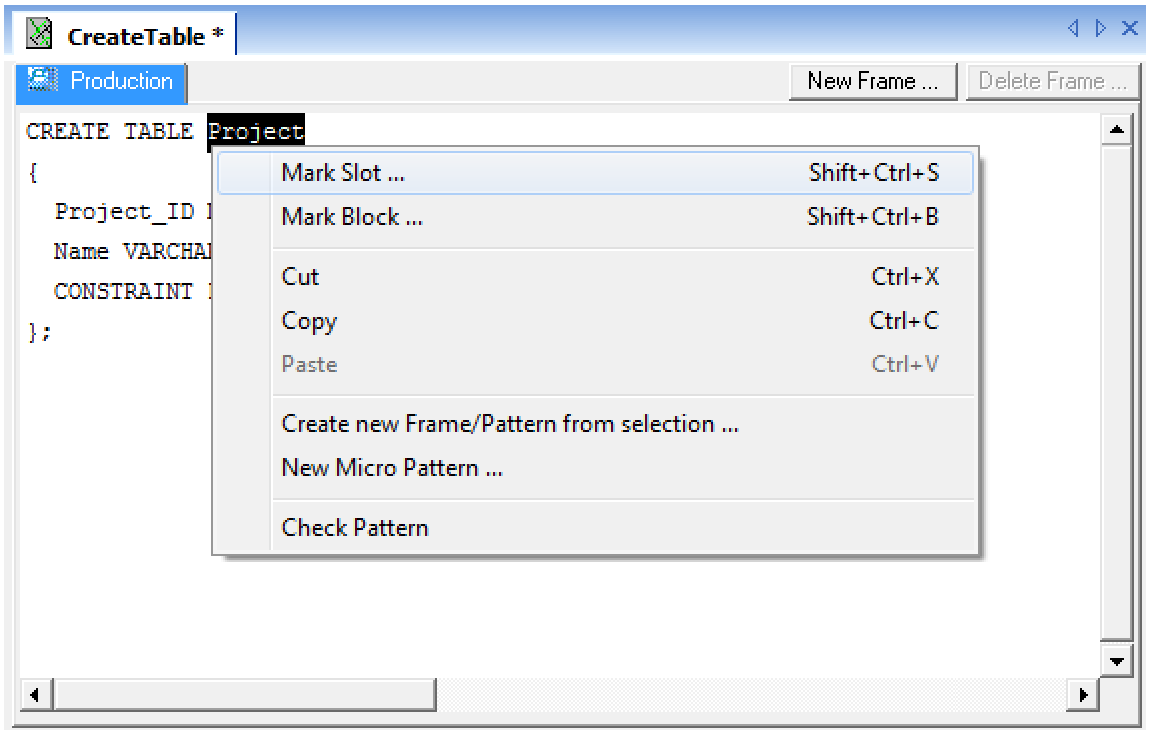
Have a closer look at the `CREATE TABLE` statements. They repeat themselves again and again. The differences lie in the changing of table names as well as the names and data types of the columns:

- `VARCHAR2(400)` as *String*
- `NUMBER(10)` as *Integer*
- `DATE` as *Datetime*

You can therefore delete all occurrences except one (e.g. `CREATE TABLE Project...`).

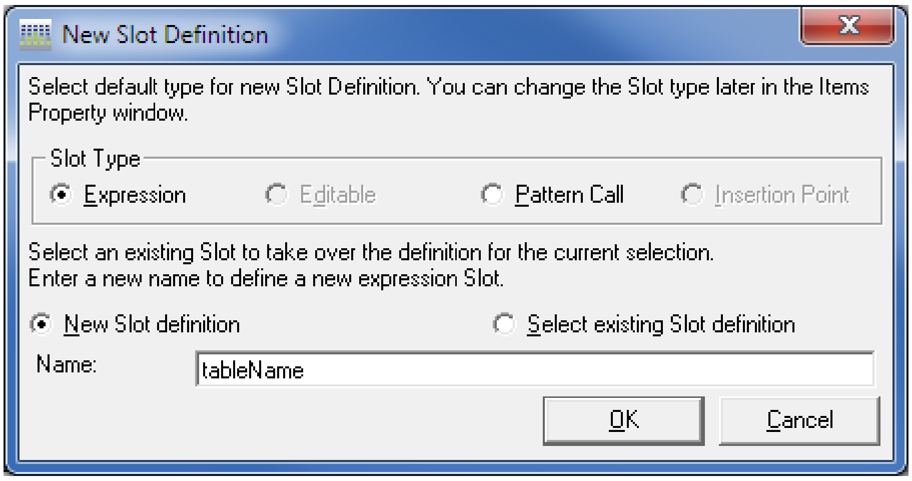
Creating a Slot

The first variable part is the table name. The name directly succeeds `CREATE TABLE...`, `Project` in this case. Mark the table name `Project`. Create a slot by opening the context menu (the context menu can be opened by right-clicking on the marked text) and selecting 'Mark Slot…'.



```
CREATE TABLE Project
{
  Project_ID I
  Name VARCHAI
  CONSTRAINT I
};
```

The 'New Slot Definition' dialog opens. There you can choose either *Expression* or *Pattern Call* as slot type. *Expressions* are used to calculate an expression or to read out a value from the configuration. A *Pattern Call* is used if another pattern should be called. Select *Expression*. Due to the fact that this is the first slot, the selection 'New Slot definition' is fine. You may choose a generic name, 'tableName' for example.



New Slot Definition

Select default type for new Slot Definition. You can change the Slot type later in the Items Property window.

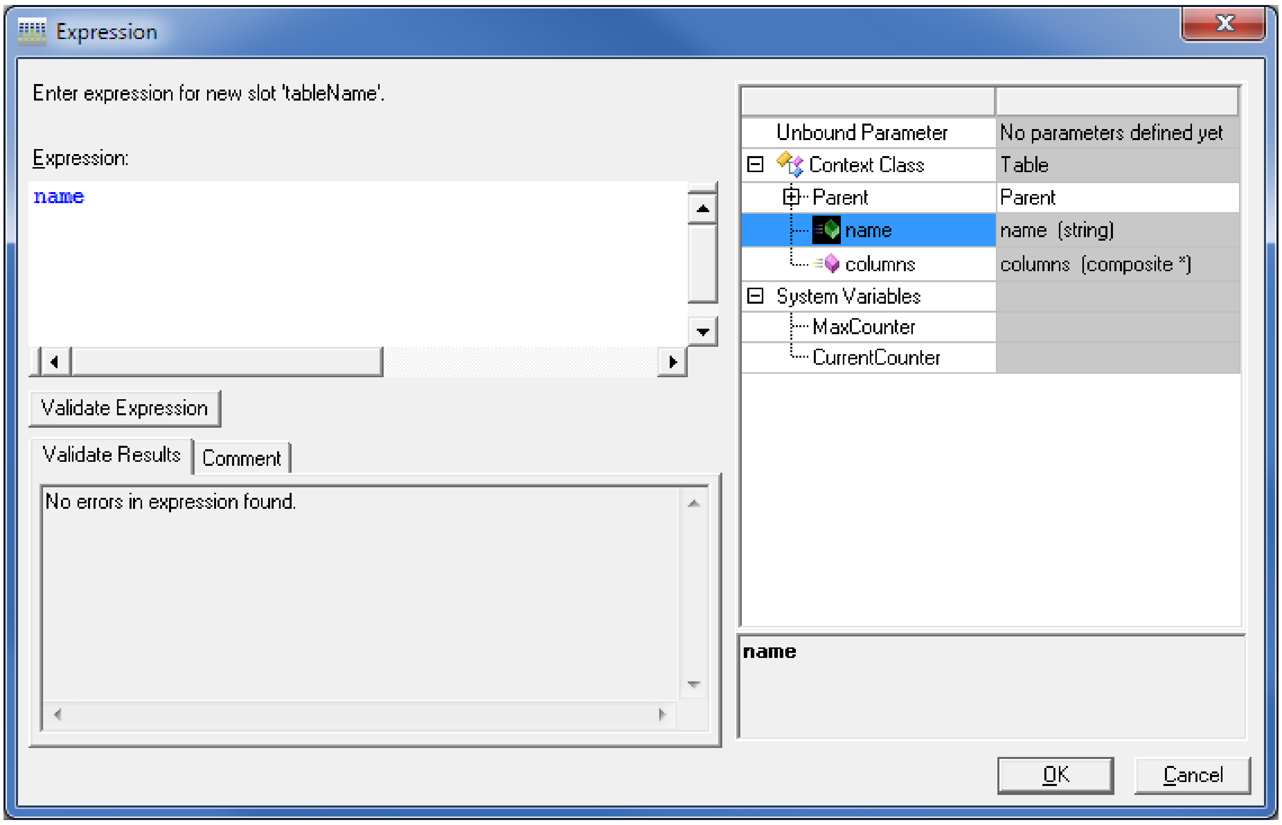
Slot Type: [✓ Expression] Editable Pattern Call Insertion Point

Select an existing Slot to take over the definition for the current selection.
Enter a new name to define a new expression Slot.

[✓ New Slot definition] Select existing Slot definition

Name: tableName

By selecting 'OK' the *Expression* dialog opens. There you have to assign a value to the slot. On the right side of the dialog you can see the context class of the pattern which is *Table*. According to the meta model, the *Table* class has the feature `name`. Select this feature and drag it into the *Expression* field. Then leave the dialog by selecting 'OK'.


Expression

Enter expression for new slot 'tableName'.

Expression:

```
name
```

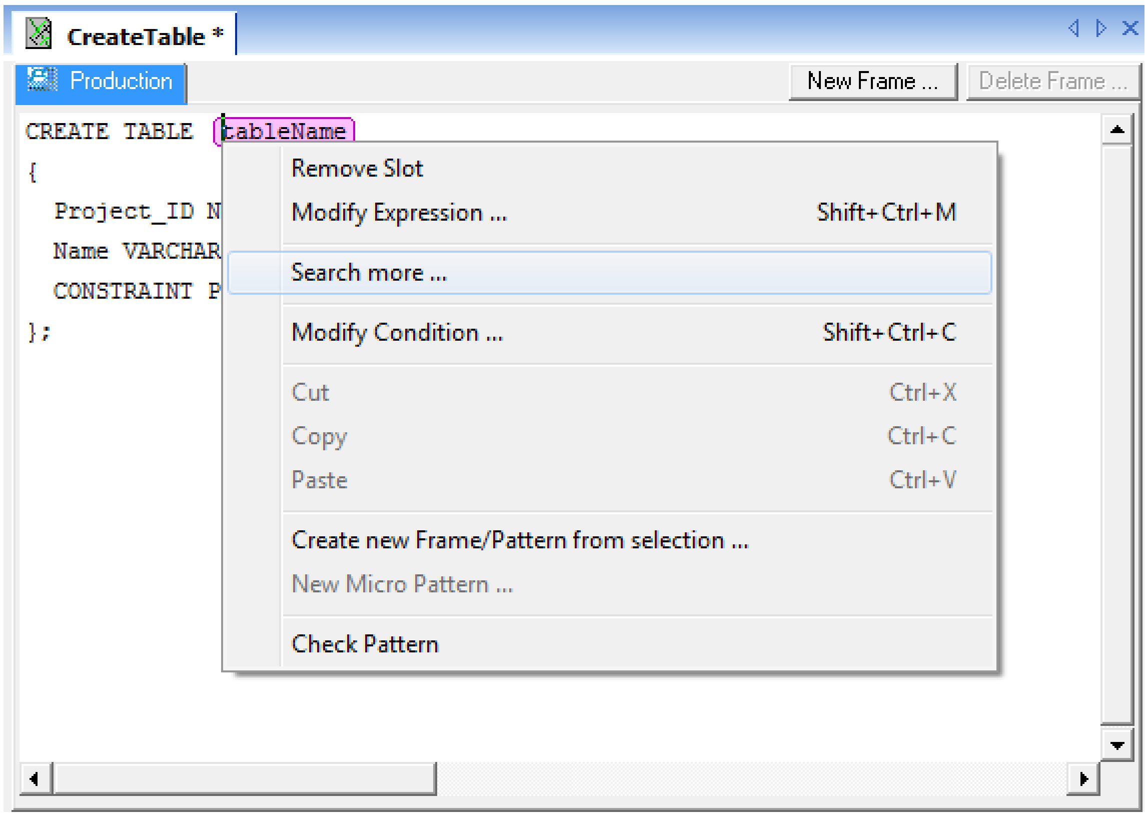
No errors in expression found.

Unbound Parameter	No parameters defined yet
Context Class	Table
Parent	Parent
name	name (string)
columns	columns (composite *)
System Variables	
MaxCounter	
CurrentCounter	

name

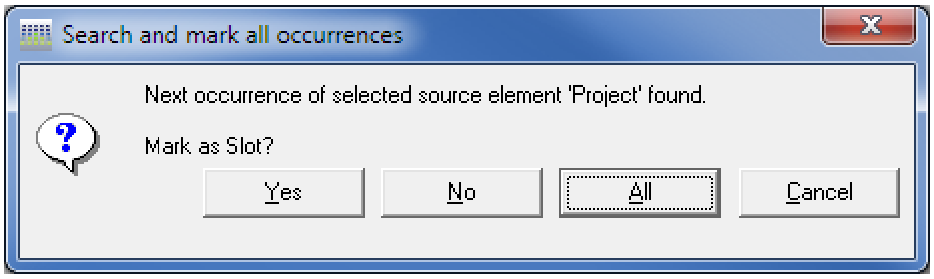
Finding the Same Slots

The table name must be replaced by a slot at further positions. Right-click on the just created slot. Then, select 'Search more...' in the context menu.



```
CREATE TABLE tableName
{
  Project_ID N
  Name VARCHAR
  CONSTRAINT P
};
```

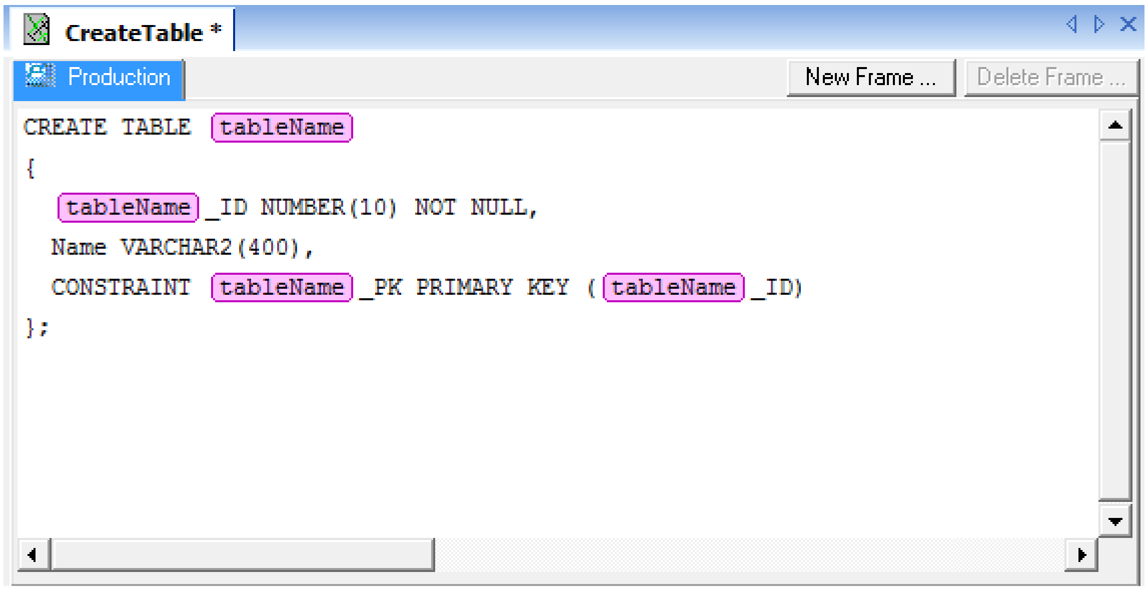
A dialog appears that enables you to replace one or all found occurences, select 'All'.



Next occurrence of selected source element 'Project' found.

Mark as Slot?

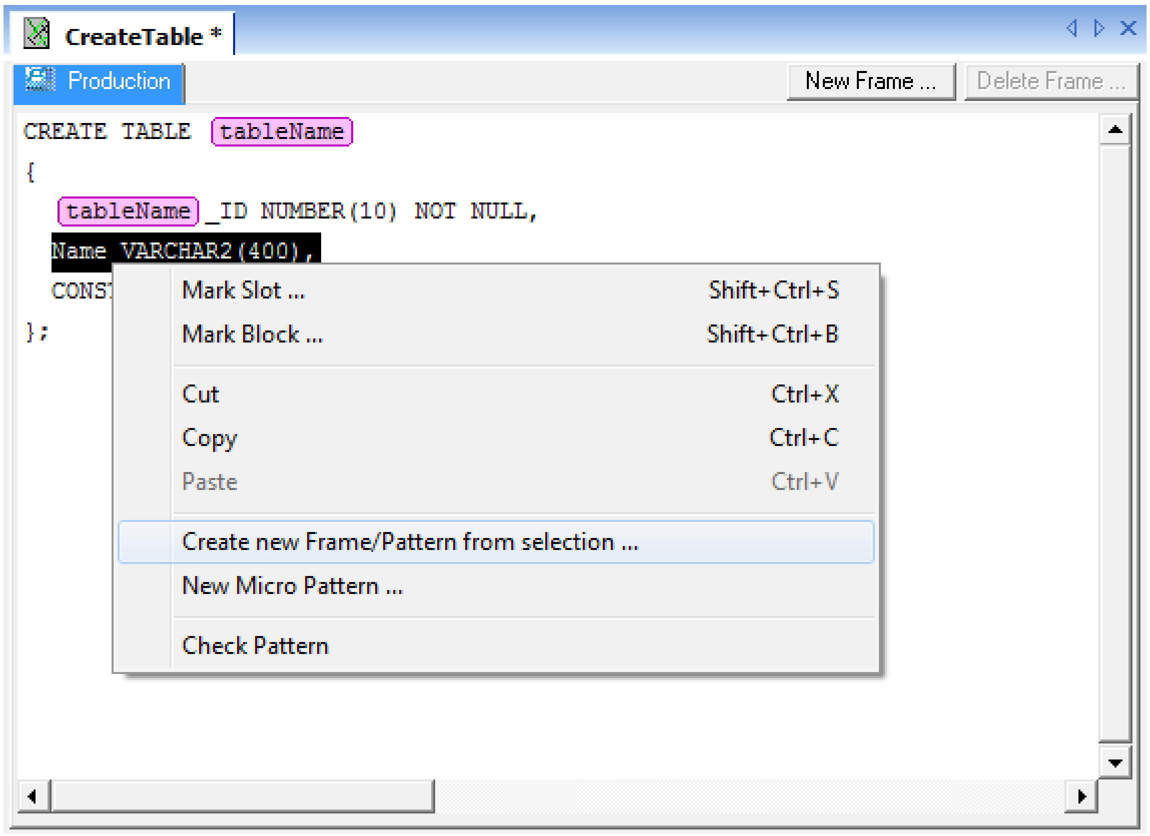
Now your pattern should look like this:



```
CREATE TABLE tableName
{
  tableName _ID NUMBER(10) NOT NULL,
  Name VARCHAR2(400),
  CONSTRAINT tableName _PK PRIMARY KEY (tableName _ID)
};
```

