



Version 12.2

Modeller Training

«Firma»

«Ort»

«Zeitraum»

camos Software und Beratung GmbH
Friedrichstraße 14
70174 Stuttgart

www.camos.de

© Copyright 2006 - 2014 by camos
Software und Beratung GmbH
Friedrichstraße 14
D-70174 Stuttgart

Telefon +49 711 780 66 - 0
Telefax +49 711 780 66 - 60

<http://www.camos.de>

All rights reserved. The transmission and the copying of these documents, as well as the usage and notification of the contents are not permitted, neither completely or partially, if it is not expressively and in written form confirmed by camos. Violations are obliged to compensations.

Contents

1.	Introduction.....	15
2.	Prerequisites to create an application	17
2.1.	Intention.....	17
2.2.	What is an application?.....	17
2.3.	Create the first application	17
2.3.2.	Practice: Create a knowledge base.....	20
2.4.	Orientation in the development environment	21
2.4.1.	camos Develop online help	21
2.4.2.	Abbreviations	22
2.4.3.	Wasele List: groups and view.....	23
2.5.	Projects and ticket integration.....	23
2.6.	Repetition.....	26
3.	Example application “Hello camos Develop”	27
3.1.	Intention.....	27
3.2.	Create an application	27
3.2.1.	Create start class.....	27
3.2.2.	Create a form with a label.....	29
3.2.3.	Constructor of the start class	33
3.2.4.	Starting the application.....	35
3.3.	Repetition.....	36
4.	Object-oriented programming	37
4.1.	Definitions and basics.....	37
4.1.1.	Objects and classes	37
4.1.2.	Inheritance	38
4.1.3.	Abstraction	38
4.1.4.	Modularity.....	39
4.1.5.	Constructors and destructors.....	40
4.2.	Object Orientation in camos Develop	41
4.2.1.	General.....	41
4.2.2.	Types of classes	41
4.2.3.	Inheritance and overload	43
4.2.4.	Components – Aggregation	44
4.2.5.	Components – Mutual use	46

4.3.	Common traps of object-oriented programming	47
4.3.1.	Mixing inheritance and association	47
4.3.2.	Wrongly used inheritance	48
4.3.3.	Correct abstraction grade	48
5.	Structure of the product	51
5.1.	Intention	51
5.2.	Set up the class structure	51
5.2.1.	Structure considerations	51
5.2.2.	Practice: Create classes	52
5.3.	Create components	53
5.3.1.	What is a component?	53
5.3.2.	Practice: Create component of class Car	54
5.4.	Selecting a model on the form	56
5.4.1.	Use of configuration boxes	56
5.4.2.	Practice: Create a configurationbox for Car	56
5.5.	Tips & Tricks	59
5.5.1.	Creating classes	59
5.5.2.	The NOVALUE symbol	59
5.5.3.	Representation of configboxes	60
5.6.	Repetition	61
6.	Defining components of the car	63
6.1.	Target	63
6.2.	Set up of the product structure	63
6.2.1.	Exercise: Show components as classes	63
6.2.2.	Exercise: Allocate components and init values	65
6.3.	Displaying components on form	67
6.3.1.	Exercise: Create configboxes for components	67
6.3.2.	Exercise: Form element Component tree	68
6.4.	Repetition	70
7.	Model selection via the product image	71
7.1.	Target	71
7.2.	Importing external graphic files	71
7.2.1.	Use of constants	71
7.2.2.	Graphic constant editor	72
7.3.	Display of graphics on a form	73

7.3.1.	Exercise: Form element Graphic	73
7.4.	Manual object instantiation.....	74
7.4.1.	Selection trigger of a form element	74
7.4.2.	Exercise: Create car object	75
7.5.	Tips & Tricks.....	75
7.5.1.	Display of graphics	75
7.5.2.	Changing several form elements simultaneously	76
7.5.3.	Undoing actions	76
7.5.4.	Tooltips	76
7.5.5.	Load graphic	77
7.6.	Repetition.....	78
8.	Basics of multilingualism.....	79
8.1.	Target.....	79
8.2.	Creating a new language in the frame	79
8.2.1.	Main- and secondary languages.....	79
8.2.2.	Exercise: Create new language.....	79
8.3.	Exercise: Migrating the language.....	83
8.4.	What is a naming and what is a name?	85
8.5.	Maintaining namings	86
8.5.1.	Object naming / Namings on classes	86
8.5.2.	NOVALUE naming on Components / Features	87
8.5.3.	Namings on components and features	88
8.6.	Tips & Tricks.....	89
8.6.1.	Definition of multilingual texts	89
8.6.2.	Display options in the form element Component tree.....	91
8.6.3.	Presetting of the NOVALUE naming	93
8.6.4.	Display of class- and component namings	94
8.6.5.	System texts for system dialogs and form elements.....	95
8.7.	Repetition.....	96
9.	Switching the language via the menu bar.....	97
9.1.	Intention.....	97
9.2.	Practice: Create a menu.....	97
9.2.1.	Integrate graphics as menu icons.....	99
9.2.2.	Assign actions to the menu items	100
9.2.3.	Allocate the menu to the form	100
9.3.	Tips & Tricks.....	101