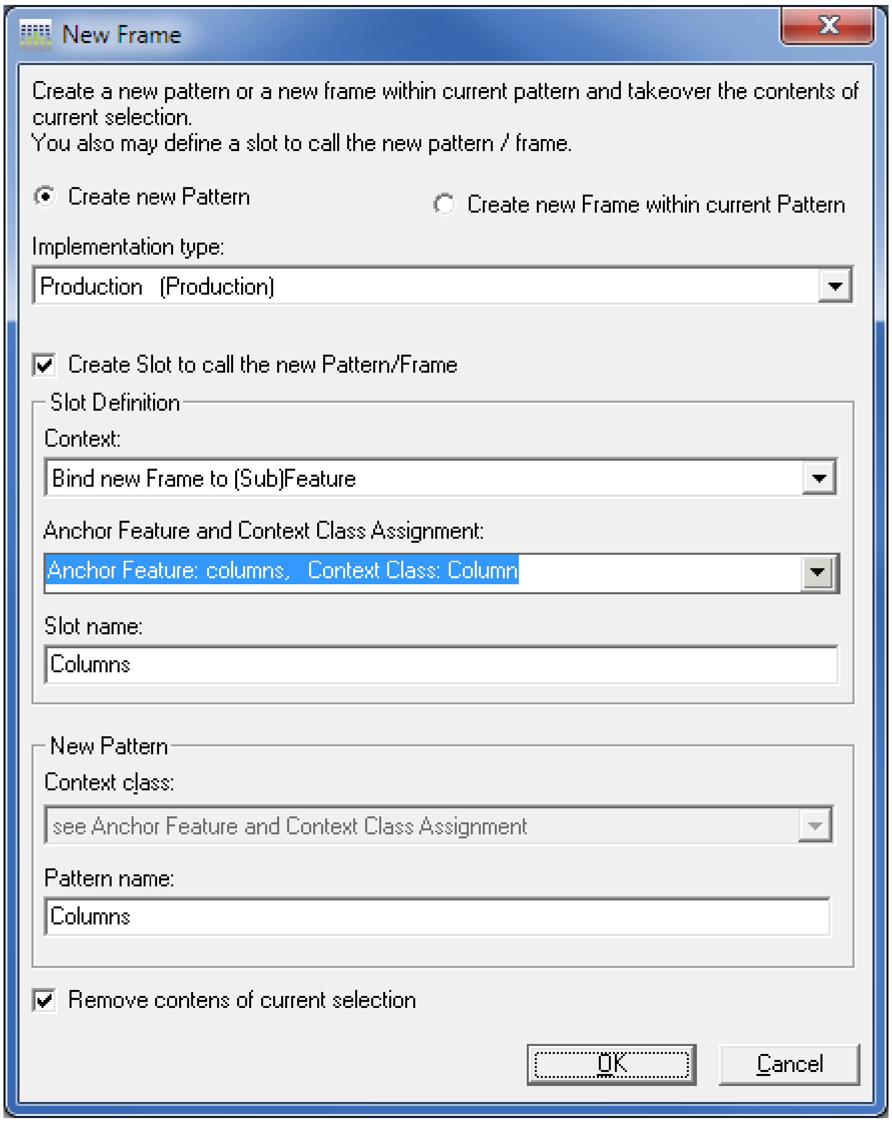
Deriving a New Pattern

Next, the table columns attract our attention as further variable position (`Name VARCHAR2(400)`). In the meta model, the variable parts of the columns are described by the features of the *Column* class. Thus we have to create a new pattern and bind it to the *Column* class. To do so mark `Name VARCHAR2(400),` and open the context menu and select 'Create new Frame/Pattern from selection…'.



```
CREATE TABLE tableName
(
  tableName_ID NUMBER(10) NOT NULL,
  Name VARCHAR2(400),
  CONS
);
```

The 'New Frame' dialog opens. 'Create new pattern' is already selected. The *Implementation Type* should be 'Production (Production)'. To ensure that the new pattern is called in the origin pattern, a slot must be created. For this reason 'Create Slot to call the new Pattern/Frame' is selected. Within the meta model the *Column* class is a sub-class of *Table*. As a new line has to be generated for each column and the calling pattern *CreateTable* is bound to meta class *Table*, we select as *Context* 'Bind new Frame to (Sub)Feature' and set 'Anchor Feature: columns, Context Class: Column'. 'Columns' would be a good choice for the *slot name* and *pattern name*. Finally, the checkbox at the bottom of the dialog is selected that deletes the marked code in the origin pattern *CreateTable*.



Create a new pattern or a new frame within current pattern and takeover the contents of current selection.
You also may define a slot to call the new pattern / frame.

● Create new Pattern ○ Create new Frame within current Pattern

Implementation type: Production (Production)

☑ Create Slot to call the new Pattern/Frame

Slot Definition

Context: Bind new Frame to (Sub)Feature

Anchor Feature and Context Class Assignment: Anchor Feature: columns, Context Class: Column

Slot name: Columns

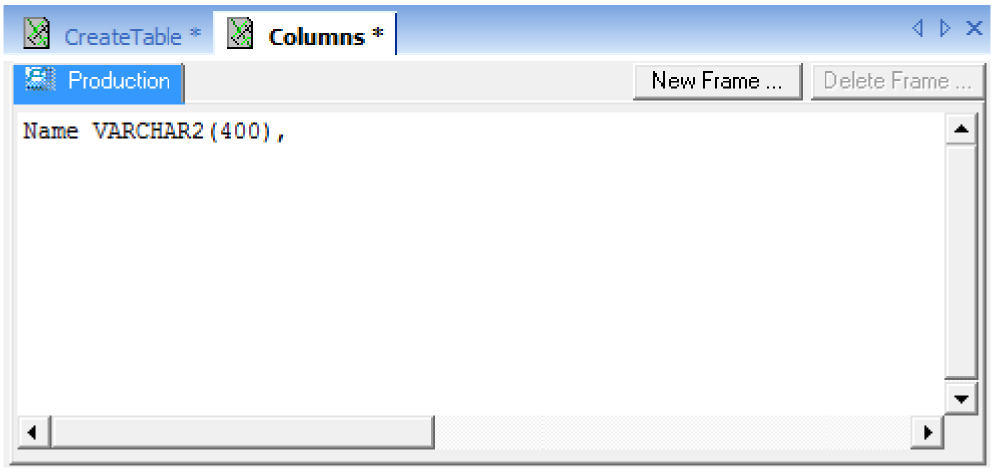
New Pattern

Context class: see Anchor Feature and Context Class Assignment

Pattern name: Columns

☑ Remove contens of current selection

By selecting 'OK', the new pattern is created and the SQL code inserted.

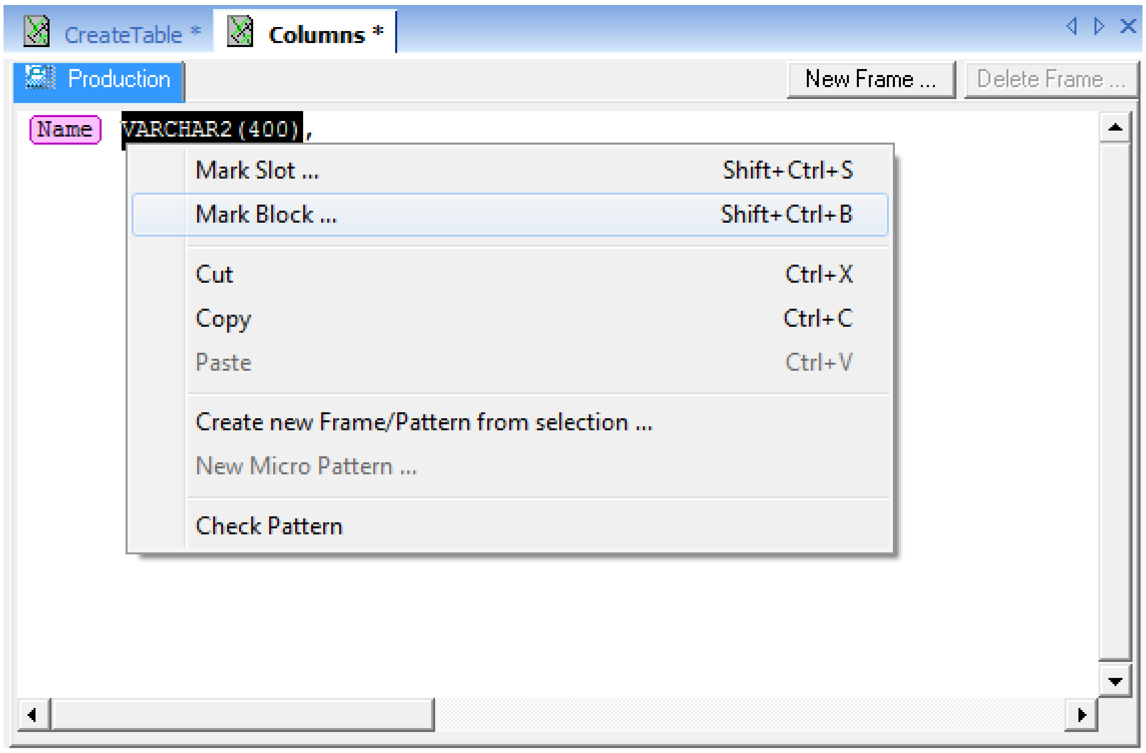


```
Name VARCHAR2(400),
```

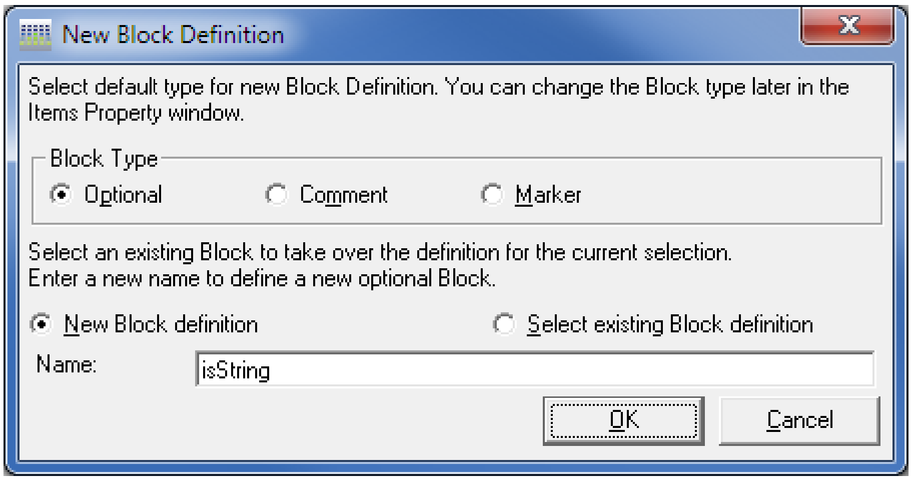
Creating an Optional Code Block with Condition

In the new pattern *Columns* we have again to find all variable parts and replace them by slots and optional blocks. For 'Name' create a new slot that contains the column name.
That means: Marking, Mark Slot…, Name (string) etc.

As not every column is of data type *String* (see chapter 'Introduction'), we pack the data types in optional code blocks with conditions. To do so mark `VARCHAR2(400)` and select 'Mark Block…' in the context menu.



In the following dialog select the *Block Type* 'Optional' and 'New Block definition'. We take 'isString' as block name.


New Block Definition

Select default type for new Block Definition. You can change the Block type later in the Items Property window.

Block Type

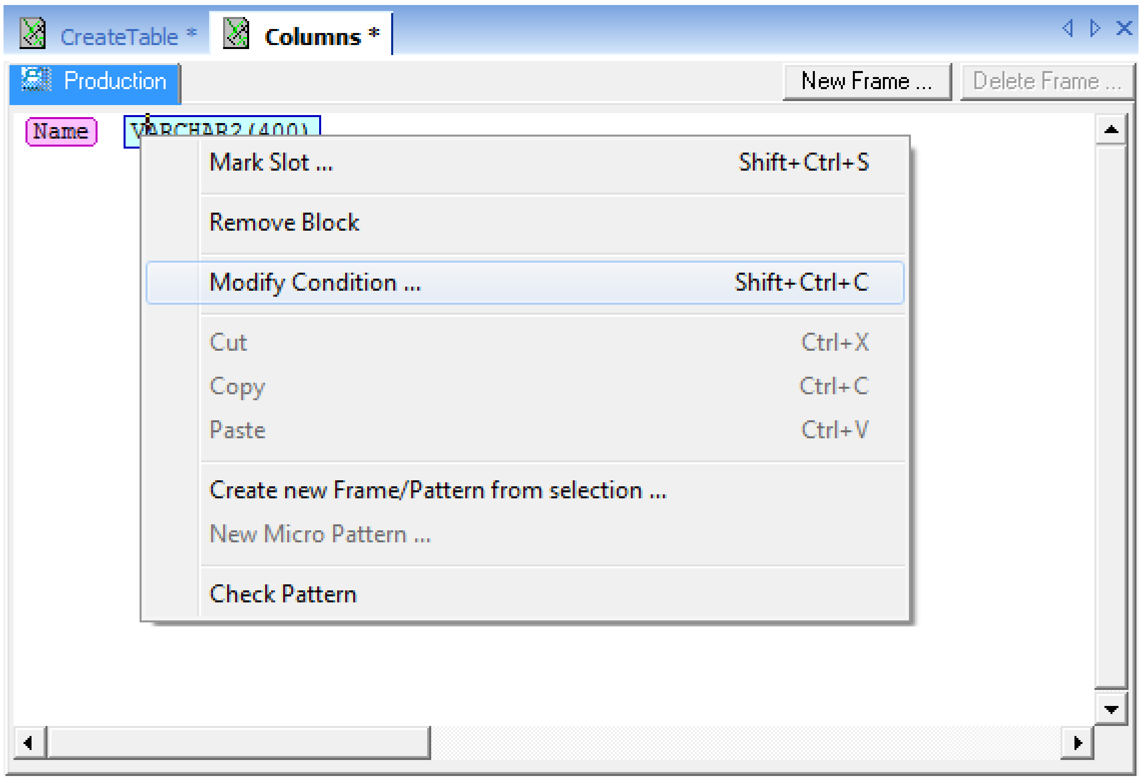
(● Optional) ○ Comment ○ Marker

Select an existing Block to take over the definition for the current selection.
Enter a new name to define a new optional Block.

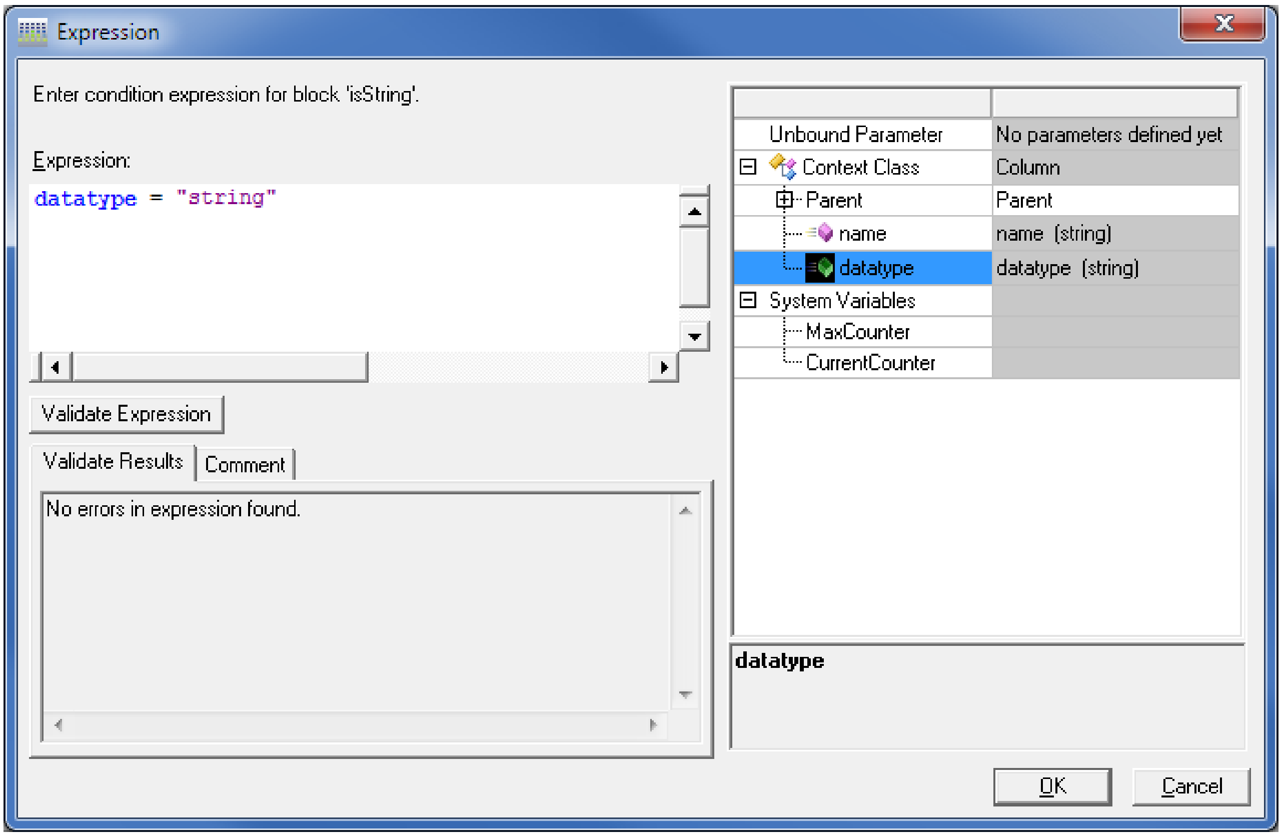
(● New Block definition) ○ Select existing Block definition

Name: isString

After selecting ‘OK‘, a block appears in the pattern but we still have to define a condition when the content of the block has to be generated. For this, place the cursor on the block and select 'Modify condition…' in the context menu.



The 'Expression' dialog opens. `VARCHAR2(400)` should always be generated when the column type is *string* `:datatype = "string"`



Enter condition expression for block 'isString'.

Expression:

```
datatype = "string"
```

No errors in expression found.

Unbound Parameter	No parameters defined yet
Context Class	Column
Parent	Parent
name	name (string)
datatype	datatype (string)
System Variables	
MaxCounter	
CurrentCounter	

datatype

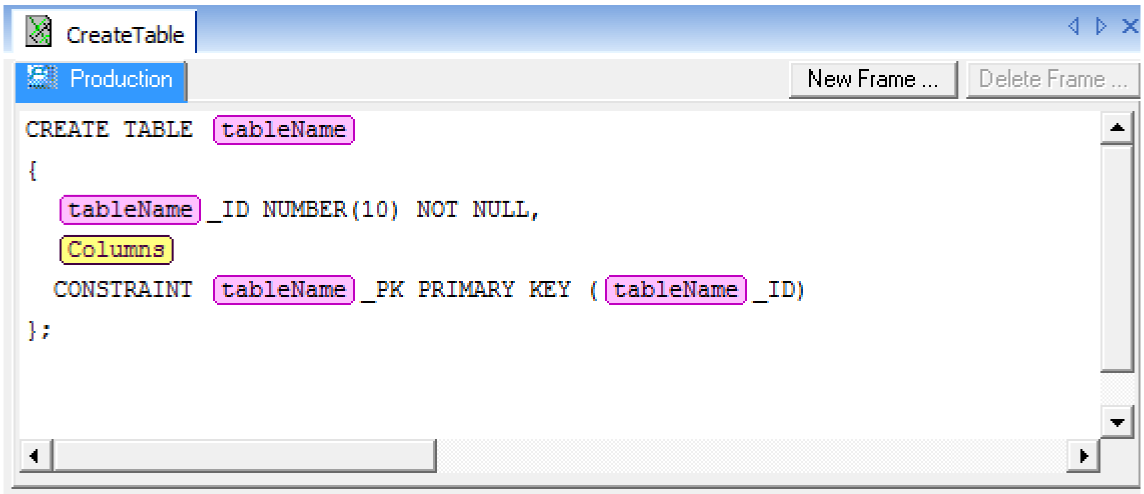
Leave the dialog by pressing 'OK'.

Creating Additional Optional Code Blocks

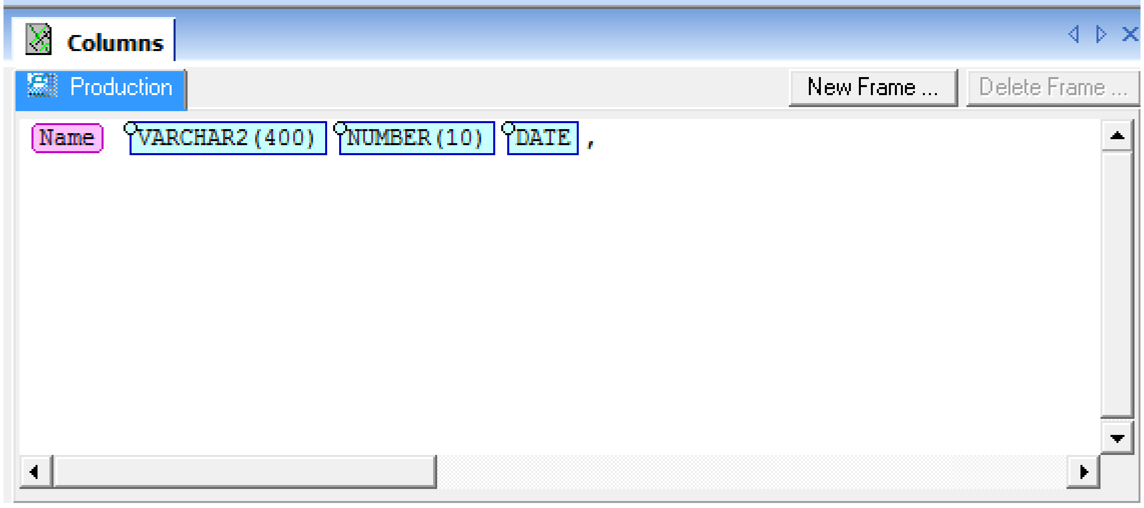
At the start we have seen that there were additional data types in the *CreateTable*, that's why we repeat the steps for creating an optional code block for the data types `int` and `datetime`. To do so type the following SQL code (in the Columns pattern) after the new block and create the according optional blocks with the associated conditions (see following screenshot):

- `NUMBER(10)`
  Block name → *isNumeric* with condition: `datatype = "int"`
- `DATE`
  Block name → *isDate* with condition: `datatype = "datetime"`

Now your patterns should look like this:

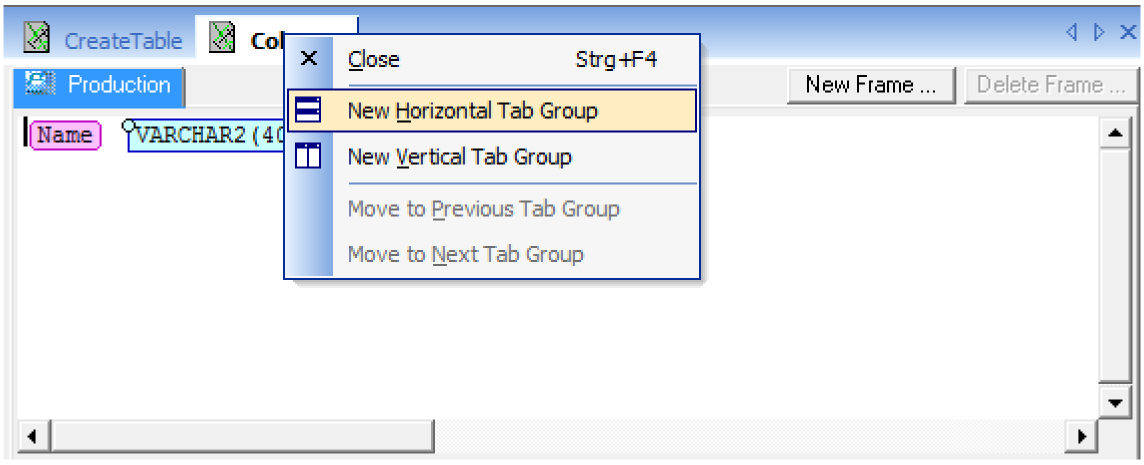


```
CREATE TABLE tableName
(
   tableName_ID NUMBER(10) NOT NULL,
   Columns
  CONSTRAINT tableName_PK PRIMARY KEY (tableName_ID)
);
```



```
Name  VARCHAR2(400) NUMBER(10) DATE ,
```

You can display the pattern window one above the other by the context menu of the pattern tabs in the main window.

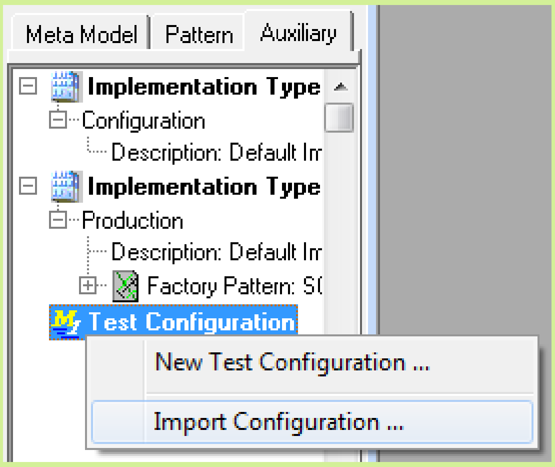


At this point you should save your changes.

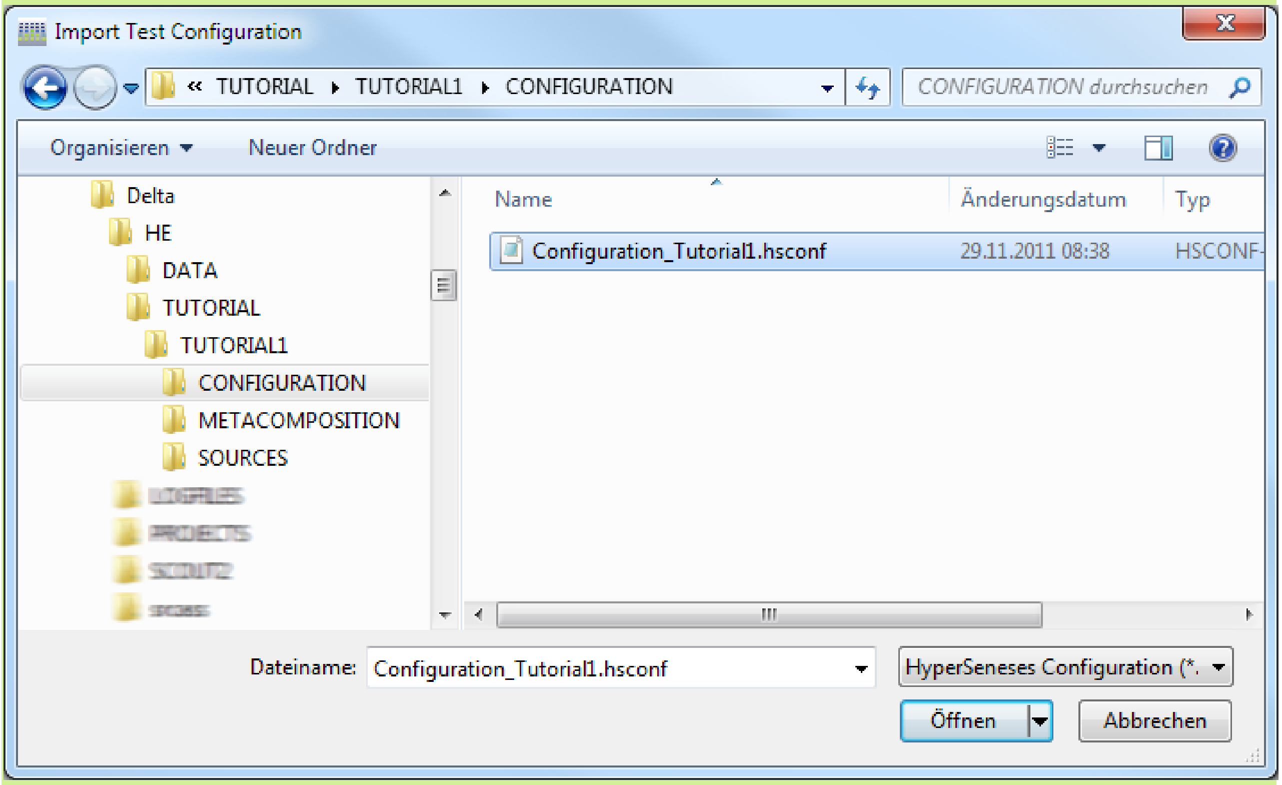
Our First Generation

Now we are ready to generate the *CreateTable* statements. For this, we use an already prepared, XML-formatted configuration.

Switch to the 'Auxiliary' tab, open the 'Test Configuration' context menu. Select 'Import Configuration'.



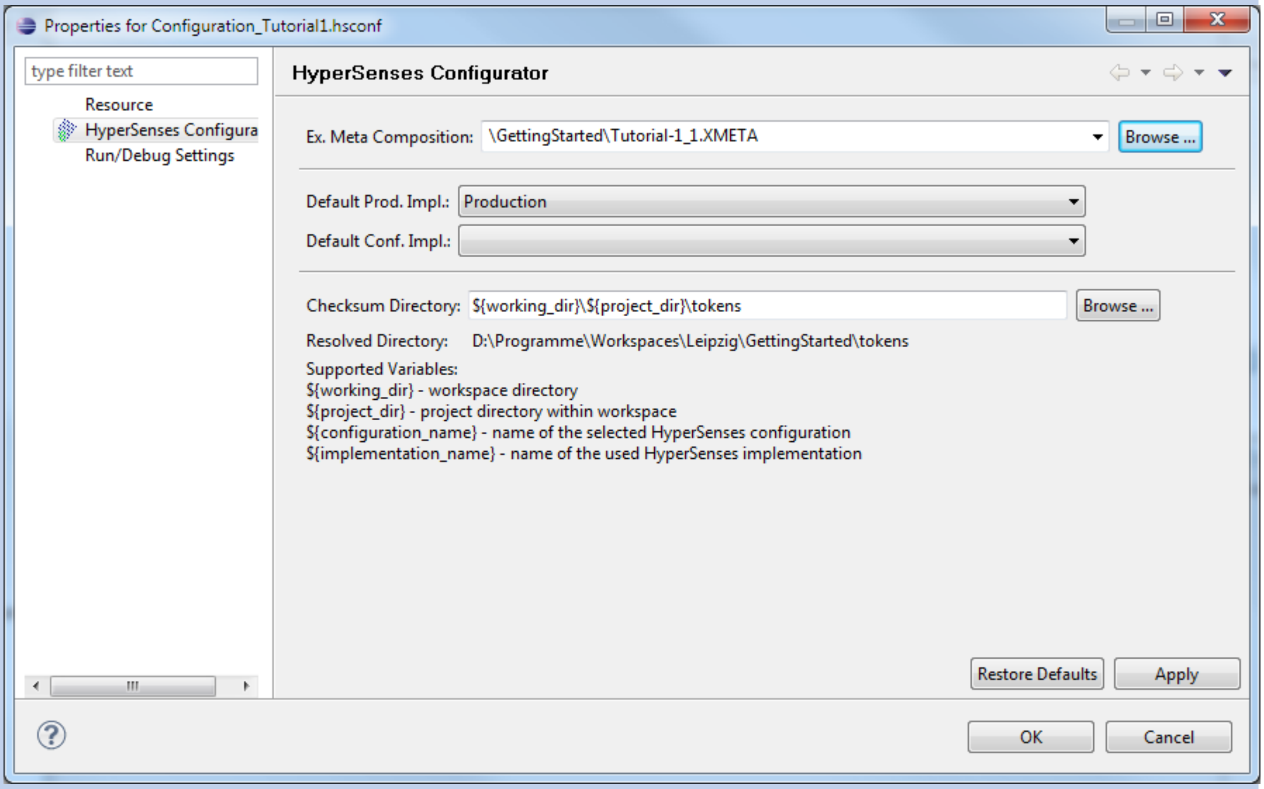
Now open the file 'Configuration_Tutorial1.hsconf' in the folder `GETTING STARTED\CONFIGURATION`.



Import Test Configuration
« TUTORIAL ▸ TUTORIAL1 ▸ CONFIGURATION
CONFIGURATION durchsuchen
Organisieren
Neuer Ordner
Delta
HE
DATA
TUTORIAL
TUTORIAL1
CONFIGURATION
METACOMPOSITION
SOURCES
Name
Änderungsdatum
Typ
Configuration_Tutorial1.hsconf
29.11.2011 08:38
HSCONF-
Dateiname:
Configuration_Tutorial1.hsconf
HyperSeneses Configuration (*.
Öffnen
Abbrechen

After you have imported the configuration, you can select the *Produce* via the context menu of the configuration file.

After saving, switch back to the Eclipse environment. You see a prepared configuration 'Configuration_Tutorial1.hsconf' in our project. Now we have to assign an *Executable Meta Composition* to this configuration by selecting the *Properties* entry in the context menu of this file. In the following dialog, in the 'HyperSenses Configurator' group, we assign the file 'Tutorial-1_1.XMETA' to the *Ex. Meta Composition*.


Properties for Configuration_Tutorial1.hsconf

HyperSenses Configurator

Ex. Meta Composition: \GettingStarted\Tutorial-1_1.XMETA

Default Prod. Impl.: Production

Default Conf. Impl.:

Checksum Directory: ${working_dir}\${project_dir}\tokens

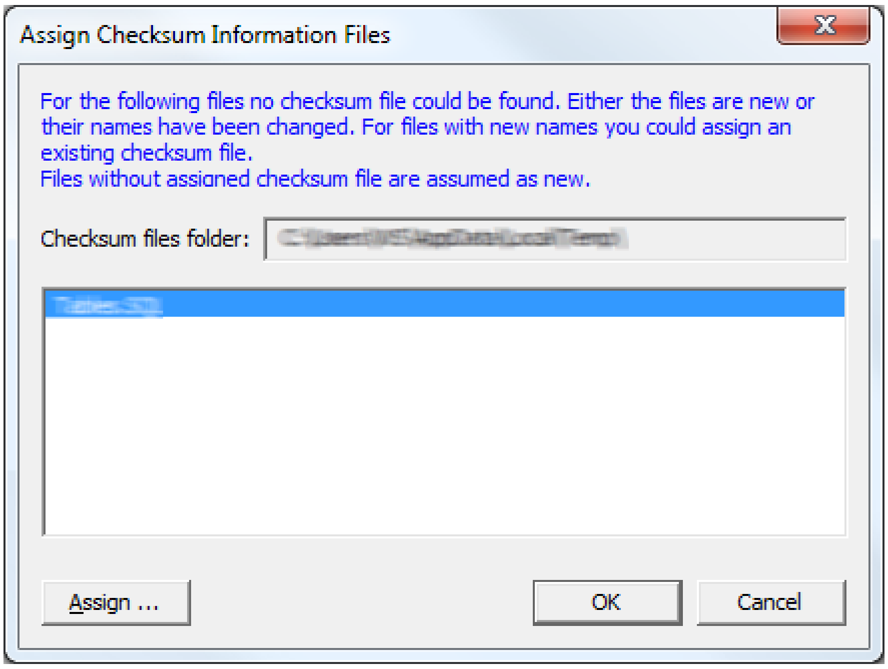
Resolved Directory: D:\Programme\Workspaces\Leipzig\GettingStarted\tokens

Supported Variables:
${working_dir} - workspace directory
${project_dir} - project directory within workspace
${configuration_name} - name of the selected HyperSenses configuration
${implementation_name} - name of the used HyperSenses implementation

Afterwards, we exit the dialog by selecting 'OK'.

After these steps, you can select the *Produce* entry via context menu call of the configuration file. Confirm the 'Assign Checksum Information Files' dialog by selecting 'OK'. After a short moment you will see the generated file *Tables.SQL* in your project folder.

A message concerning *Checksum Files* appears, select ‘OK’.


Assign Checksum Information Files

For the following files no checksum file could be found. Either the files are new or their names have been changed. For files with new names you could assign an existing checksum file.
Files without assigned checksum file are assumed as new.

Checksum files folder: [illegible]

[illegible]

Your generated code should look like this:

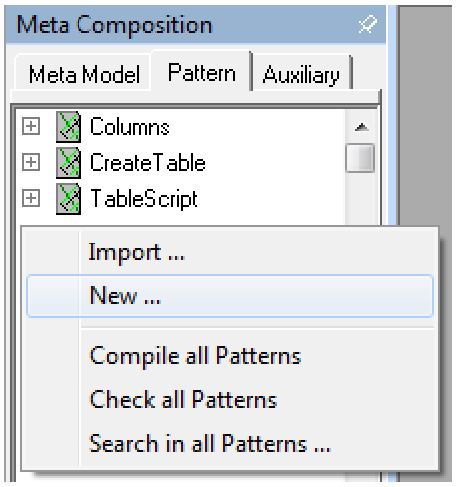
```
CREATE TABLE Person
{
  Person_ID NUMBER(10) NOT NULL,
  Name VARCHAR2(400),
  ProjectRole VARCHAR2(400),
  CONSTRAINT Person_PK PRIMARY KEY (Person_ID)
};
CREATE TABLE Repository
{
  Repository_ID NUMBER(10) NOT NULL,
  CONSTRAINT Repository_PK PRIMARY KEY (Repository_ID)
};
CREATE TABLE Project
{
  Project_ID NUMBER(10) NOT NULL,
  Name VARCHAR2(400),
  CONSTRAINT Project_PK PRIMARY KEY (Project_ID)
};
CREATE TABLE WorkItem
{
  WorkItem_ID NUMBER(10) NOT NULL,
  Opened DATE,
  Closed DATE,
  Description VARCHAR2(400),
  CONSTRAINT WorkItem_PK PRIMARY KEY (WorkItem_ID)
};
CREATE Table FileAttachement
{
  FileAttachemend_ID NUMBER(10) NOT NULL,
  Path VARCHAR2(400),
  CONSTRAINT FileAttachement_PK PRIMARY KEY(FileAttachement_ID)
};
```

Exercise 2

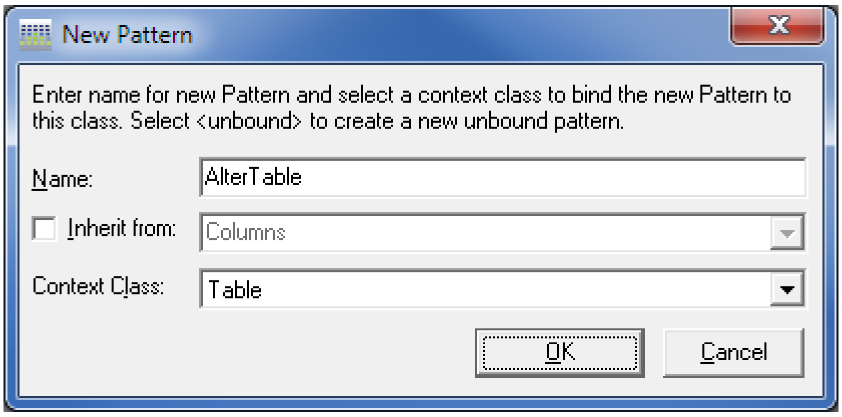
For the second exercise you can use your *MetaComposition* or you open the prepared *MetaComposition* Tutorial-1_2.META.

Creating New Pattern AlterTable

Now we want to generalize the SQL code for *AlterTable*. First, we have to import the origin SQL code. For this, we switch to the 'Pattern' tab. Select 'New Pattern…' via context menu in the pattern tree.



Enter 'AlterTable' as name with the context class 'Table' in the 'New Pattern' dialog.


New Pattern

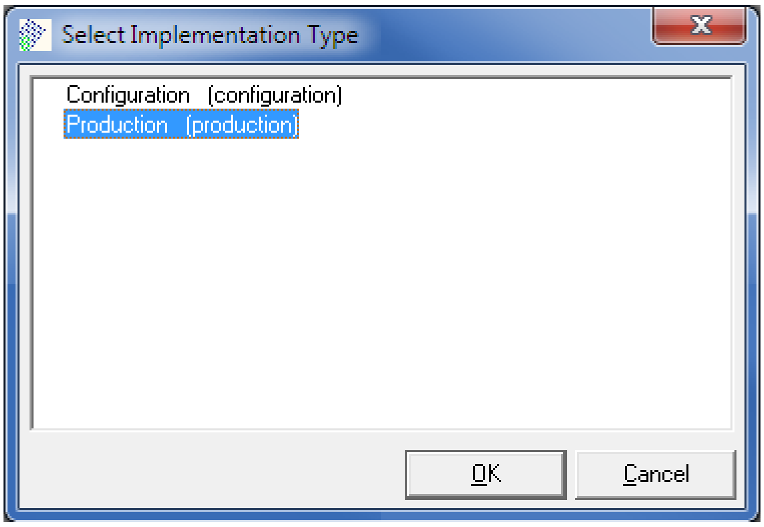
Enter name for new Pattern and select a context class to bind the new Pattern to this class. Select <unbound> to create a new unbound pattern.

Name: AlterTable

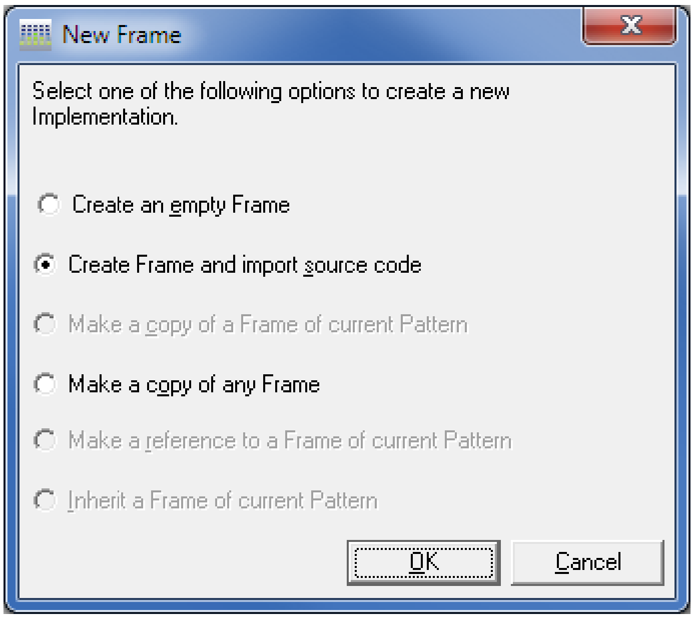
☐ Inherit from: Columns

Context Class: Table

As *Implementation Type* we select ‘Production (Production)’.