9.3.1.	How to activate and deactivate menu items.....	101
9.3.2.	How to indicate a menu item as being selected.....	102
9.3.3.	Icon for a deactivated menu item.....	103
9.4.	Repetition.....	103
10.	Pricing.....	105
10.1.	Intention.....	105
10.2.	Main- and secondary currencies.....	105
10.2.1.	Practice: Create the currency US Dollar in the frame	105
10.3.	Features	106
10.3.1.	Practice: Create the feature Price.....	107
10.3.2.	The feature editor	108
10.3.3.	Practice: Set the init value	109
10.3.4.	Practice: Overloading the feature Price.....	110
10.3.5.	Practice: Configurationbox columns to display the Price	111
10.4.	Option “Reinitialize when object changes”.....	113
10.5.	Tips & Tricks	115
10.5.1.	Columns of a configuration box.....	115
10.5.2.	Change the column width in the form preview	116
10.5.3.	Lines for the tree representation of a configbox.....	117
10.6.	Repetition.....	117
11.	Price calculation	119
11.1.	Intention.....	119
11.2.	Practice: Defining list prices.....	119
11.3.	Preliminary considerations with regard to the price calculation.....	121
11.3.1.	When is the price calculation carried out?.....	121
11.3.2.	Saving the calculation result	121
11.3.3.	Access to properties of a superior object.....	122
11.4.	Practice: Implementing the price calculation.....	123
11.5.	Error Sources.....	124
11.6.	Repetition.....	125
12.	Integrate further modules	127
12.1.	Intention.....	127
12.2.	Practice: Extending the class structure	127
12.3.	Practice: Create components.....	128
12.4.	List components.....	129

12.4.1. Practice: Create a list component.....	130
12.5. Practice: Add the modules to the MainForm	131
12.5.1. Representation of list components.....	131
12.6. Repetition.....	133
13. Options of the component tree	135
13.1. Intention.....	135
13.2. Component tree columns.....	135
13.3. How to open the component tree automatically.....	137
13.4. Change the sorting of the components	138
13.5. Tips & Tricks.....	141
13.5.1. Find properties.....	141
13.5.2. Overloaded component sorting.....	143
13.6. Repetition.....	144
14. Color of the paintwork	145
14.1. Intention.....	145
14.2. Decision: feature or class?.....	145
14.3. Practice: Create the feature Color	146
14.4. Practice: Create and assign feature values.....	147
14.5. Configurationbox feature.....	149
14.5.1. Practice: Extend the form by a configurationbox feature.....	149
14.5.2. Notes for the form element configbox feature.....	151
14.6. Tips & Tricks.....	152
14.6.1. Representation options for the configurationbox feature	152
14.7. Repetition.....	153
15. Power of the engine	155
15.1. Intention.....	155
15.2. Practice: Create the feature Power.....	155
15.3. Practice: Define units of quantity	155
15.3.2. Practice: Overload the init value of the feature Power	158
15.3.3. Practice: Extend the form by the Power	158
15.4. Tips & Tricks.....	160
15.4.1. How to assign units of quantity dynamically	160
15.5. Repetition.....	160
16. Settings and features in the development system.....	161