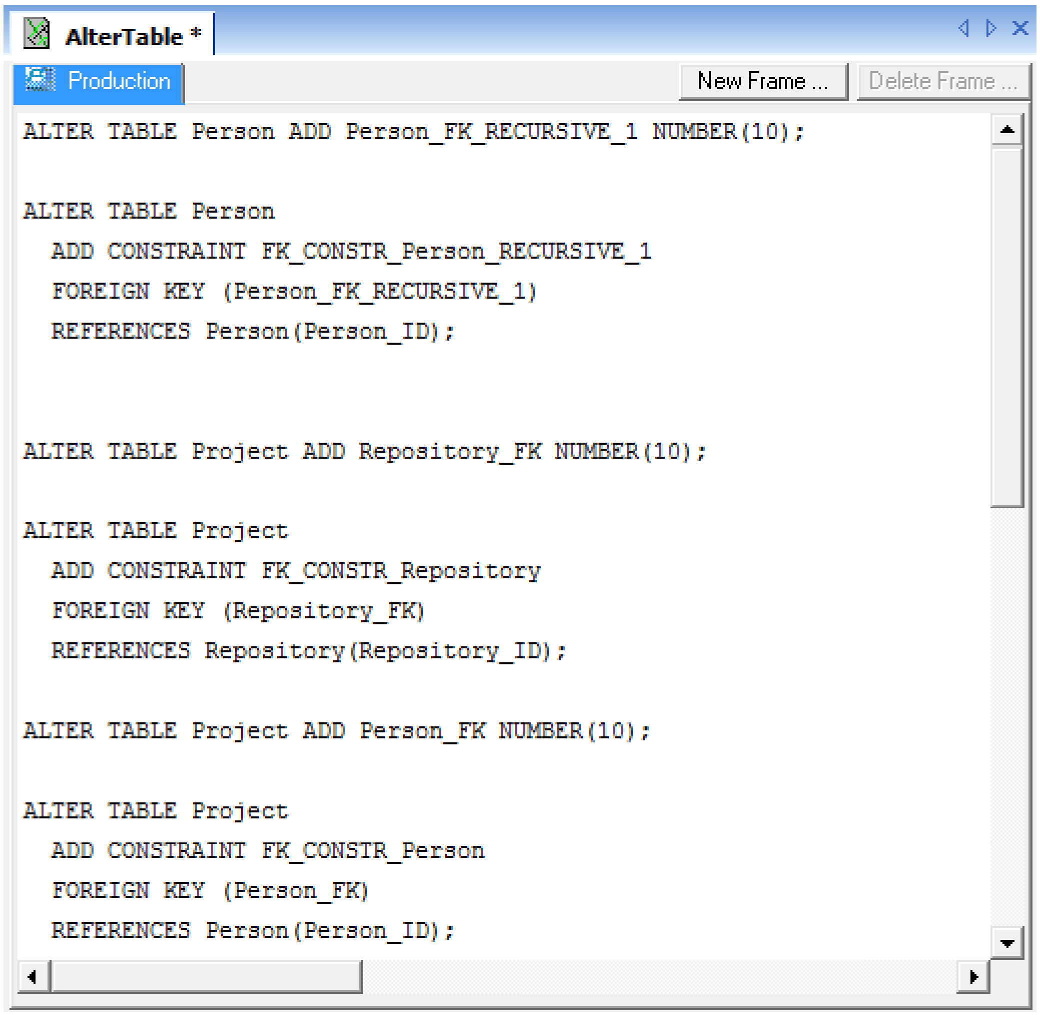
In the next dialog 'New Frame', we want to import a completed SQL code from 'AlterTable.txt'. For this, we select *Create Frame and import source code*. Then, switch to the folder `TUTORIAL\GETTING STARTED\SOURCES` and select 'AlterTable.txt'.



Select one of the following options to create a new Implementation.

- ( ) Create an empty Frame
- [✓ Create Frame and import source code]
- ( ) Make a copy of a Frame of current Pattern
- ( ) Make a copy of any Frame
- ( ) Make a reference to a Frame of current Pattern
- ( ) Inherit a Frame of current Pattern

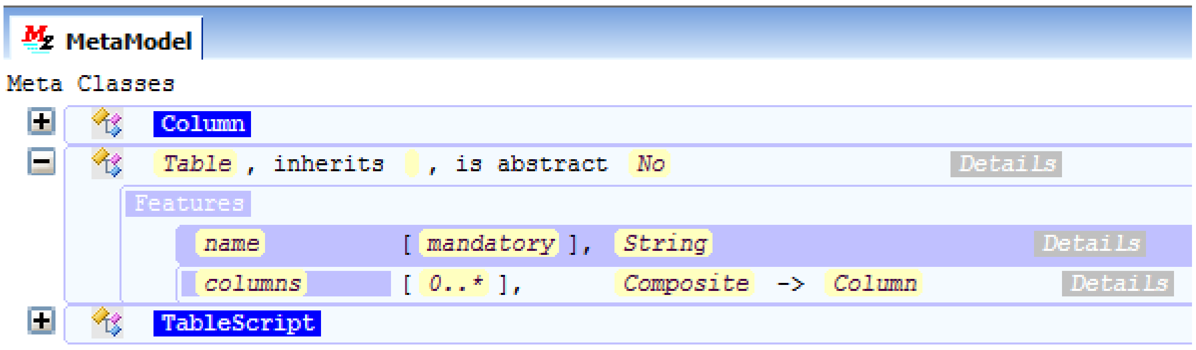
Now you have created the *AlterTable* pattern that is bound to the context class *Table* and imported the already completed SQL code. Don’t forget to save your changes.



```
ALTER TABLE Person ADD Person_FK_RECURSIVE_1 NUMBER(10);

ALTER TABLE Person
  ADD CONSTRAINT FK_CONSTR_Person_RECURSIVE_1
  FOREIGN KEY (Person_FK_RECURSIVE_1)
  REFERENCES Person(Person_ID);


ALTER TABLE Project ADD Repository_FK NUMBER(10);

ALTER TABLE Project
  ADD CONSTRAINT FK_CONSTR_Repository
  FOREIGN KEY (Repository_FK)
  REFERENCES Repository(Repository_ID);

ALTER TABLE Project ADD Person_FK NUMBER(10);

ALTER TABLE Project
  ADD CONSTRAINT FK_CONSTR_Person
  FOREIGN KEY (Person_FK)
  REFERENCES Person(Person_ID);
```

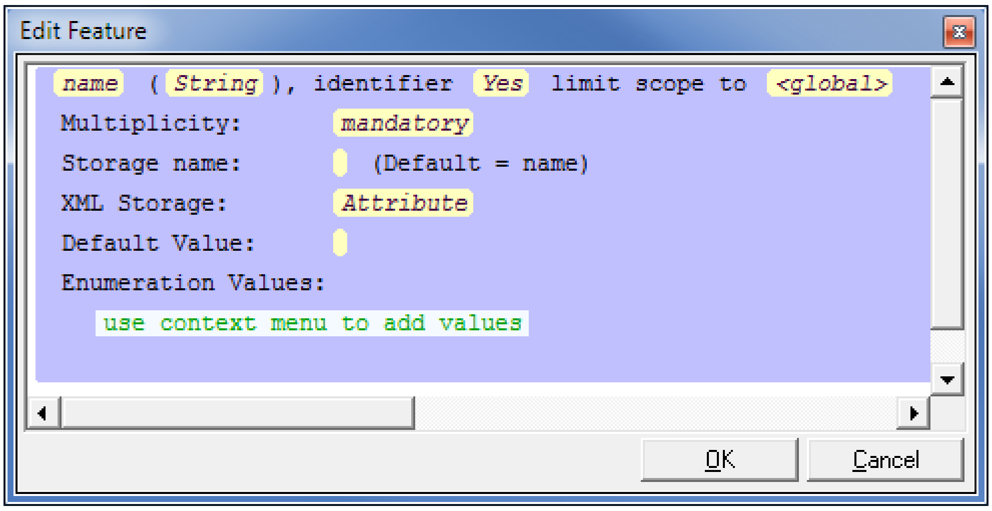
Extending the Meta Model

If you look at the *AlterTable* pattern, you notice that there are 'FOREIGN KEY' relations. But these relations are not part of our meta model yet.
Switch to the 'MetaModel' tab, open the `Table` class.



```
MetaModel
Meta Classes
  Column
  Table , inherits  , is abstract No                         Details
    Features
      name        [ mandatory ],  String                     Details
      columns     [ 0..* ],       Composite  ->  Column      Details
  TableScript
```

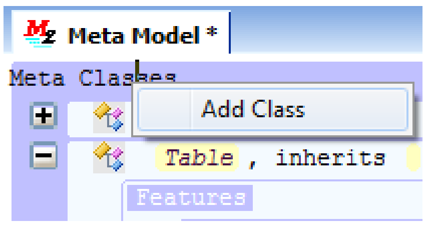
For feature *name* open its details by clicking and set the `identifier` to `Yes`, in this way the tables can later be clearly referenced.
Exit the dialog by selecting 'OK'.



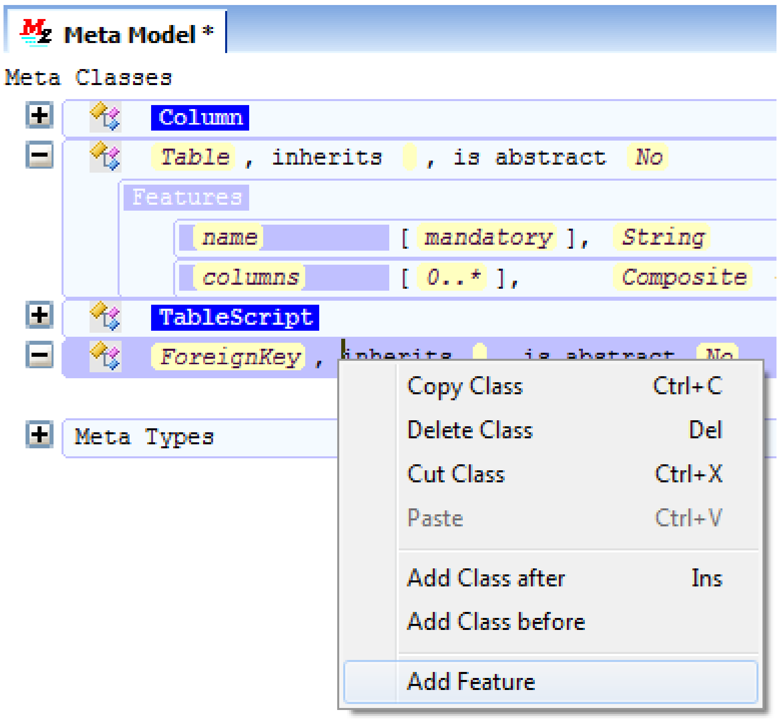
```
Edit Feature
name ( String ), identifier Yes limit scope to <global>
Multiplicity:         mandatory
Storage name:            (Default = name)
XML Storage:          Attribute
Default Value:
Enumeration Values:
    use context menu to add values
```

Creating Meta Class and Meta Feature

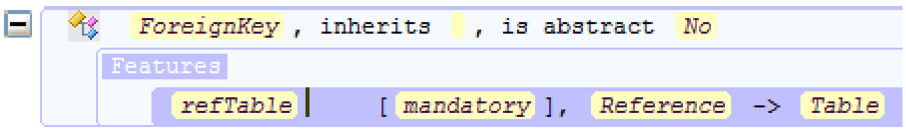
Then, right-click on 'MetaClasses' and select 'Add Class' in the context menu.



The new class gets the name `ForeignKey`. You create a `refTable` feature via 'Add Feature' in the context menu of this class.

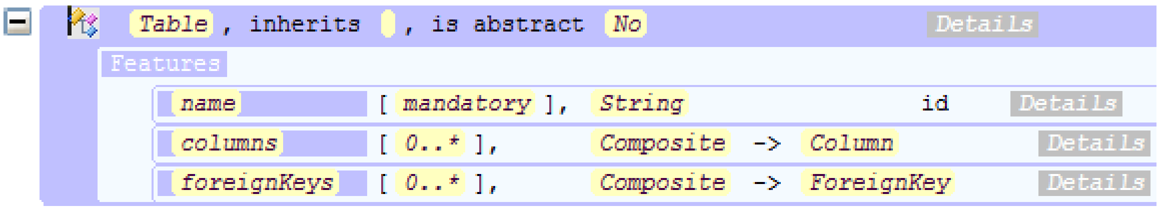


This feature is a mandatory reference to `Table`.



```
ForeignKey , inherits , is abstract No
  Features
    refTable [ mandatory ], Reference -> Table
```

Additionally, in the class `Table` create a feature named `foreignKeys` as `Composite` of the `ForeignKey` class with the multiplicity '0..*'.

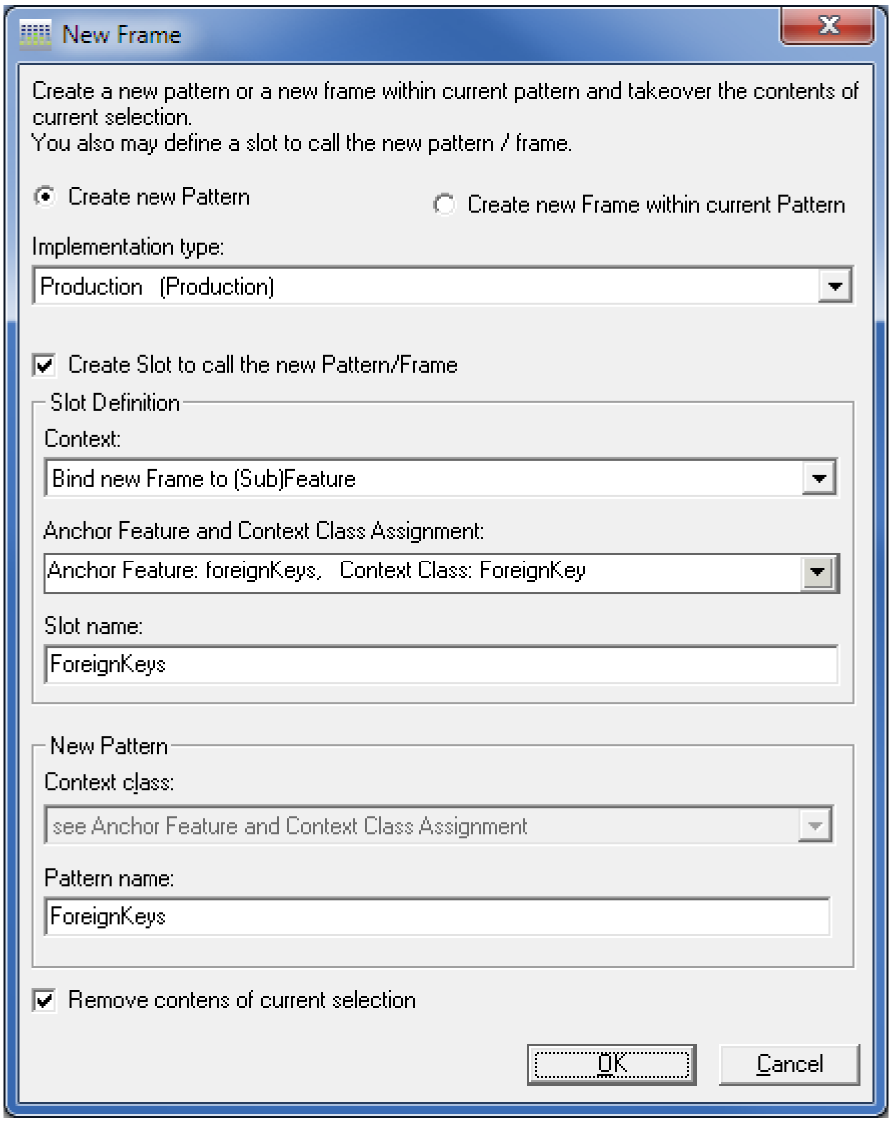


```
Table , inherits , is abstract No                    Details
  Features
    name        [ mandatory ], String               id   Details
    columns     [ 0..* ],      Composite -> Column       Details
    foreignKeys [ 0..* ],      Composite -> ForeignKey   Details
```

Now we have successfully extended the Meta model. Time for saving the changes. ☺

Generalizing AlterTable

Switch back to the 'Pattern' tab and open the *AlterTable* pattern. We delete the unnecessary, ‘double’ code (retain the first 6 lines). Now we want to display the Foreign Key relations of tables, it is thereby noticeable that the *AlterTable* pattern is bound to the context class *Table*. That means we have to relocate the SQL code to a SubPattern with the context class *ForeignKey*. For this purpose, we have to create a pattern call by marking the complete SQL code and call again 'Create new Frame/Pattern from selection…' using the context menu. Select then the following values:


New Frame

Create a new pattern or a new frame within current pattern and takeover the contents of current selection.
You also may define a slot to call the new pattern / frame.

[✓ Create new Pattern] Create new Frame within current Pattern

Implementation type: Production (Production)

[✓] Create Slot to call the new Pattern/Frame

Slot Definition

Context: Bind new Frame to (Sub)Feature

Anchor Feature and Context Class Assignment: Anchor Feature: foreignKeys, Context Class: ForeignKey

Slot name: ForeignKeys

New Pattern

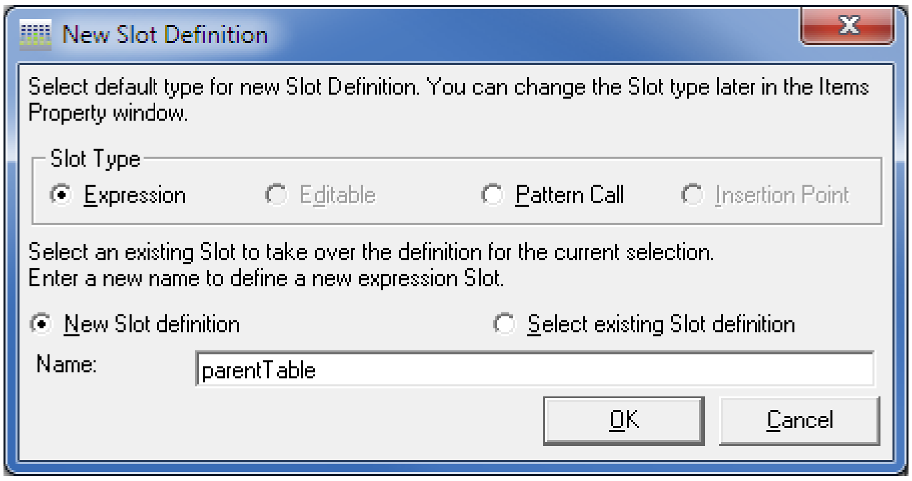
Context class: see Anchor Feature and Context Class Assignment

Pattern name: ForeignKeys

[✓] Remove contens of current selection

Creating New Slots

In the newly created pattern `ForeignKeys` we have to create slots for all variable parts, these are the parent table and the reference table. Both have the same name in our code, because we have kept the recursive call.
First, create a slot to the parent table name 'Person' which is located directly after *ALTER TABLE*. 'parentTable' would be a suitable slot name.



New Slot Definition

Select default type for new Slot Definition. You can change the Slot type later in the Items Property window.

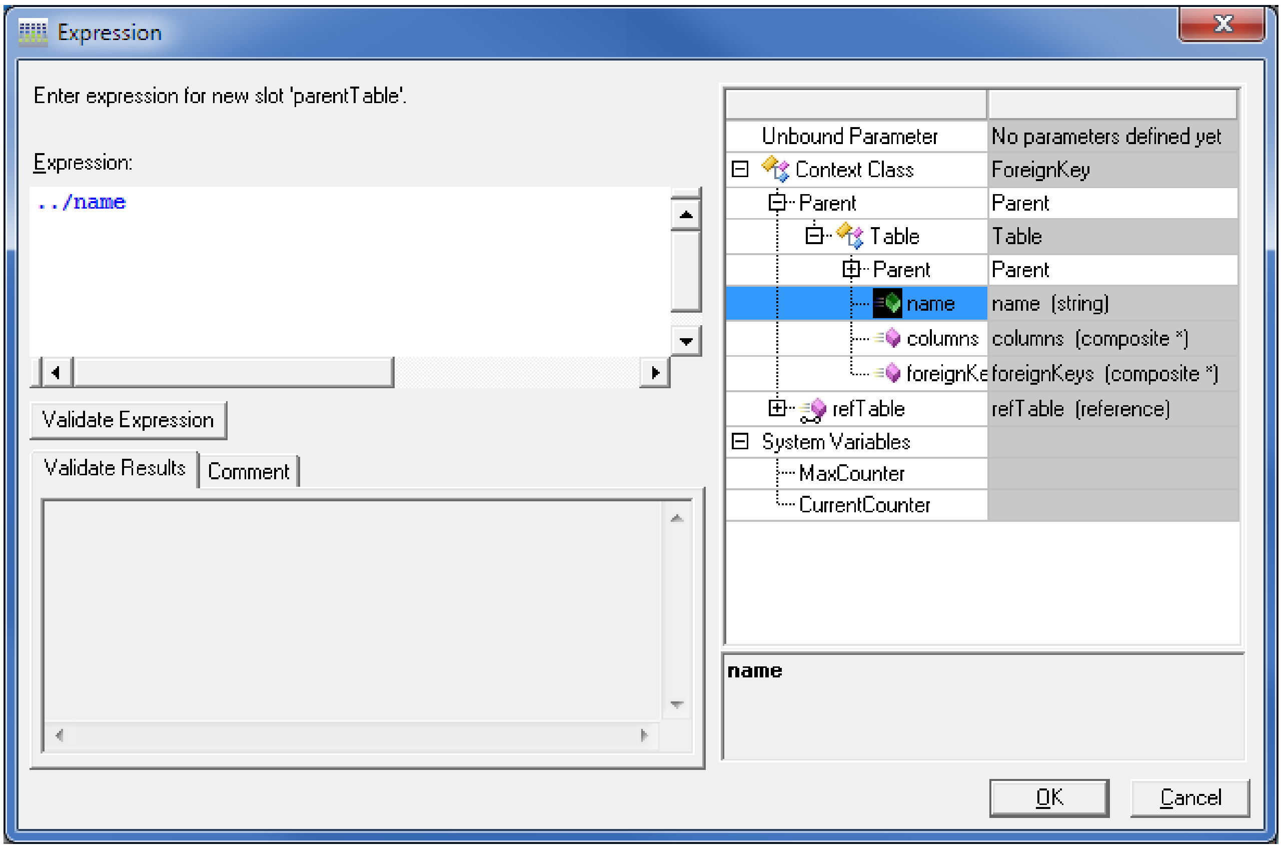
Slot Type: [✓ Expression] Editable Pattern Call Insertion Point

Select an existing Slot to take over the definition for the current selection.
Enter a new name to define a new expression Slot.

[✓ New Slot definition] Select existing Slot definition

Name: parentTable

Within the meta model, *Table* is the parent class of `ForeignKey`, thus open *Parent* → *Table* (in the 'Expression' dialog on the right side) and select 'name (string)'.



Expression

Enter expression for new slot 'parentTable'.

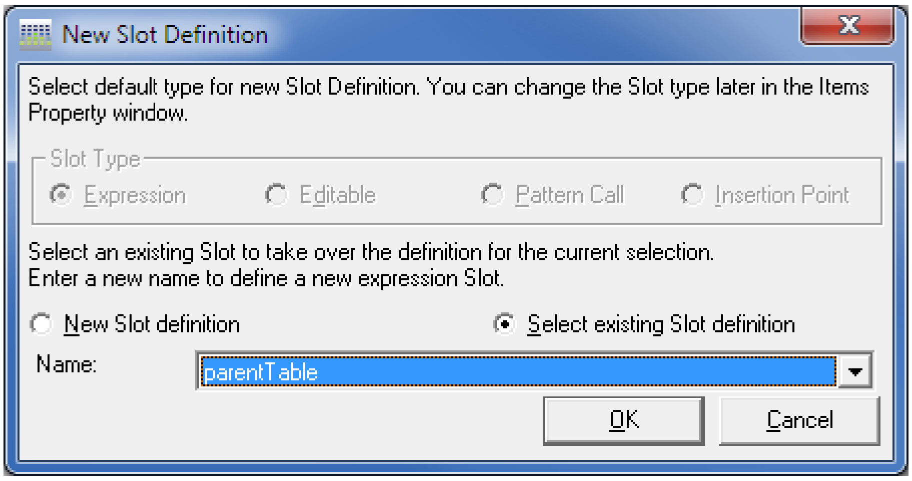
Expression:

```
../name
```

Unbound Parameter	No parameters defined yet
Context Class	ForeignKey
Parent	Parent
Table	Table
Parent	Parent
name	name (string)
columns	columns (composite *)
foreignKe	foreignKeys (composite *)
refTable	refTable (reference)
System Variables	
MaxCounter	
CurrentCounter	

name

Another occurrence of the parent table name can be found in the next line directly succeeding *ALTER TABLE*. Mark *Person* and perform *Mark Slot…* and then 'Select existing Slot definition'. Select the before created slot 'parentTable' and press 'OK'.



New Slot Definition

Select default type for new Slot Definition. You can change the Slot type later in the Items Property window.

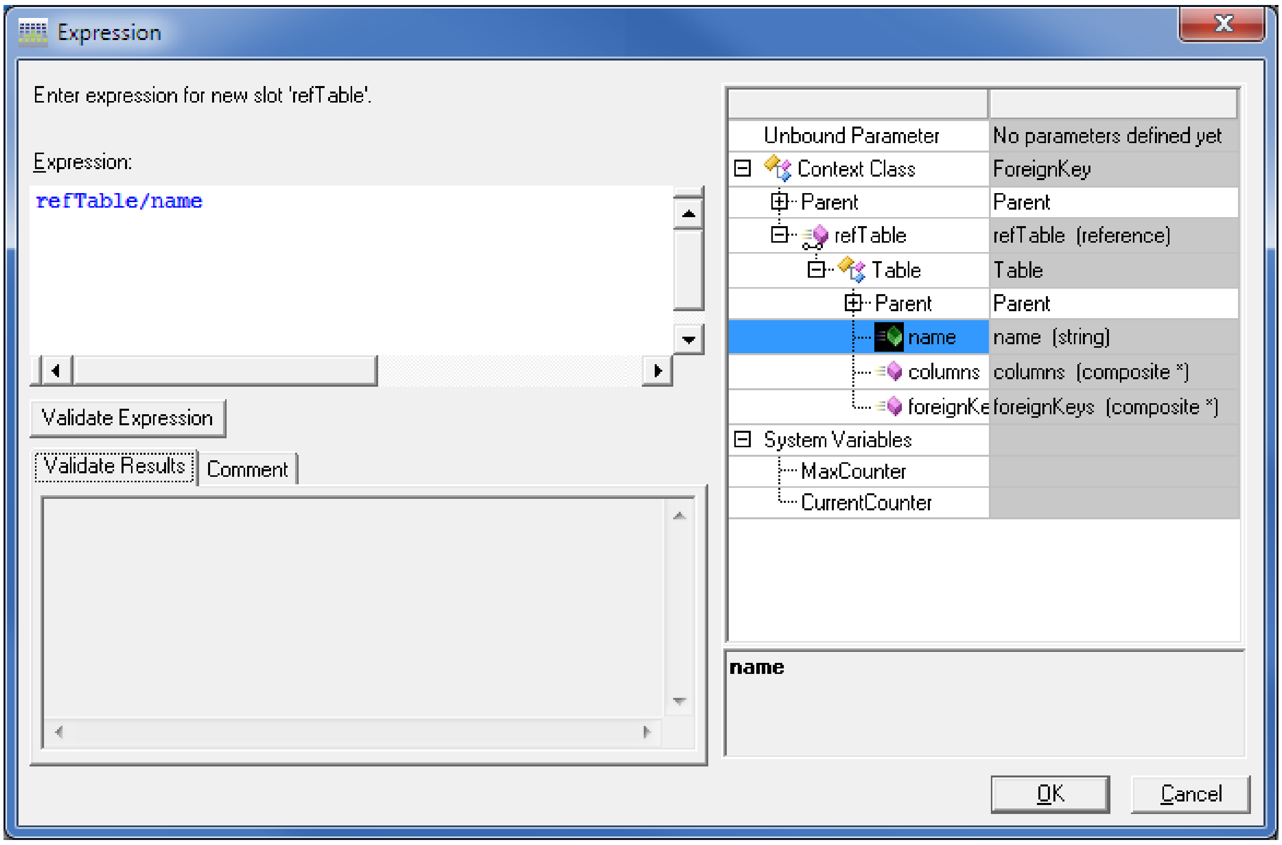
Slot Type: [✓ Expression] Editable Pattern Call Insertion Point

Select an existing Slot to take over the definition for the current selection.
Enter a new name to define a new expression Slot.

New Slot definition [✓ Select existing Slot definition]

Name: parentTable

Now repeat these steps for the referenced table, which are all remaining occurences of *Person*. In contrast to the slots before, the new slots (slot name: 'refTable') have to calculate the name of the referenced table. Within the 'Expression' dialog select *refTable* → *Table* → *name (string)* to calculate the correct name.



Expression

Enter expression for new slot 'refTable'.

Expression:

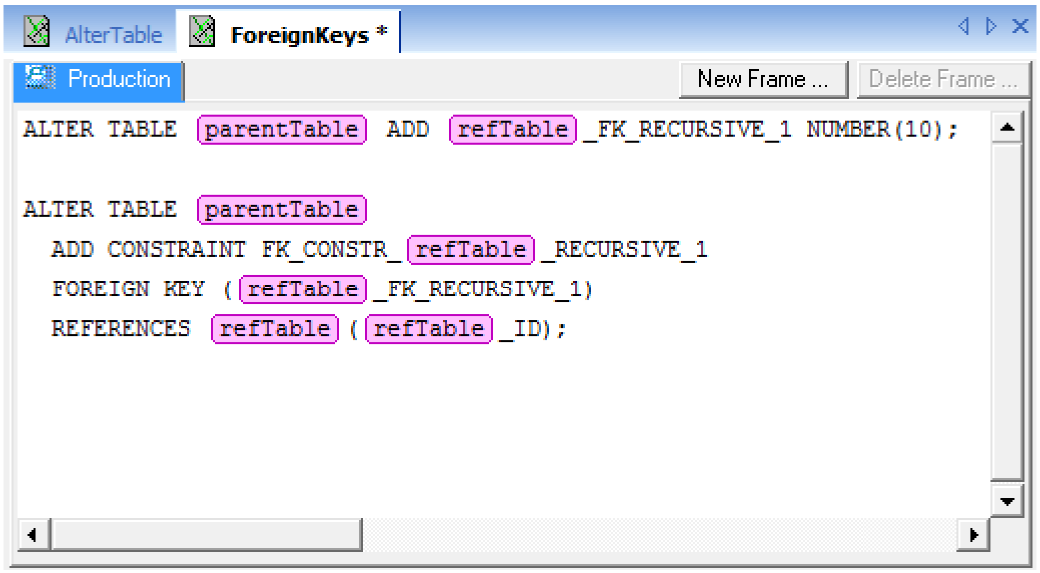
```
refTable/name
```

Unbound Parameter	No parameters defined yet
Context Class	ForeignKey
Parent	Parent
refTable	refTable (reference)
Table	Table
Parent	Parent
name	name (string)
columns	columns (composite *)
foreignKe	foreignKeys (composite *)
System Variables	
MaxCounter	
CurrentCounter	

name

AlterTable: The First Result

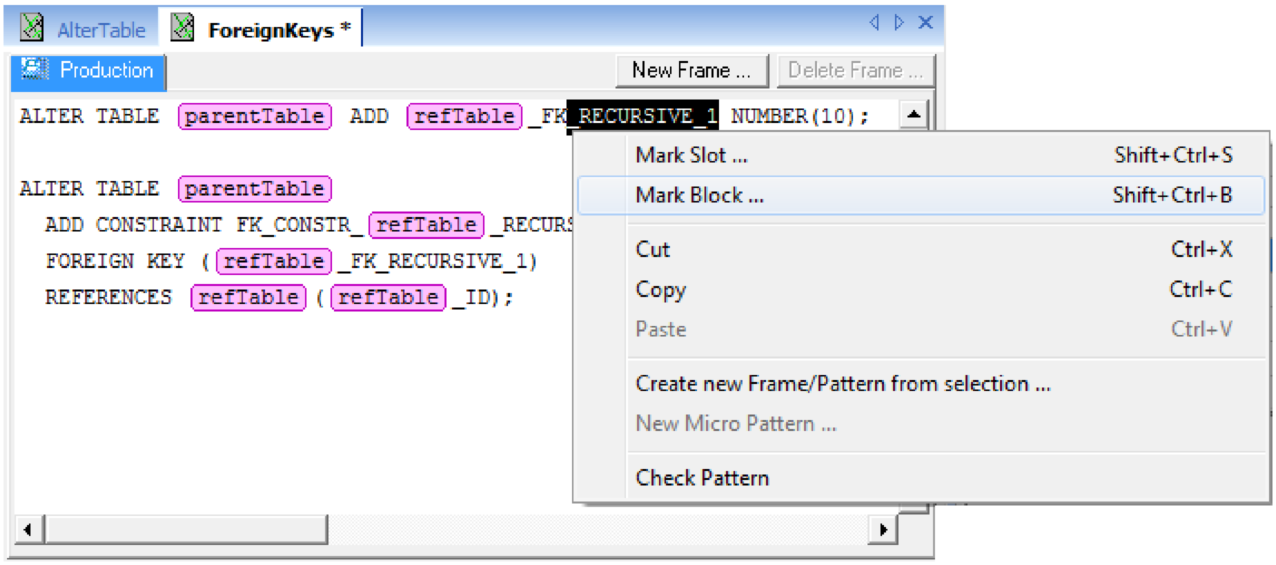
Afterwards, your pattern should look like this:



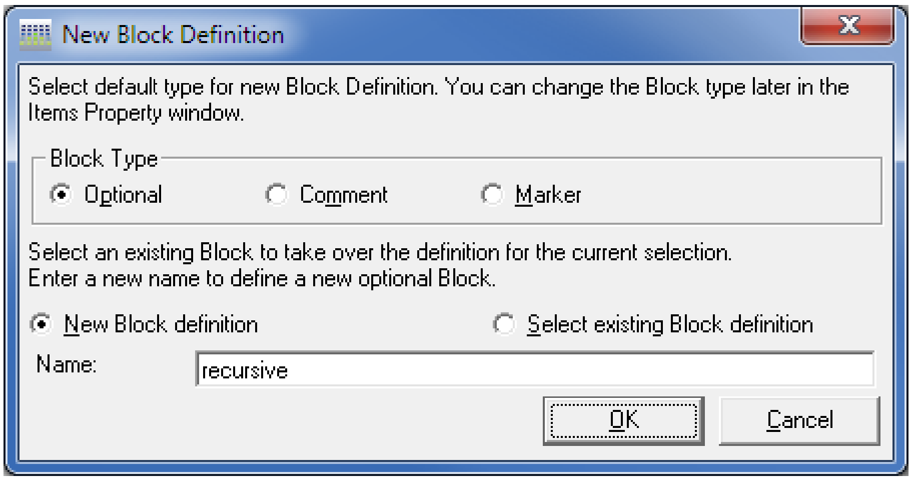
```
ALTER TABLE parentTable ADD refTable _FK_RECURSIVE_1 NUMBER(10);

ALTER TABLE parentTable
  ADD CONSTRAINT FK_CONSTR_ refTable _RECURSIVE_1
  FOREIGN KEY ( refTable _FK_RECURSIVE_1)
  REFERENCES refTable ( refTable _ID);
```

Creating Optional Blocks

There are tables without recursive calls, therefore we have to pack the recursive calls `_RECURSIVE_1` into optional blocks.



```
ALTER TABLE parentTable ADD refTable _FK_RECURSIVE_1 NUMBER(10);

ALTER TABLE parentTable
  ADD CONSTRAINT FK_CONSTR_ refTable _RECURS
  FOREIGN KEY ( refTable _FK_RECURSIVE_1)
  REFERENCES refTable ( refTable _ID);
```



New Block Definition

Select default type for new Block Definition. You can change the Block type later in the Items Property window.

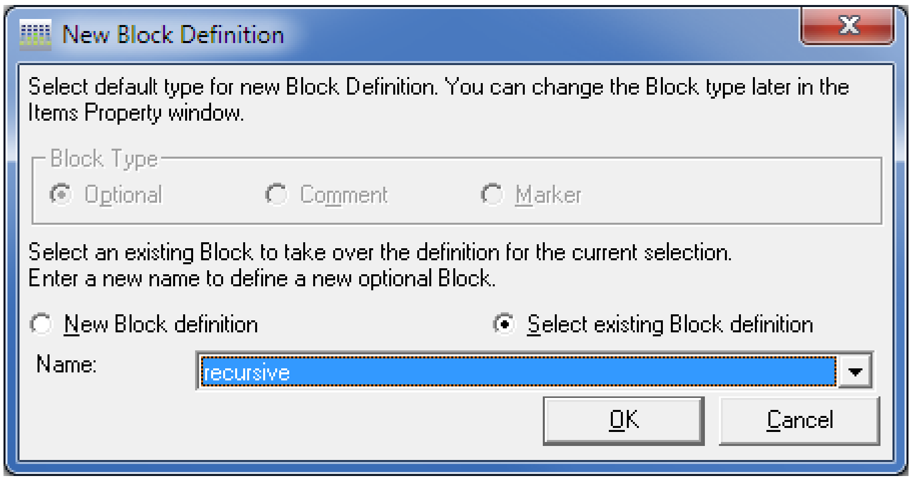
Block Type: [✓ Optional] Comment Marker

Select an existing Block to take over the definition for the current selection.
Enter a new name to define a new optional Block.

[✓ New Block definition] Select existing Block definition

Name: recursive

As the same block has to be created three times, you can select “Select existing Block definition” for the second and third creation of the block.


New Block Definition

Select default type for new Block Definition. You can change the Block type later in the Items Property window.

Block Type: [● Optional] ○ Comment ○ Marker

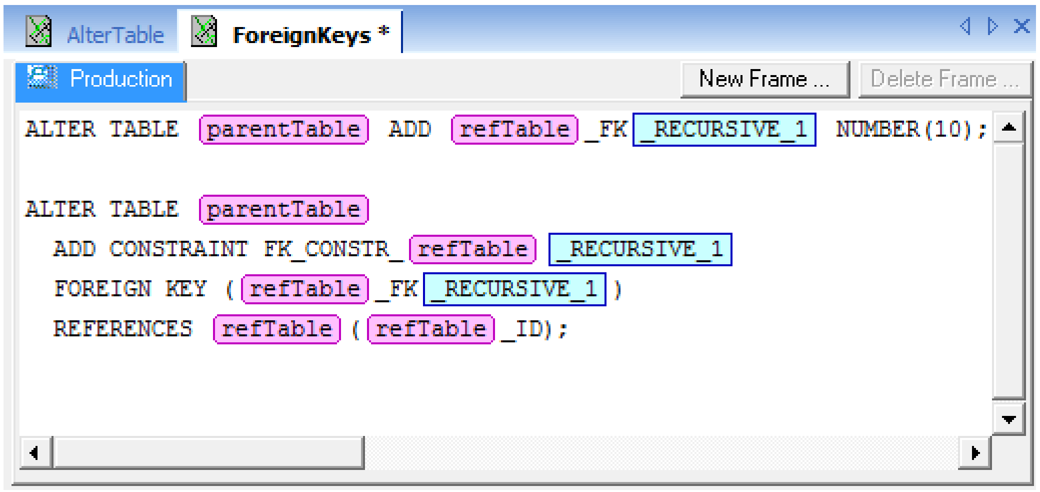
Select an existing Block to take over the definition for the current selection.
Enter a new name to define a new optional Block.