16.1.	Intention.....	161
16.2.	Group-oriented structure.....	161
16.3.	User options.....	163
16.4.	Knowledge base options.....	164
16.5.	Repetition.....	165
17.	Introduction to the rule work.....	167
17.1.	Intention.....	167
17.2.	Before creating a rule.....	167
17.2.1.	Formulating rules.....	167
17.2.2.	License for the rule work.....	167
17.2.3.	The property “Rules enabled”	168
17.2.4.	The property “Check relevant”	168
17.3.	Rule types.....	169
17.4.	Creating a rule	169
17.4.1.	The rule editor.....	169
17.4.2.	Always allowed?.....	172
17.4.3.	Conditions.....	173
17.4.4.	Practice: Creating rules.....	174
17.4.5.	Condition element Multibox.....	178
17.5.	Interpretation of rules.....	180
17.5.1.	Validity symbols.....	180
17.5.2.	Rule explanations.....	182
17.5.3.	The inference machine.....	184
17.6.	Tips & Tricks	184
17.6.1.	Handling in the rule editor	184
17.6.2.	How to administrate rule names and explanations in the frame.....	185
17.6.3.	System texts for Default rule explanations.....	187
17.6.4.	User defined rule types	188
17.6.5.	How to hide forbidden values.....	190
17.6.6.	Use of rule groups.....	191
17.6.7.	Deactivating the rule check.....	192
17.6.8.	Restricting the rule check to subtrees.....	192
17.7.	Repetition.....	193
18.	Constraints.....	195
18.1.	Intention.....	195
18.2.	Use of constraints.....	195

18.2.1.	Definition of constraints	195
18.2.2.	The constraint editor.....	195
18.3.	Practice: Constraint for accessories	196
18.4.	Tips & Tricks.....	198
18.4.1.	Display of constraint rules in the structure tree	198
18.4.2.	Fill constraints with data from an Excel file.....	198
18.4.3.	Constraints with database links	198
18.5.	Repetition.....	199
19.	Discount calculation.....	201
19.1.	Intention.....	201
19.2.	Practice: Create the feature Discount.....	201
19.2.1.	Determine a format	202
19.3.	Practice: Defining the unit of quantity Percent.....	204
19.4.	Practice: Implement the discount calculation	205
19.4.1.	Practice: Determine the value range	207
19.4.2.	Practice: End price calculation only with valid values.....	208
19.5.	Tips & Tricks.....	210
19.5.1.	Display the validity of the discount	210
19.5.2.	Confirm an entry through Return.....	211
19.6.	Repetition.....	211
20.	Creating a quotation	213
20.1.	Intention.....	213
20.2.	Create a result	214
20.2.1.	Practice: Define a result in the frame	214
20.3.	Practice: Create a test result.....	216
20.4.	Fill, assign and display the quotation	218
20.4.1.	Practice: Fill the result with values.....	218
20.4.2.	Practice: Include the product picture of the model.....	225
20.5.	Tips & Tricks.....	229
20.5.1.	Display the naming instead of the name	229
20.5.2.	Optimize result window.....	230
20.5.3.	How to hide individual components.....	231
20.5.4.	Change the order of the text modules in the result	233
20.5.5.	Adjust the result to consider the dialog language	233
20.5.6.	Display the paintwork color and engine power	234

20.6. Convert RTF-Text to a Textelement.....	235
20.7. Repetition.....	238
21. Multi-user access.....	239
21.1. Intention	239
21.2. Basics of the access maintenance	239
21.2.1. Access to properties of the knowledge base	240
21.2.2. Access to the frame properties	240
21.2.3. Access to classes	240
21.3. Tips & Tricks	241
21.3.1. Search for reserved classes.....	241
21.3.2. Reserving via hotkey	242
21.3.3. Versioning of classes	242
21.3.4. Optimize the performance of the development environment	242
21.4. Repetition.....	244
22. Versioning.....	245
22.1. Intention	245
22.2. What are work- and release versions?	245
22.3. Practice: Creating a new release version	246
22.4. Practice: Creating a new working version.....	248
22.5. Repetition.....	250
23. Model-specific components.....	251
23.1. Intention	251
23.2. Practice: Create model-specific components.....	251
23.3. Tips & Tricks	255
23.4. Repetition.....	255
24. Dynamic form structure.....	257
24.1. Introduction	257
24.2. What is a subform?	257
24.3. Practice: Using subforms.....	257
24.4. Tips & Tricks	260
24.5. Repetition.....	261
25. Discount calculation via 2D-table	263
25.1. Intention.....	263
25.2. The wasele 2D-table	263