○ New Block definition [● Select existing Block definition]

Name: recursive

AlterTable: The Second Result

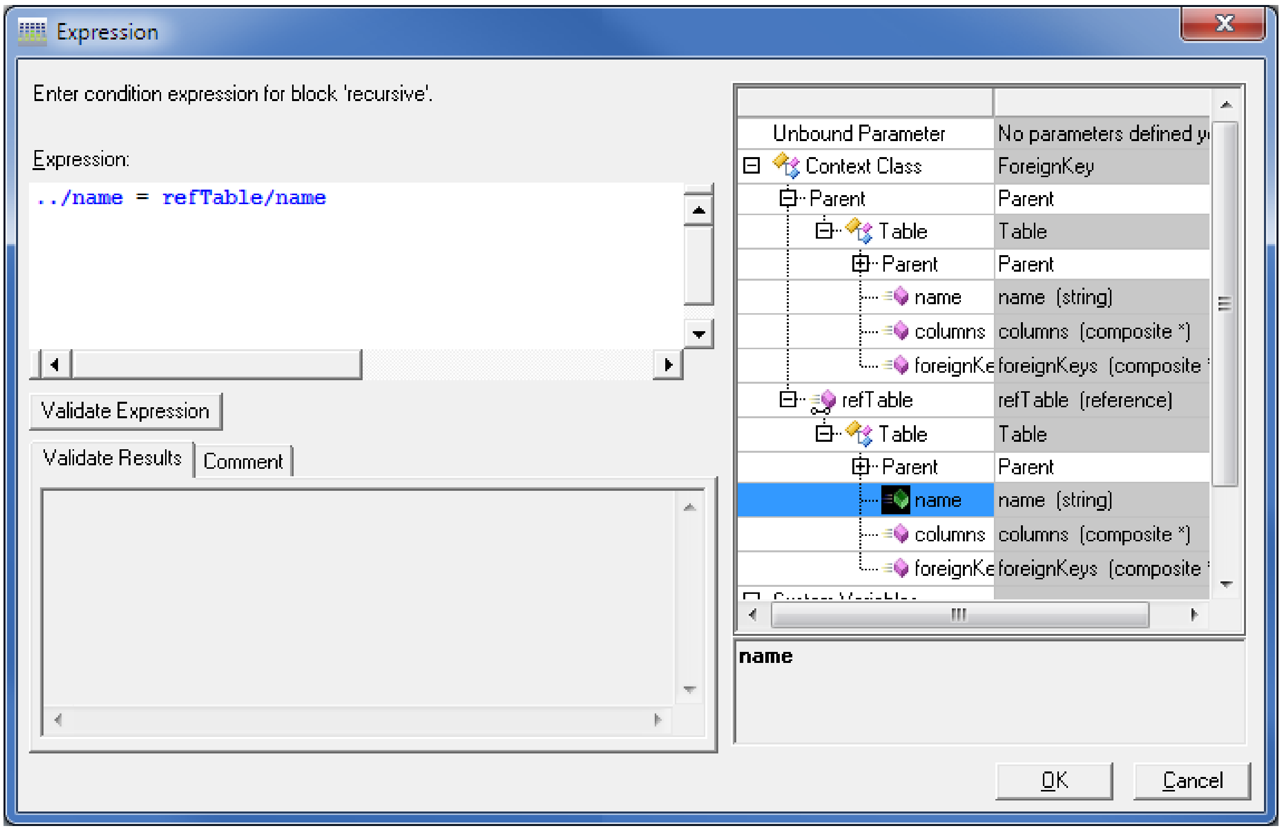
Afterwards, our pattern looks like this:



```
ALTER TABLE parentTable ADD refTable_FK_RECURSIVE_1 NUMBER(10);

ALTER TABLE parentTable
  ADD CONSTRAINT FK_CONSTR_refTable_RECURSIVE_1
  FOREIGN KEY (refTable_FK_RECURSIVE_1)
  REFERENCES refTable(refTable_ID);
```

Creating Block Conditions

These blocks may only be generated if there is a recursive call, i.e. if the parent table is equal to the reference table. This condition is still missing. Right-click on one of the blocks, select 'Modify Condition…' and create the following condition:



Expression

Enter condition expression for block 'recursive'.

Expression:

```
../name = refTable/name
```

Unbound Parameter	No parameters defined y
Context Class	ForeignKey
Parent	Parent
Table	Table
Parent	Parent
name	name (string)
columns	columns (composite *)
foreignKe	foreignKeys (composite
refTable	refTable (reference)
Table	Table
Parent	Parent
name	name (string)
columns	columns (composite *)
foreignKe	foreignKeys (composite

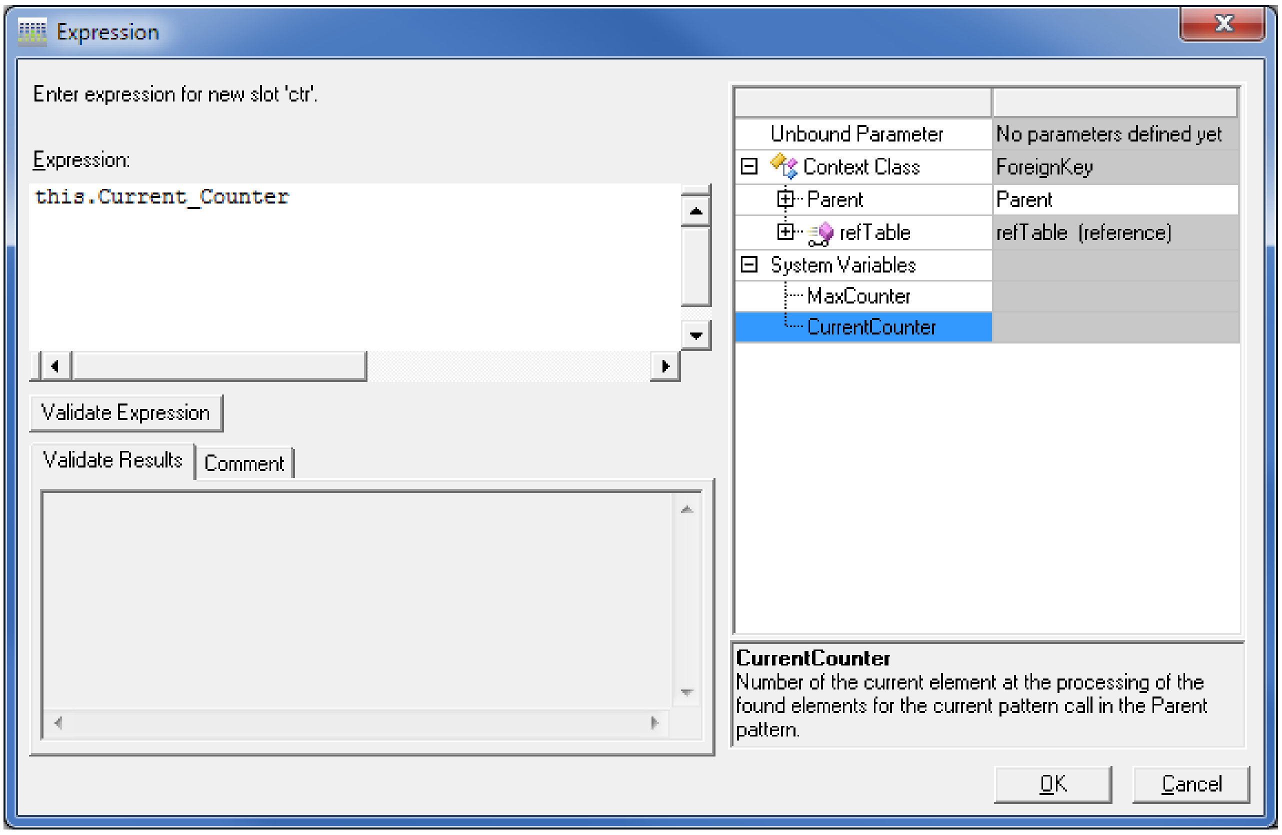
name

As the blocks where created by reference, it is sufficient to define the condition only once.

Don’t forget to save your changes! ☺

Creating a Counter

Now, the recursive calls have to be enumerated, to do so we create a slot ('ctr') which replaces the "1" in the optional blocks. The system variable *CurrentCounter* is assigned to this slot.



Enter expression for new slot 'ctr'.

Expression:

```
this.Current_Counter
```

Unbound Parameter	No parameters defined yet
Context Class	ForeignKey
Parent	Parent
refTable	refTable (reference)
System Variables	
MaxCounter	
CurrentCounter	

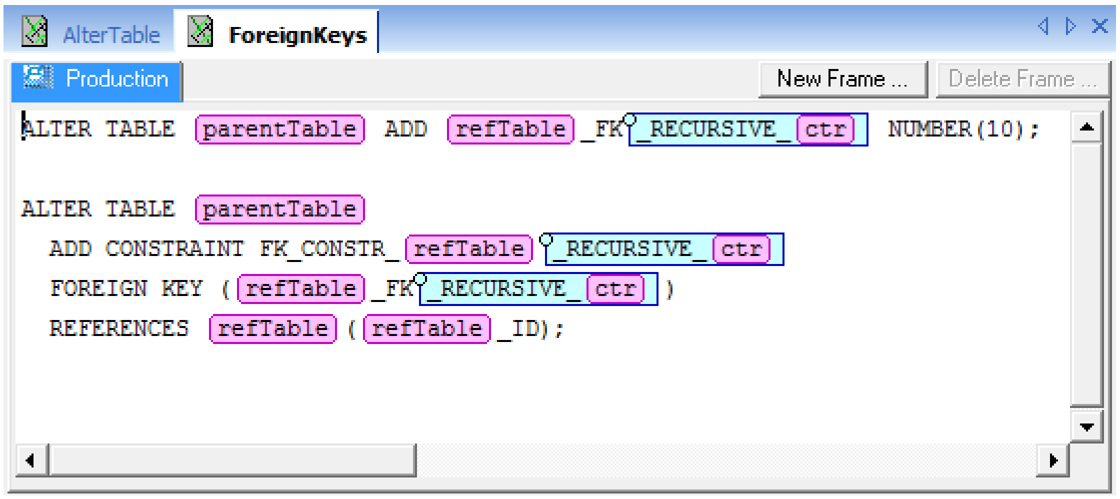
CurrentCounter
Number of the current element at the processing of the found elements for the current pattern call in the Parent pattern.

HyperSenses provides the two system variables *CurrentCounter* and *MaxCounter*. Both variables are local counters. *CurrentCounter* counts how often the frame is executed in the same context. *MaxCounter* specifies how often the frame has been overall executed within a context. That means, if *CurrentCounter* and *MaxCounter* are used in one frame, they display the same value at the final call.

Replace all occurrences of "1" by this new slot.

AlterTable: The Third Result

Now we have created the following pattern:



```
ALTER TABLE parentTable ADD refTable _FK_RECURSIVE_ctr NUMBER(10);

ALTER TABLE parentTable
  ADD CONSTRAINT FK_CONSTR_refTable _RECURSIVE_ctr
  FOREIGN KEY (refTable _FK_RECURSIVE_ctr )
  REFERENCES refTable (refTable _ID);
```

The production patterns are complete!

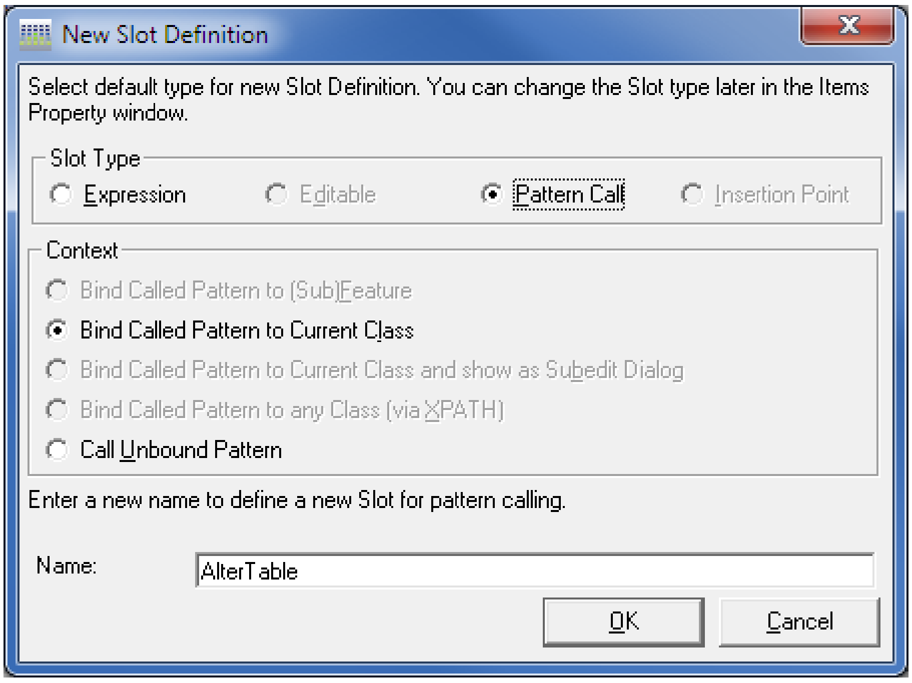
Please save your changes...

Creating a Test Configuration

If you want to test your work, you can do this by using a test configuration like in the first exercise. But first we have to assign the new *AlterTable* pattern to the Factory Pattern to ensure that it is considered for production.

To ensure that you are able to use the production patterns for the code production, you have to specifiy the following information: which files must be created, the names of these files, where to store them and which production patterns are relevant for a file. For this description we use *Factory Patterns*.

Switch to 'Auxiliary'. Open "Implementation Types for Production" and then open the factory pattern 'Production' by a double-click. Insert a new line below the pattern call *CreateTable*. Create a new pattern call ('Mark Slot…') via context menu.



Select default type for new Slot Definition. You can change the Slot type later in the Items Property window.

Slot Type

Expression [✓ Pattern Call] Editable Insertion Point

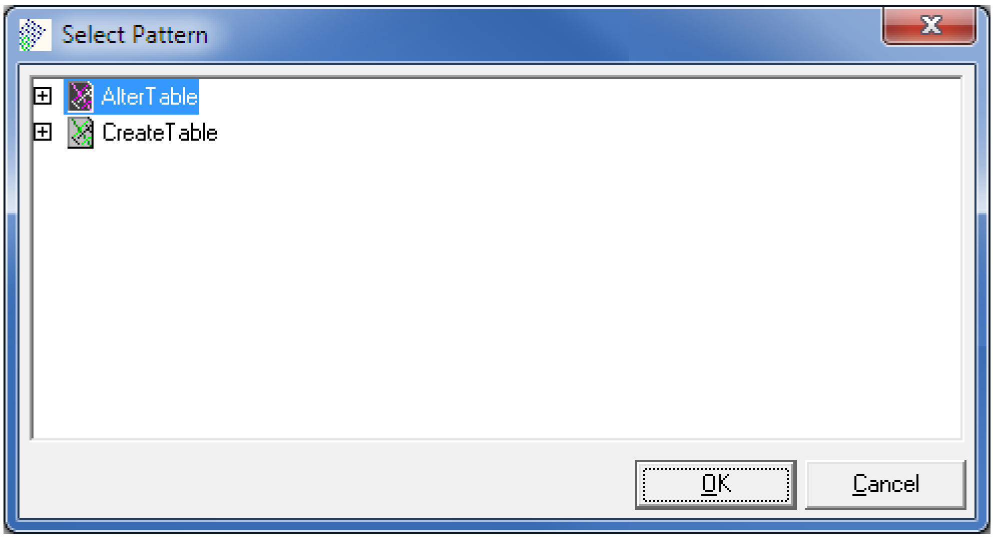
Context

Bind Called Pattern to (Sub)Feature
[✓ Bind Called Pattern to Current Class]
Bind Called Pattern to Current Class and show as Subedit Dialog
Bind Called Pattern to any Class (via XPATH)
Call Unbound Pattern

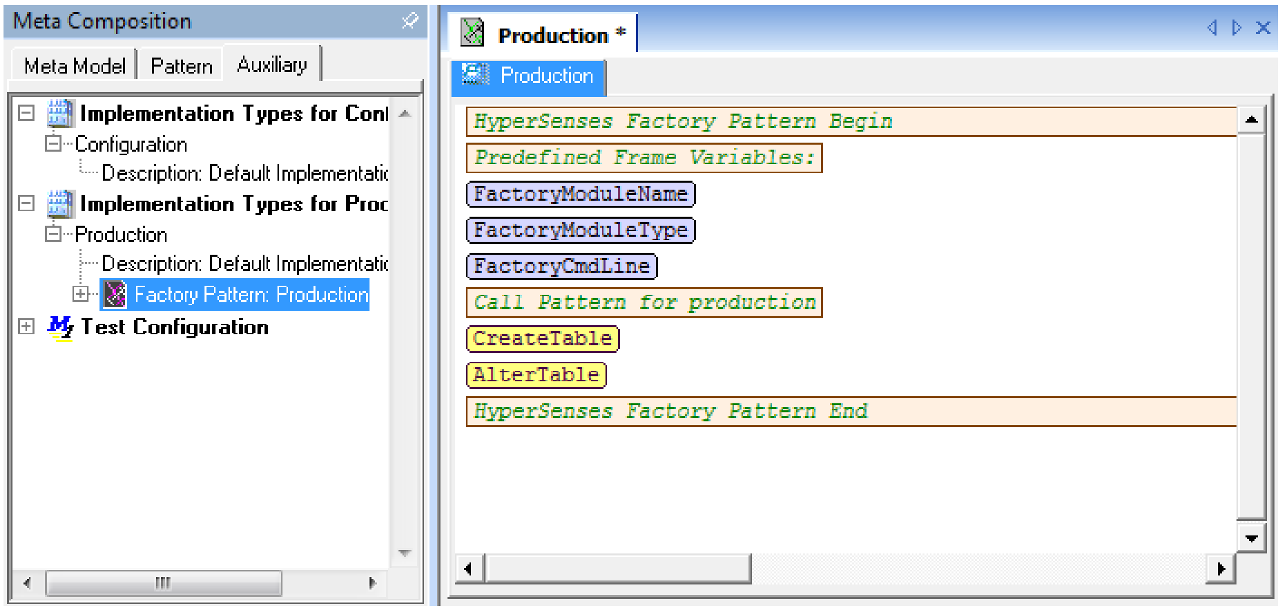
Enter a new name to define a new Slot for pattern calling.

Name: AlterTable

Select *AlterTable*:



Now, your Factory Pattern should look like this:


Meta Composition
Meta Model
Pattern
Auxiliary
Implementation Types for Conf
Configuration
Description: Default Implementatic
Implementation Types for Proc
Production
Description: Default Implementatic
Factory Pattern: Production
Test Configuration
Production *
Production

```
HyperSenses Factory Pattern Begin
Predefined Frame Variables:
FactoryModuleName
FactoryModuleType
FactoryCmdLine
Call Pattern for production
CreateTable
AlterTable
HyperSenses Factory Pattern End
```

After saving your changes, you can start a 'Produce…' again on the test configuration (see Exercise 1). You then get the *AlterTable* and *CreateTable* SQL code.

Ecplise tip: Please remember that you have to assign the new XMETA to the configuration via 'Properties' dialog.

Your result should look like this:

CREATE TABLE

```
CREATE TABLE Person
{
  Person_ID NUMBER(10) NOT NULL,
  Name VARCHAR2(400),
  ProjectRole VARCHAR2(400),
  CONSTRAINT Person_PK PRIMARY KEY (Person_ID)
};
CREATE TABLE Repository
{
  Repository_ID NUMBER(10) NOT NULL,
  CONSTRAINT Repository_PK PRIMARY KEY (Repository_ID)
};
CREATE TABLE Project
{
  Project_ID NUMBER(10) NOT NULL,
  Name VARCHAR2(400),
  CONSTRAINT Project_PK PRIMARY KEY (Project_ID)
};
CREATE TABLE WorkItem
{
  WorkItem_ID NUMBER(10) NOT NULL,
  Opened DATE,
  Closed DATE,
  Description VARCHAR2(400),
  CONSTRAINT WorkItem_PK PRIMARY KEY (WorkItem_ID)
};
CREATE Table FileAttachement
{
  FileAttachemend_ID NUMBER(10) NOT NULL,
  Path VARCHAR2(400),
  CONSTRAINT FileAttachement_PK PRIMARY KEY(FileAttachement_ID)
};
```

ALTER TABLE

```
ALTER TABLE Person ADD Person_FK_RECURSIVE_1 NUMBER(10);
ALTER TABLE Person
  ADD CONSTRAINT FK_CONSTR_Person_RECURSIVE_1
  FOREIGN KEY (Person_FK_RECURSIVE_1)
  REFERENCES Person(Person_ID);
ALTER TABLE Person ADD Person_FK_RECURSIVE_2 NUMBER(10);
ALTER TABLE Person
  ADD CONSTRAINT FK_CONSTR_Prson_RECURSIVE_2
  FOREIGN KEY (Person_FK_RECURSIVE_2)
  REFERENCES Person(Person_ID);
ALTER TABLE Project ADD Repository_FK NUMBER(10);
ALTER TABLE Project
  ADD CONSTRAINT FK_CONSTR_Repository
  FOREIGN KEY (Repository_FK)
  REFERENCES Repository(Repository_ID);
ALTER TABLE Project ADD Person_FK NUMBER(10);
ALTER TABLE Project
  ADD CONSTRAINT FK_CONSTR_Person
  FOREIGN KEY (Person_FK)
  REFERENCES Person(Person_ID);
ALTER TABLE WorkItem ADD Project_FK NUMBER(10);
ALTER TABLE WorkItem
  ADD CONSTRAINT FK_CONSTR_Project
  FOREIGN KEY (Project_FK)
  REFERENCES Project(Project_ID);
ALTER TABLE FileAttachement ADD WorkItem_FK NUMBER(10);
ALTER TABLE FileAttachement
  ADD CONSTRAINT FK_CONSTR_WorkItem
  FOREIGN KEY (WorkItem_FK)
  REFERENCES WorkItem(WorkItem_ID);
```

Exercise 3

In the next exercise we want to create configurations by using a form DSL.