25.3.	Practice: 2D-table for cash discount calculation	263
25.4.	Tips & Tricks.....	266
25.5.	Repetition.....	268
26.	Integration of car keys	269
26.1.	Intention.....	269
26.2.	Practice: Extend the class structure by car keys.....	269
26.3.	What are quantities?	271
26.3.1.	Why no feature “Quantity” or no list of keys?	271
26.4.	Practice: Configure the number of keys	271
26.5.	Practice: Define and assign the unit of quantity	272
26.6.	Practice: Create quantity rules.....	274
26.7.	The system method ChangeQuantity	276
26.7.1.	Practice: Adapt the price calculation for the keys	276
26.8.	Tips & Tricks.....	277
26.8.1.	Permanent display of spinbuttons	277
26.8.2.	Display the number of keys and their price in the quotation	277
26.8.3.	How to forbid the quantity 0?	278
26.9.	Repetition.....	279
27.	Assign components rule-controlled.....	281
27.1.	Intention.....	281
27.2.	Definition of decision tables.....	281
27.3.	The decision table editor	281
27.3.1.	Condition part.....	282
27.3.2.	Action part	282
27.3.3.	Rule type.....	282
27.4.	Practice: Special models for the Golf	282
27.5.	Tips & Tricks.....	286
27.5.1.	Process order.....	286
27.5.2.	Invalid expressions	286
27.5.3.	Use of operators and methods.....	287
27.5.4.	Toolbar	287
27.5.5.	Keyboard shortcuts.....	288
27.6.	Repetition.....	288
28.	Users and security	289
28.1.	Intention.....	289

28.2.	User maintenance.....	289
28.3.	Securities on the knowledge base.....	291
28.4.	Securities on the frame	292
28.5.	Securities on classes.....	293
28.5.1.	Default securities	293
28.5.2.	Collective changing of the securities on classes	294
28.6.	Tips & Tricks	295
28.6.1.	Login with different username	295
28.6.2.	First start of the user maintenance.....	296
28.7.	Repetition.....	297
29.	Import and export knowledge bases	299
29.1.	Intention.....	299
29.2.	Knowledge base export.....	299
29.3.	Practice: Export the car configurator.....	300
29.4.	Knowledge base import	301
29.5.	Practice: Import frame and knowledge base	303
29.6.	Tips & Tricks	304
29.6.1.	Import into an existing knowledge base.....	304
29.6.2.	Package-Export and –Import	305
29.6.3.	Project-Export	307
29.7.	Repetition.....	308
30.	From the knowledge base to the application	309
30.1.	Intention.....	309
30.2.	What is a KIF?.....	309
30.3.	Practice: Create KIF	309
30.4.	Practice: Start KIF.....	312
30.5.	Tips & Tricks	314
30.5.1.	Further start parameters.....	314
30.5.2.	How to define your own parameters.....	314
30.5.3.	Error message with the start – possible causes	315
30.6.	Repetition.....	316
31.	Administration of texts in the Textmanager	317
31.1.	Textelements	318
31.1.1.	Practice: Creating Textelements	319
31.1.2.	Textelements with included cause variables	320

31.1.3. Textelements as possible values.....	322
31.2. How to integrate Textelements into the KIF	322
31.3. How to ex- and import Textelements	323
31.4. Namings for textelements.....	324
31.5. Tips & Tricks.....	325
31.5.1. Save a filter	325
31.5.2. How to replace multilanguage texts with Textelements	325
31.5.3. How to set the mode for new string constants	326
31.6. Repetition.....	326
32. Common requirements for an application	327
32.1. Intention.....	327
32.2. Navigation in the class tree.....	327
32.3. Dealing with the Recycle Bin	327
32.4. Improving the user interface via tab pages.....	329
32.4.1. Practice: Division into tab pages.....	329
32.5. Alignment of form elements / form alignment	333
32.5.1. Tips & Tricks.....	336
32.6. Display of the result on the form	336
32.7. Extend the result by address and creation date	338
32.7.1. Fade out objects from the component tree	341
32.8. Transparent graphics.....	342
32.9. Individual title bar.....	343
32.10. Basic properties of form elements/abstract representation.....	345
33. Hotkeys and Shortcuts	349
33.1. Hotkeys in the Class Tree.....	349
33.2. Hotkeys in wasele lists, groups and classes	349
33.3. Hotkeys in the procedure editor	349
33.4. Hotkeys in the decision table	350
33.5. Hotkeys in the debugger	351
33.6. Hotkeys in the RTF Editor	351
33.7. Navigation in the class tree, tree and component tree.....	351
33.8. Navigation in the rule editor.....	351
33.9. Navigation in the form preview.....	352
33.10. Other hotkeys.....	352
33.11. Identify form elements	352

33.12. Copy text from a WinMessage.....	352
34. Index of Figures.....	353
35. Topic Overview	357

1. Introduction

Welcome to the training documents for camos Develop 12.2!