Open 'Tutorial-1_3.META' or continue working with your *MetaComposition*.

Creating Configuration Patterns

Configuration patterns are necessary to generate a DSL (domain-specific language). The resulting DSL is similar to an electronic form. Such a pattern contains static and dynamic parts:

- Static parts: Explanatory text, to describe which variable parts exist in the system and must or can be defined by the configuration.
- Dynamic parts: These are the parts that must be set during the creation of a configuration. They are linked to the meta model, just like the dynamic parts of the production patterns.

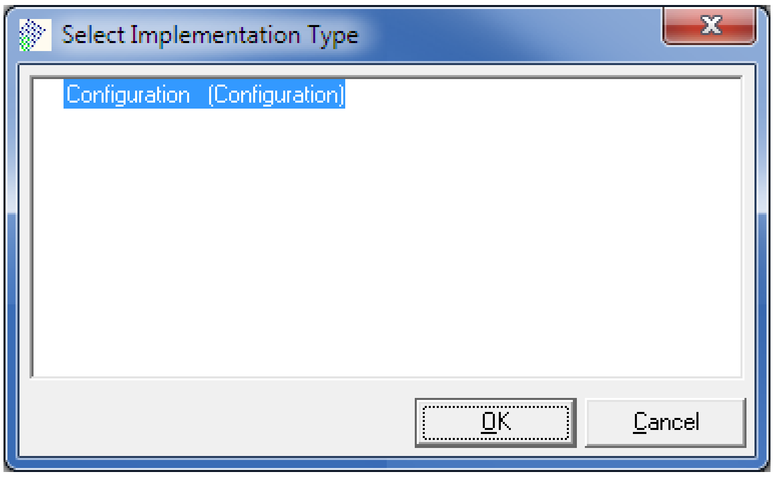
The meta model and the configuration patterns define the DSL together.

Configuration Patterns can be stand alone patterns or a part of any other pattern, in that case a configuration frame is inserted into an existing pattern.

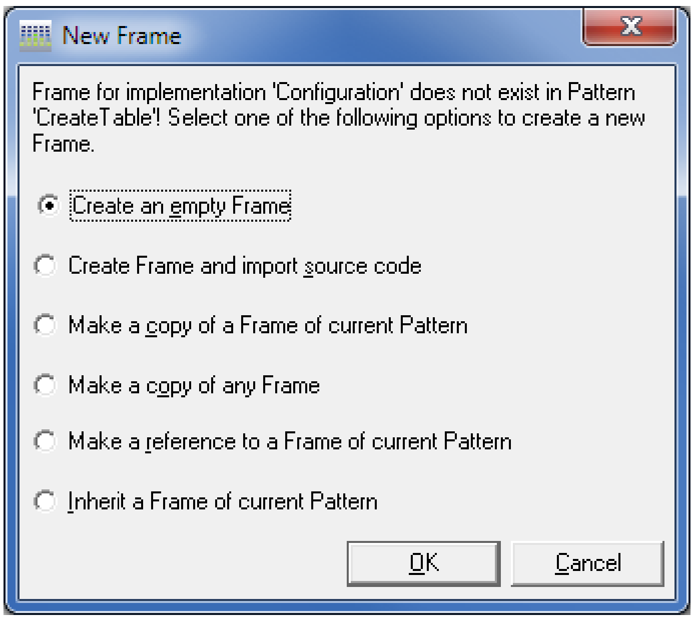
For simplicity reasons we create a configuration frame for each meta class.

Creating a Configuration Frame

Switch to the 'Pattern' tab and open the pattern *CreateTable*.
To create a configuration frame, click on the 'New Frame…' button in the top right corner of the Pattern view. Select 'Configuration (configuration)' in the 'Select Implementation Type' dialog.



Next, you can select the contents of the frame.



Frame for implementation 'Configuration' does not exist in Pattern 'CreateTable'! Select one of the following options to create a new Frame.

[✓ Create an empty Frame]

Create Frame and import source code

Make a copy of a Frame of current Pattern

Make a copy of any Frame

Make a reference to a Frame of current Pattern

Inherit a Frame of current Pattern

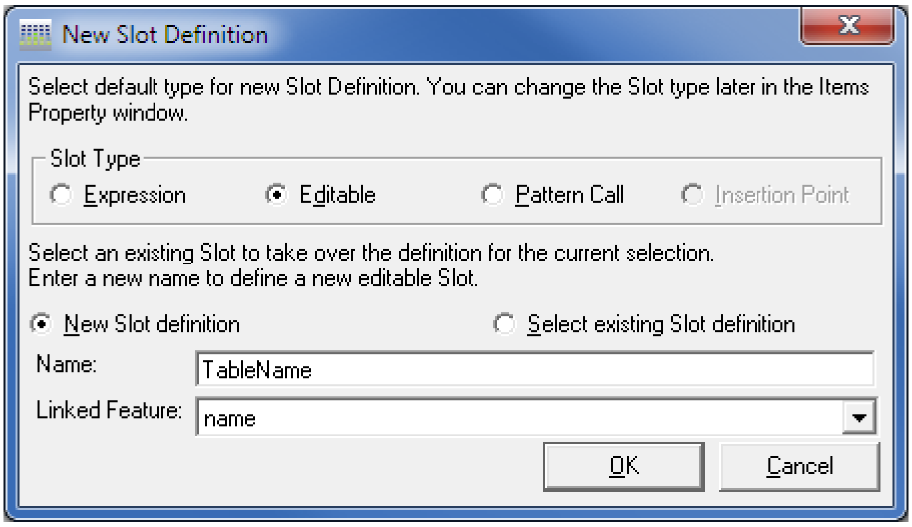
The selection “Create an empty Frame” is fine.

Just like in production patterns, slots are created in configuration patterns to describe variable parts. The creation of slots is executed by the context menu. *Expression*, *Editable* and *Pattern Call* are available as slot types.

- *Expression*: Calculates the value.
- *Editable*: The slot is linked to a feature of the meta model. Depending on the feature type, the user of the resulting form DSL can enter either a text (*String*), a number (*Numeric*) or select an entry from a list (*Boolean*, *Enumeration*).
- *Pattern Call*: Another configuration pattern is called.

Creating Editable Slots

In the first step, create a field to enter the table name. To do so we select *Editable* as slot type here. The slot has a name, *TableName* in this case, and is linked to a feature of the meta model class *Table*. Only The *Linked Feature* 'name' is available in this example.



New Slot Definition

Select default type for new Slot Definition. You can change the Slot type later in the Items Property window.

Slot Type: Expression [✓ Editable] Pattern Call Insertion Point

Select an existing Slot to take over the definition for the current selection.
Enter a new name to define a new editable Slot.

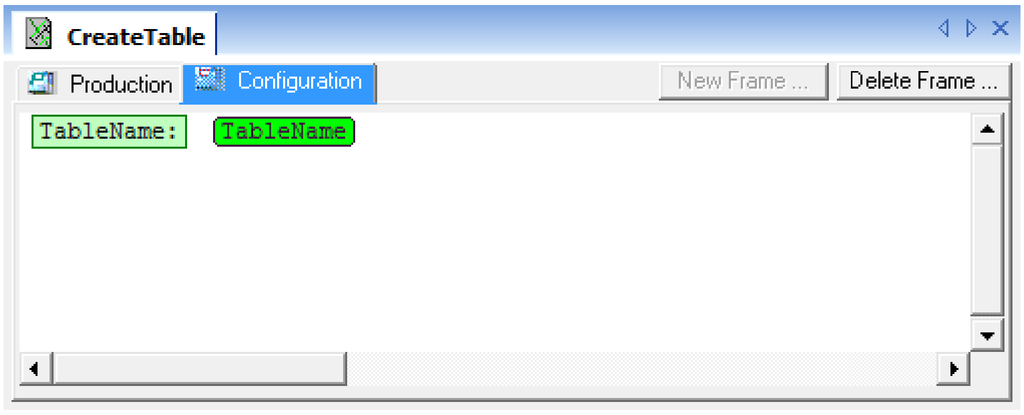
[✓ New Slot definition] Select existing Slot definition

Name: TableName

Linked Feature: name

Alternatively, you can drag the feature directly into the pattern via Drag&Drop (*'Pattern Properties'* → *'Interface'* tab). It is correctly created as *Editable* slot and bound to the feature.

In addition to the slots, explanatory text can be inserted. It is displayed 1:1 in the resulting form. To ensure that the user knows that he has to enter a table name, `TableName` is inserted before the slot in this example.



CreateTable

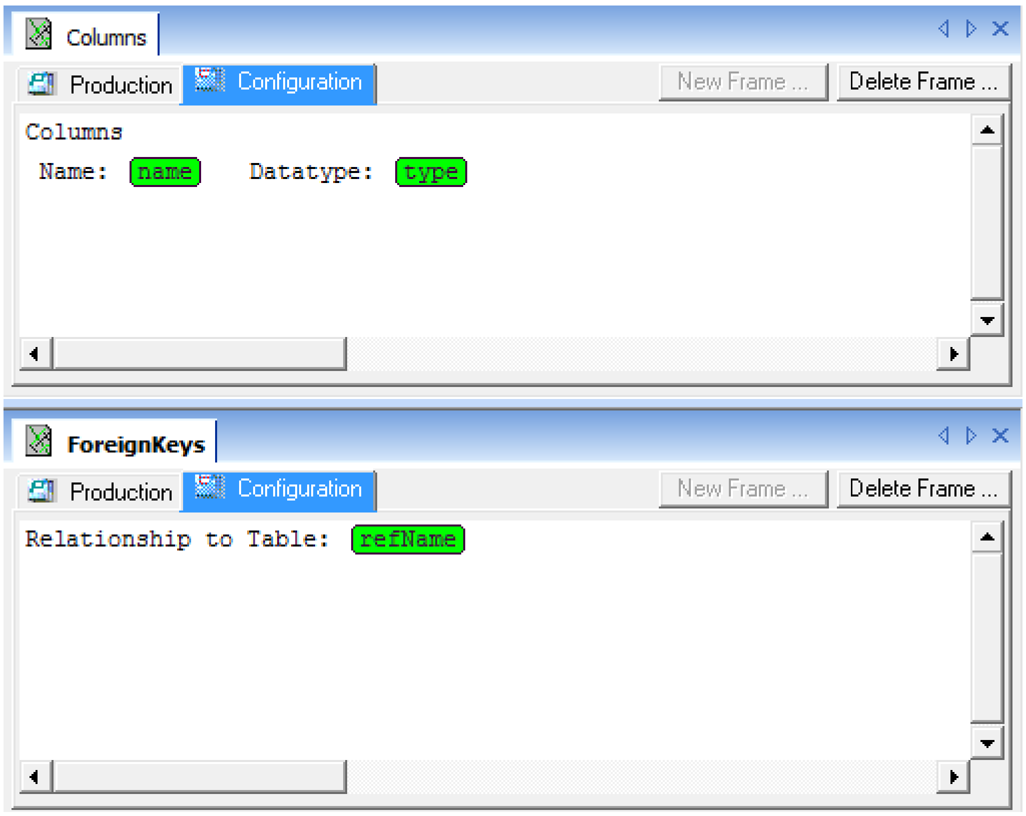
TableName: TableName

For the optical layout, you can place a marker block around the explanatory text.

As the other two *Linked Features* of the *Table* class are 'Composites', they cannot be created as *Editable*, only as *Pattern Call*. But before we can do that we have to create the according pattern that is bound to *Column*.

Creating Additional Configuration Patterns

Afterwards, we first create a configuration frame in the *Columns* and *ForeignKey* pattern. In the *Columns* pattern, `name` and `datatype` must be entered, in the *ForeignKey* pattern the reference to a table must be entered:

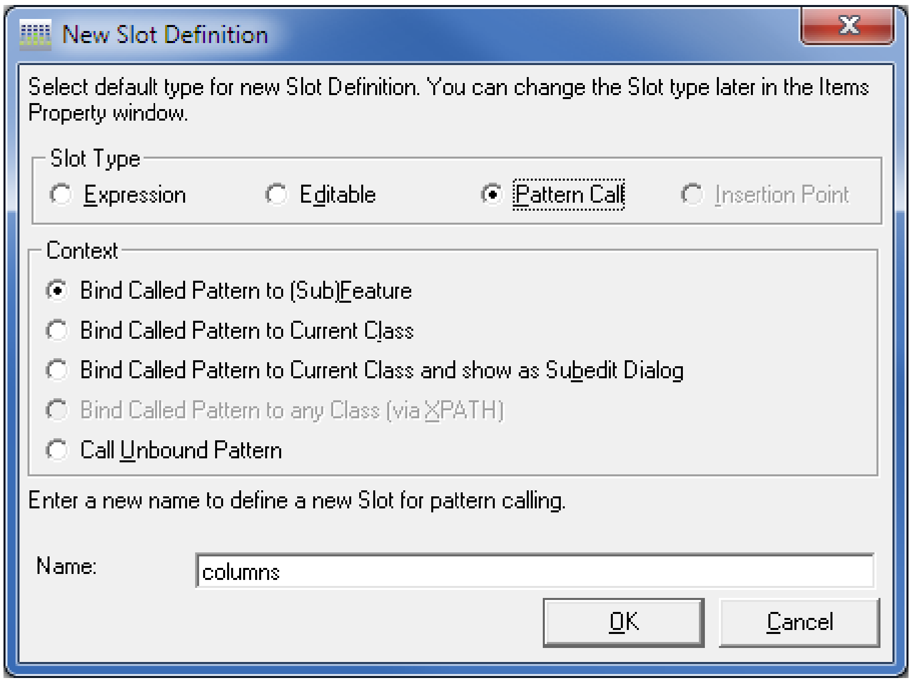


```
Columns
 Name: name    Datatype: type
```

```
Relationship to Table: refName
```

Then we save all patterns.

We switch back to *CreateTable* where we now can create the still missing calls of the newly created patterns. Create each in a separate new line by using the context menu 'Mark Slot...', then 'Pattern Call' with the context 'Bind called Pattern to (Sub)Feature' and enter the name 'columns' and the other time the name 'foreignKeys'.


New Slot Definition

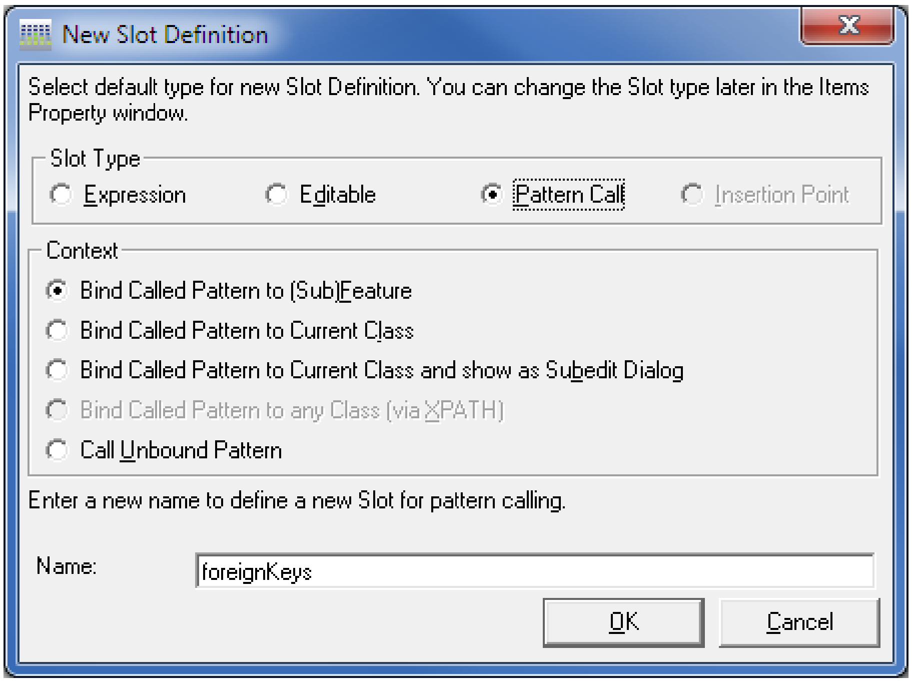
Select default type for new Slot Definition. You can change the Slot type later in the Items Property window.

Slot Type: Expression [✓ Pattern Call] Editable Insertion Point

Context:
[✓ Bind Called Pattern to (Sub)Feature]
Bind Called Pattern to Current Class
Bind Called Pattern to Current Class and show as Subedit Dialog
Bind Called Pattern to any Class (via XPATH)
Call Unbound Pattern

Enter a new name to define a new Slot for pattern calling.

Name: columns


New Slot Definition

Select default type for new Slot Definition. You can change the Slot type later in the Items Property window.

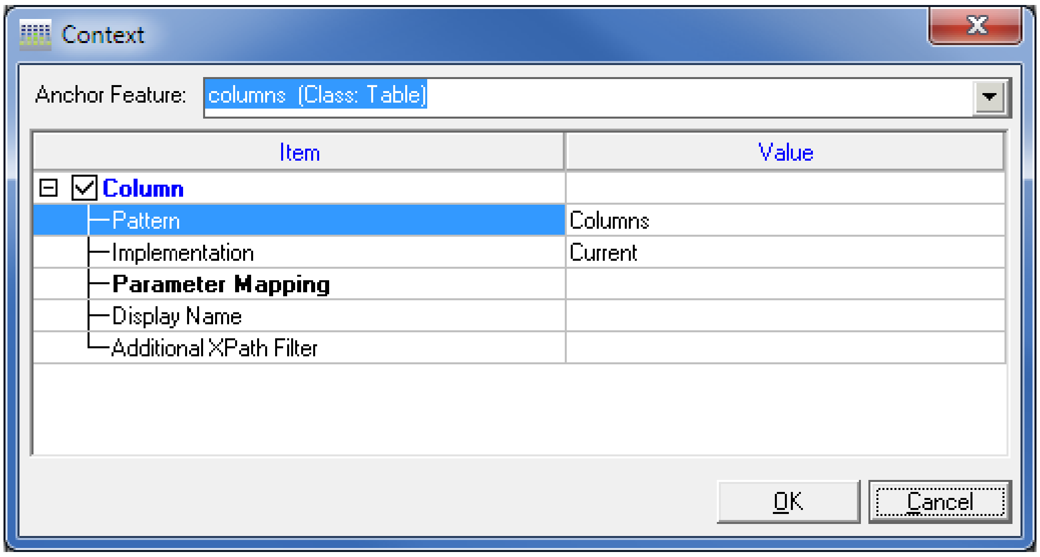
Slot Type: Expression [✓ Pattern Call] Editable Insertion Point

Context:
[✓ Bind Called Pattern to (Sub)Feature]
Bind Called Pattern to Current Class
Bind Called Pattern to Current Class and show as Subedit Dialog
Bind Called Pattern to any Class (via XPATH)
Call Unbound Pattern

Enter a new name to define a new Slot for pattern calling.

Name: foreignKeys

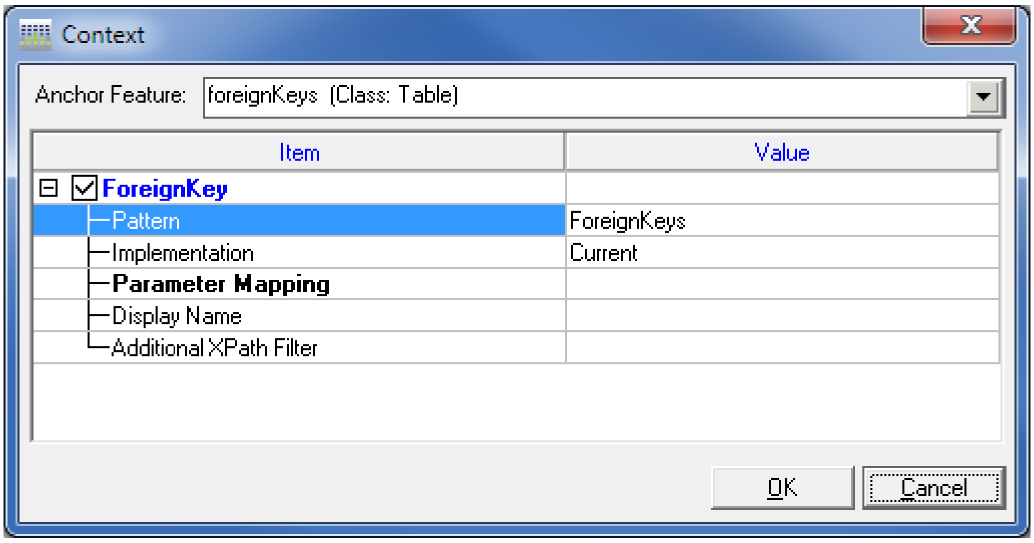
In the 'Context' dialog, we accordingly select the respective *Anchor Feature* and pattern.



Context

Anchor Feature: columns (Class: Table)

Item	Value
☑ **Column**	
Pattern	Columns
Implementation	Current
Parameter Mapping	
Display Name	
Additional XPath Filter	

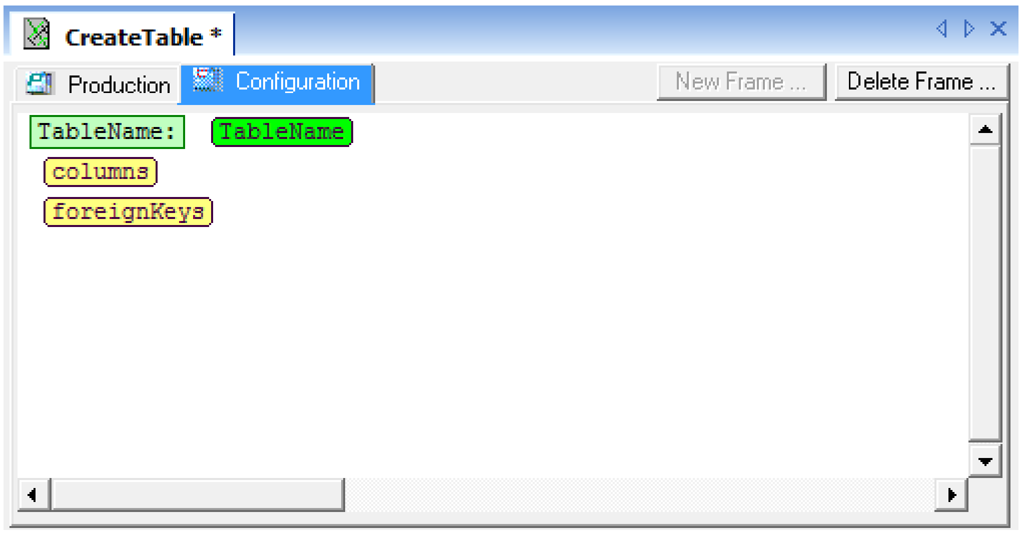


Context

Anchor Feature: foreignKeys (Class: Table)

Item	Value
☑ **ForeignKey**	
Pattern	ForeignKeys
Implementation	Current
Parameter Mapping	
Display Name	
Additional XPath Filter	

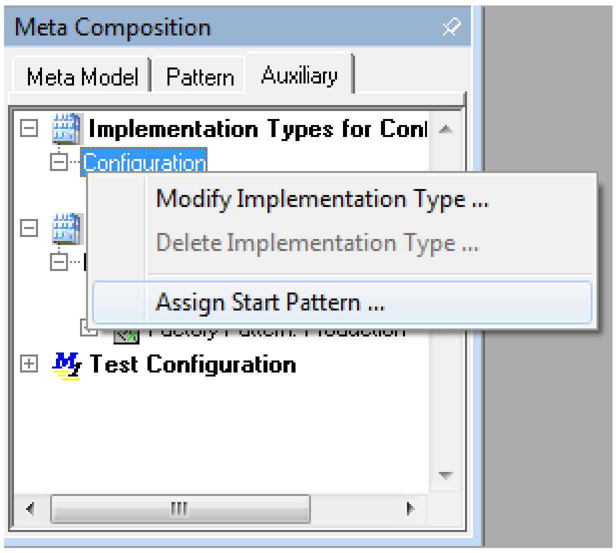
Result: CreateTable Pattern



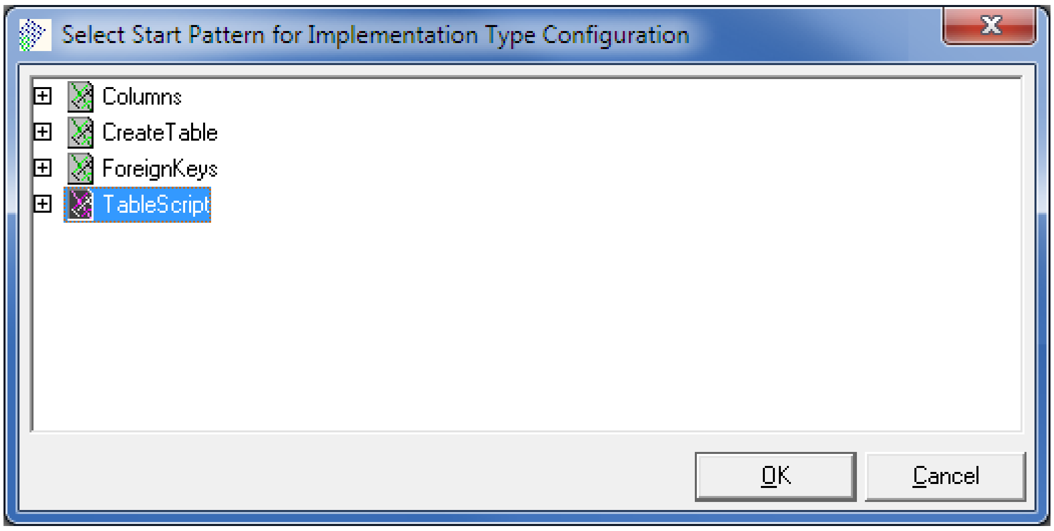
```
TableName:  TableName
 columns
 foreignKeys
```

Defining the Start Pattern

To generate the form DSL, one of the defined configuration patterns has to be defined as start pattern. In our example, we choose the configuration pattern 'TableScript' that we already prepared for you. A start pattern can be defined by switching to the 'Auxiliary' tab and opening 'Implementation Types for Configuration'. Open the context menu right-clicking on Configuration and select 'Assign Start Pattern...'.



The 'Select Start Pattern for Implementation Type Configuration' dialog opens. Select the start pattern *TableScript*.

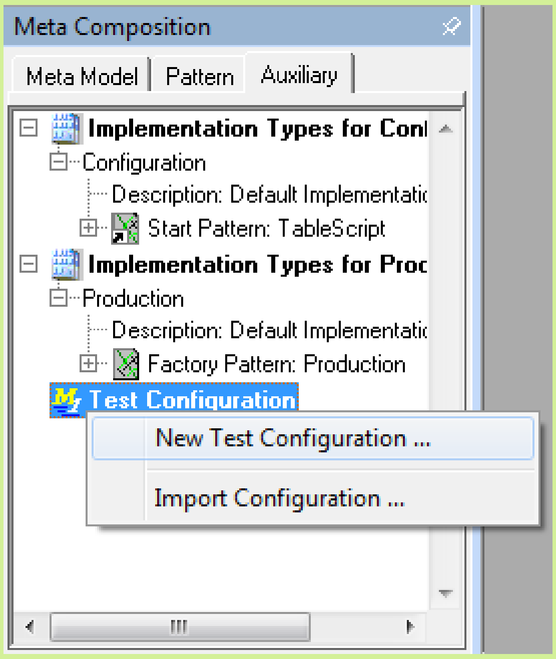


The result is also contained in the *MetaComposition* 'Tutorial-1_4.META'.

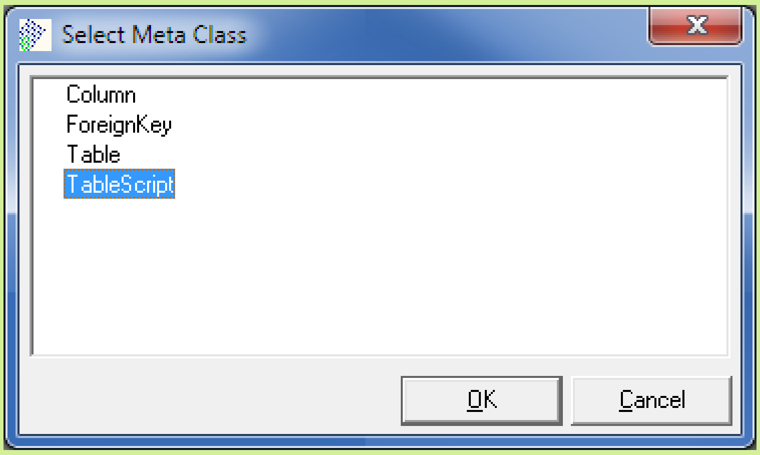
Creating a Test Configuration

Now all configuration patterns have been created and the start pattern has been defined, so a configuration can be created.

Select *Auxiliary* → *Test Configuration*. The context menu opens by a right-click. Select 'New Test Configuration…'.


Meta Composition
Meta Model
Pattern
Auxiliary
Implementation Types for Conf
Configuration
Description: Default Implementatic
Start Pattern: TableScript
Implementation Types for Proc
Production
Description: Default Implementatic
Factory Pattern: Production
Test Configuration
New Test Configuration ...
Import Configuration ...

Select the start class of the meta model (*TableScript*) in the "Select Meta Class" dialog.

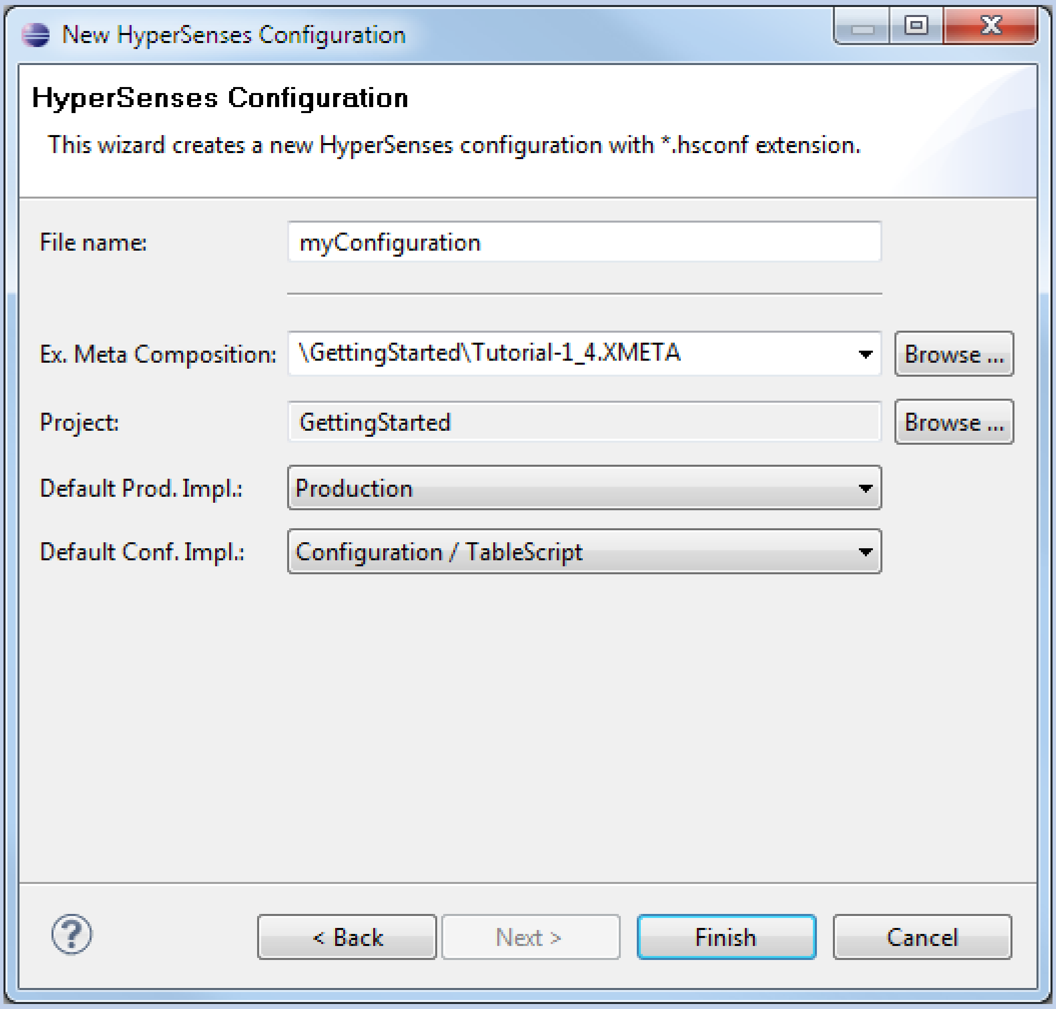


Afterwards, save the configuration. After saving a new dialog appears that asks if you want to open the configuration. Select 'Yes'.

Switch to Eclipse.

To create a new configuration within your Exclipse workspace, call the menu item ***File/New/Other ...*** (Ctrl+N) and select the type *Configuration* in the group *HyperSenses*.

The following dialog appears where you can set the properties of the new configuration.


New HyperSenses Configuration

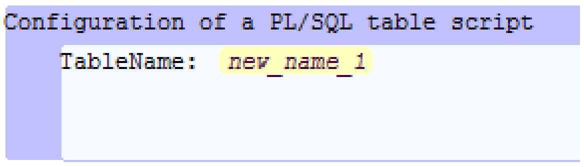
HyperSenses Configuration

This wizard creates a new HyperSenses configuration with *.hsconf extension.

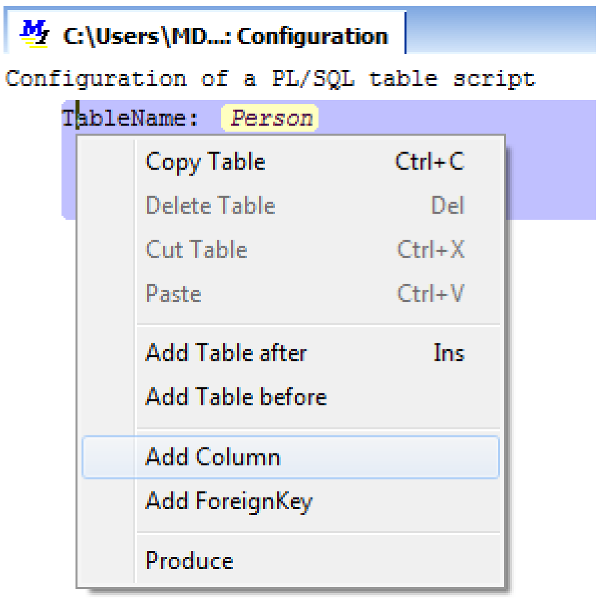
File name:	myConfiguration
Ex. Meta Composition:	\GettingStarted\Tutorial-1_4.XMETA
Project:	GettingStarted
Default Prod. Impl.:	Production
Default Conf. Impl.:	Configuration / TableScript

By selecting *Finish*, the configuration editor opens.

The following configuration looks like a form.

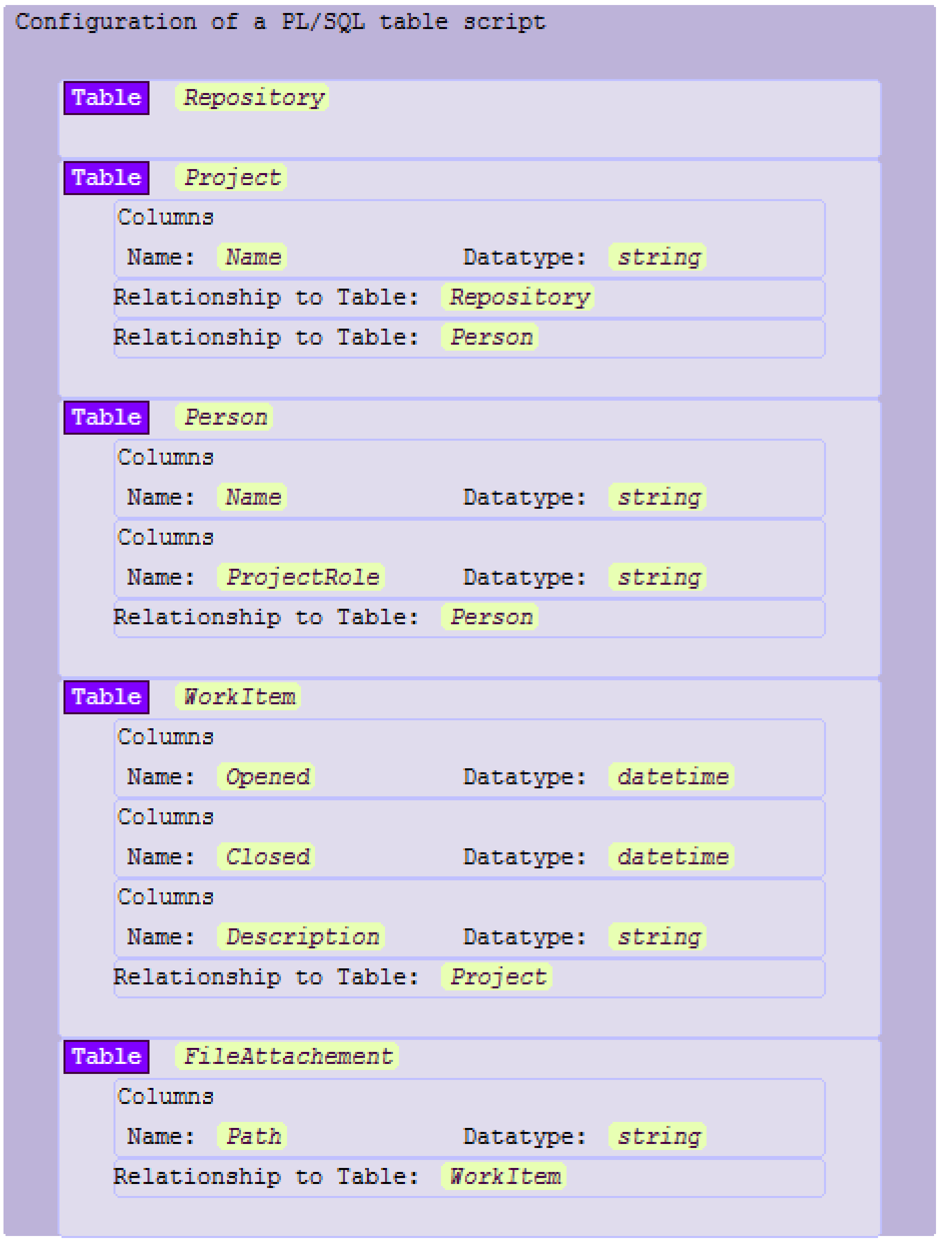

```
Configuration of a PL/SQL table script
    TableName:  new_name_1
```

In the displayed form, we configure the tables with their columns and *ForeignKey* relations. All classes and features can be created and edited via context menu.


```
Configuration of a PL/SQL table script
    TableName:  Person
```

As the configuration patterns are linked to the meta model, only configurations which fit to the application domain can be created. Please notice that the conditions of optional blocks and expressions of slots within the production patterns are case-sensitive.

Please configure the following tables:



```
Configuration of a PL/SQL table script

Table  Repository

Table  Project
    Columns
     Name:  Name                  Datatype:  string
    Relationship to Table:  Repository
    Relationship to Table:  Person

Table  Person
    Columns
     Name:  Name                  Datatype:  string
    Columns
     Name:  ProjectRole           Datatype:  string
    Relationship to Table:  Person

Table  WorkItem
    Columns
     Name:  Opened                Datatype:  datetime
    Columns
     Name:  Closed                Datatype:  datetime
    Columns
     Name:  Description           Datatype:  string
    Relationship to Table:  Project

Table  FileAttachement
    Columns
     Name:  Path                  Datatype:  string
    Relationship to Table:  WorkItem
```

If you have finished the editing of the form, you can call the 'Produce…' command in the context menu. In the following dialog “Select Implementation Type for Production”, we decide for which implementation we want to produce.

Now the patterns are called as they have been configured and filled with the values from the configuration. Afterwards, the individual files (that are defined by the Factory Patterns) are created and exported.

If you have any questions or suggestions, please use our forum where you can also request new Tokens: http://www.d-s-t-g.com/forum