These documents are issued as accompanying material in the camos Develop Modeler Training and are supposed to repeat and consolidate the obtained knowledge.

The aim is to get a deep insight in the structure and functionality of the development environment in order to be able to work independently with camos Develop. To this generally belongs the organization of a product structure and the definition and maintenance of complex rule-works.

In order to learn the programming with camos Develop it is not necessary to be already familiar with a programming language or the object-oriented programming (OOP). However, you should be familiar with the operating system Windows.

The intention of the training is the development of a car configurator. Contrary to the realistic course with the development of an application, the topics are discussed problem-related and the corresponding solution possibilities are introduced.

Note: The training documents do not explain the installation and the initiation of camos Develop, but they assume an installed and functioning local development environment. Information for the setup can be found in the *Installation instruction* on your camos Develop Training-CD.

For a better orientation the following formatting is agreed:

- Path specifications (menus, file) and labels (buttons, dialogs) as well as proper names are indicated italic, e.g. *File* -> *Class new ...*
- Program instructions are emphasized by a special font, e.g. RETURN VariableA;
- Terms and catchwords that are dealt with for the first time in a chapter are marked **bold**.

Structure of the individual chapters:

The training documents are structured in a way that first the intention or the problem is explained and corresponding solution possibilities are shown.

Most chapters have a subchapter “Tips & Tricks” which provides suggestions for extensions, simple work methods or the answer to frequently asked questions.

Each chapter is concluded with a short repetition in which the most important learn contents are listed in headwords.

In each chapter you will find various formatting and icons:



Work instructions that you carry out in order to comprehend the training example are highlighted in the formatting of this paragraph.



At diverse places you will find notes for your own later development work. These notes are formatted like this paragraph.



Paragraphs that are indicated by the flash icon should draw your attention to frequent error sources.

Definition

Basic term definitions are summarized in a separate box.



Facts that are described more detailed in the online help of camos Develop (F1 in the development system) are indicated by the blue and white help icon. Additionally the path to the corresponding chapter is specified, e.g. *Basics/Search dialog*.

2. Prerequisites to create an application

2.1. Intention

In this chapter the prerequisites and partial steps are discussed that are necessary to create a new application with camos Develop. Since you should become familiar with the surface and the navigation in the development system, you create a small example application following the well-known introductory program “Hello, World!”

2.2. What is an application?

The environment in which an application is created is called **knowledge base**. Only if the knowledge base is started, you talk about an application.

A knowledge base contains the complete knowledge that is required to manage the type of problem. If you want to e.g. program a pocket calculator, it has to be defined in the knowledge base how the pocket calculator looks like, which significance the individual numeric keys have and what happens if e.g. the button “+” is pressed.

Most of the customers of camos use camos Develop to set up product configurators, e.g. for cars, cranes and electro systems. The knowledge base of a product configurator contains among other things all possible variants of the product, e.g. different models, prices and discounts as well as rules which combinations of components and variants are possible or not. Therefore the knowledge base of a product configurator is also called *configuration logic*.

2.3. Create the first application

In order to be able to create a new knowledge base, first a so-called **frame** has to be created.

The frame of a knowledge base contains the definition of basic properties that are important for the application, such as e.g. which languages, currencies and fonts have to be available.

First, a little example should be programmed in which the text “Hello camos Develop!” is displayed on the screen.

To do so, first the following two steps are necessary:

1. define a frame
2. create a new knowledge base and assign the frame to it

2.3.1.1. Practice: Create Frame

Start camos Develop and open the *frame maintenance* via the main menu item *Maintainances -> Frame maintenance*.



Select the icon and allocate a name, e.g. "TrainingFrame" and the version 1.0. After confirming with *OK* the frame "TrainingFrame" is created in the working version 1.0.

Open the frame via double click on version 1.0 in the table in the middle or the context menu item *Open* or the corresponding icon from the toolbar above the table in the middle.

As mentioned at the beginning, the most different basic settings for an application are entered in the frame. These basic settings are divided in several areas, that can be selected via the navigation-area, that is located in the left bottom of the dialog:

Colors

Colors that are frequently used in the knowledge base, e.g. the colors of the company logo or fonts and background colours, can be defined in the frame. Then they are available on all elements of the surface (form elements) and don't have to be mixed again.

Stylesheets

Stylesheets offer the possibility to pre-define the look of form elements. By this you can define very comfortably a uniform design (colors, alignment, fonts) for the whole application.

If the properties of a stylesheet are changed afterwards, these modifications automatically affect all form elements using this stylesheet.

Units of quantity

Via units of quantity several measuring units such as centimeter, kilogram, etc. can be defined. These units can be used in the knowledge base to be able to work with values such as "diameter = 15 mm". Units of quantity are defined in Units of quantity groups with a conversion factor to the base unit of quantity. If the current measuring unit in the application is e.g. switched from millimeter to centimeter, an automatic conversion is carried out.

Defaults

In this area standard properties for several elements of the knowledge base can be set.

A new created element automatically receives the settings that are defined here. But these settings can be changed later in the development system if necessary. A modification on the default, that is carried out later, has no effect on already existing elements.

Fonts

In the area *Fonts* so-called **fontaliases** are defined. Depending on the current dialog language the fontalias determines which font type and font size should be used to display texts on the surface of the application.

The fontalias Standard is automatically created and cannot be deleted. It has to contain a font definition for all defined dialog languages.

Styles

Here the global Stylesheets for the RTF-Editor can be maintained, e.g the alignment and the distance from the Edges can be defined.

Languages

Here the dialog languages are entered which should be available for the application. If several languages exist, the current language can be switched via a camos Develop function.

The **main language** *United States* is automatically created if camos Develop is operated with the system language *English*.

Currencies

If prices and currencies are used in the application, at least one currency has to be defined – the so-called **main currency**.

Rule Types

In this area the validity terms and icons of the several system-rules can be edited. Furthermore user-defined rule types, called recommended rules, can be created.

Options

In the area *Options* textelements for texts in system dialogs and form elements are defined. Therefore these texts can be changed and / or translated into several languages. Furthermore e.g. design of WinMessage-Dialogues can be influenced.

To edit the settings of an area you have to select it in the navigation-area and then **reserve** it (double click in the first column or context menu). This is necessary because the frame can be opened by multiple users at the same time and for that reason an access-control for editing (see chapter 28.4) is necessary.

To be able to reserve an area in the frame an active ticket is necessary (see chapter 2.5).



Figure 1: Navigation and access administration in the frame

An area that is reserved by you is marked with a green icon. If an area is reserved by another camos Develop-developer, the area is an orange mark is displayed.

A frame can be simultaneously allocated to several knowledge bases; in this case all applications have the same settings. Changes in the frame have as well an effect on all of these knowledge bases.



To import an existing frame (file extension *.fmx), create a new frame and select the context menu icon Import from the column in the middle of the frame maintenance.



For the example "Hello, World" the frame must have only the language *United States* and a corresponding definition of the *Fontalias Standard*. Since these settings automatically exist, you can close the frame without changes.

2.3.2. Practice: Create a knowledge base

Now a frame exists and therefore a new knowledge base can be created and connected to the *TrainingFrame*.



Select the main menu item *Knowledge base -> New ...* and enter in the dialog *New Knowledge base* the name "TrainingExample". Via the icon below you assign the *TrainingFrame*. Click OK to create the knowledge base.

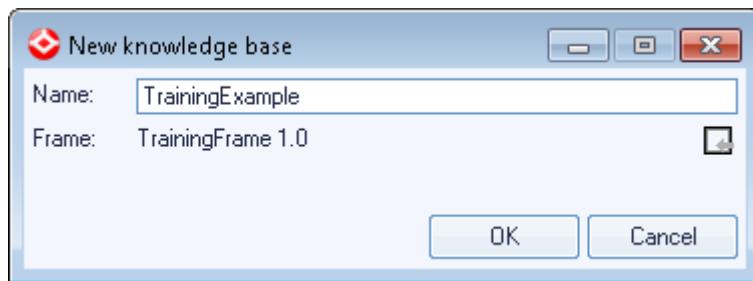


Figure 2: Creating a knowledge base

2.4. Orientation in the development environment

Figure 3 shows the normal view during the development.

The left third of the window is the **class tree** while the complete right part is described as **workbench**. The workbench consists of the **structure tree**, the **wasele lists** and the class editor or the **editor** of the currently opened wasele.

The most frequently used menu items are additionally accessible via icons in the toolbar of camos Develop and the individual editors of the surface. All icons have a tooltip that provides a brief description for the respective function.

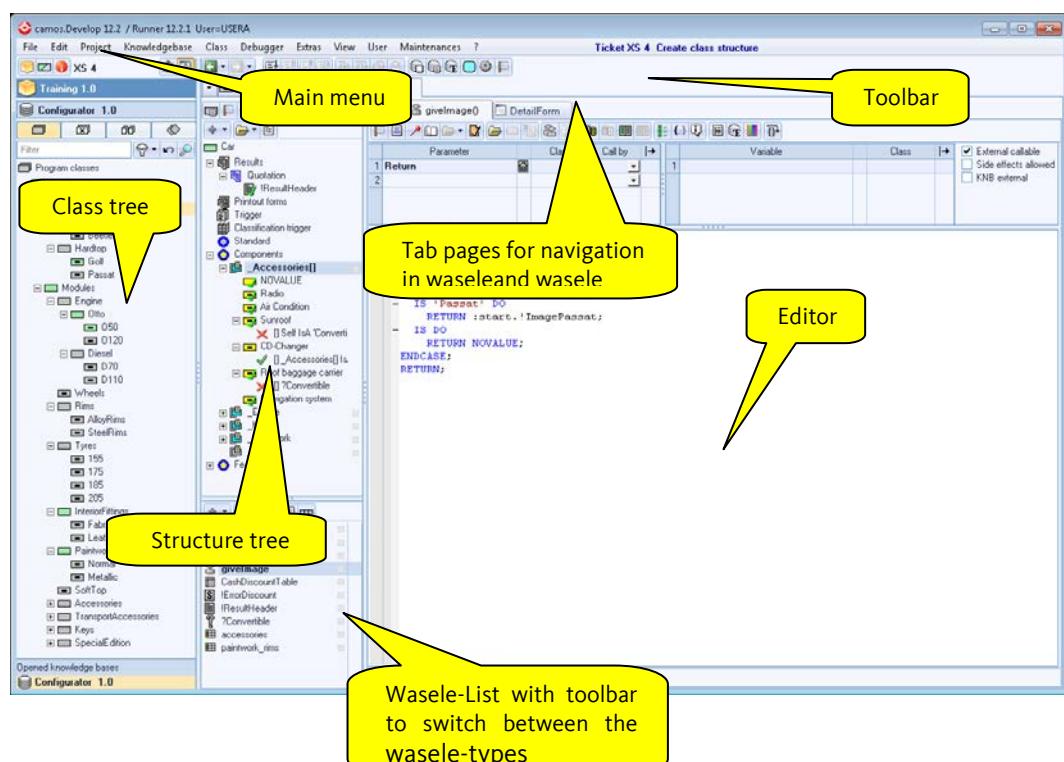


Figure 3: Surface of the camos Develop development environment

2.4.1. camos Develop online help

The structure of the workbenches was also considered in the structure of the camos Develop online help. The online help contains all wasele, dialogs and structures in camos Develop. If you are searching for information about the form element Pushbutton, you will find them in the chapter path *Workbench/wasele/Form/Form elements/Pushbutton*. Useful information and developer tips can also be found in the chapters *Basics* and *FAQ*.

The online help is started via the main menu item ? -> *Help for camos Develop* or via the hotkey F1. It is a context-related help system, i.e. if the focus is e.g. in the class tree and you press F1, the online help opens and the chapter *Class tree* is automatically selected.

Within the online help you can navigate through the complete structure on the tab page *Tree*. The tab page *Index* contains all chapters in alphabetical order. On the tab page *Search* you can additionally carry out a headings- and/or fulltext search.

In the toolbar you can see an icon. Via this icon you can send your point of view on the actual read chapter by e-Mail to camos. Here you can deposit own texts, in which you can describe, what has been very helpful for you or what information you haven't found.

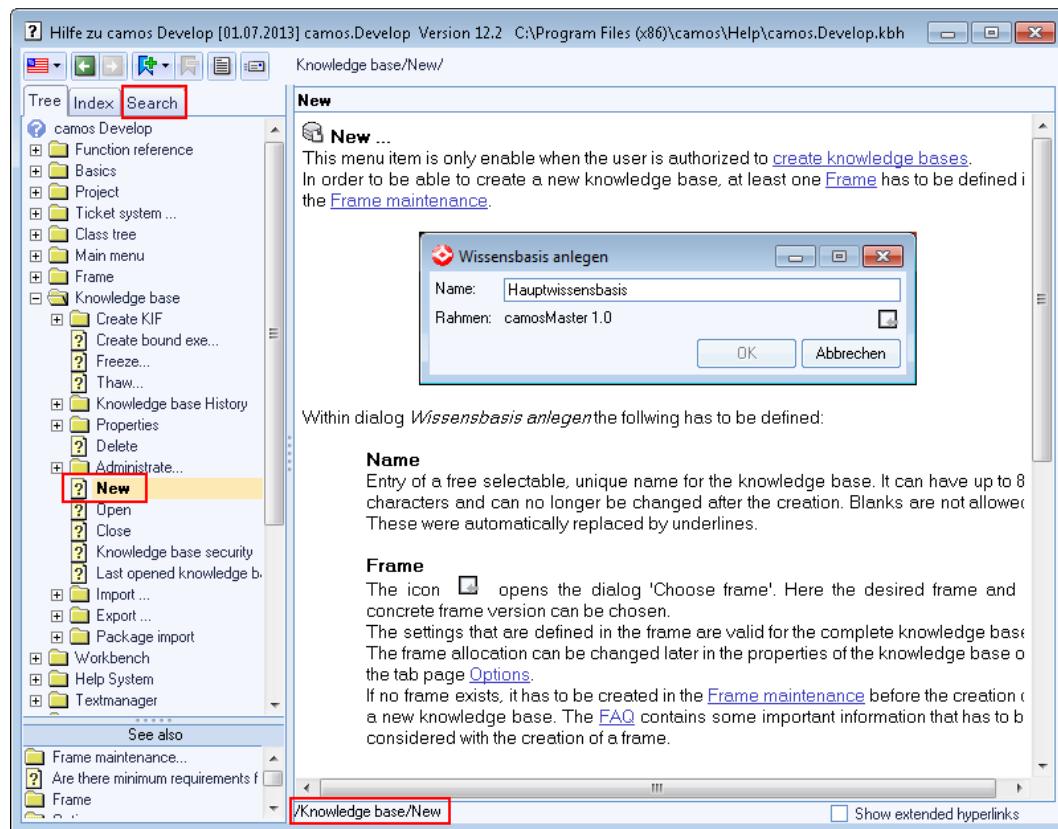


Figure 4: Dealing with the camos Develop online help

In the training documents you will find a lot of references to the online help, e.g.: [Knowledge base/New](#). In order to get further information, copy the path and press F1 and in the opened online help the shortcut Ctrl + D. In the dialog *Search Docitem* paste the copied path and click OK.

Of course you can manually navigate to the specified chapter, e.g. open chapter *Knowledge base* and click on page *New*.

Via the hotkey Ctrl + L you can start a Look-Up-Search (quick search) in the chapter tree.

2.4.2. Abbreviations

In addition to the subareas of the surface of camos Develop that are described in Figure 3, some more terms have to be explained for a better understanding:

- "... selecting an element" = simple click on the element
- "... opening an element" = double click on the element
- wasele stands for "Wissensbasiselement" (knowledge base element)
- EFG stands for "Einflussgröße" which means cause variable
- D&D means "Drag and Drop"
- *Help* means the online help (see 2.4.1). The *Userhelp* however is the integrated documentation tool with which you can write an own context-related help for your application.

Wasele stands for "Wissensbasiselement" (knowledge base element) and is a collective term for properties of a class, e.g. the windows (called *forms*), variables (called *features*) and procedures (called *methods*).

2.4.3. Wasele List: groups and view

Groups can be defined within the wasele list of a class.

Groups are used to structure the nine different wasele types, e.g. according to type (group "menu") or use (group "dynamic menu").

If a group is not yet defined, the group *Standard* is used. In order to create a new group, the icon  in the dialog to create a new wasele is activated.

The wasele list can be displayed in two **views**: tree and list.

If the **list view** is enabled (icon  in the head of the wasele list), the wasele are displayed independent of their group affiliation. Via the icon  in the head of the wasele list can be set if one or more wasele types have to be displayed. The respective types are enabled or disabled in the toolbar.

In the **tree view**  all groups and the allocated wasele are displayed.

In child classes additionally the icon  is available via which the inherited, not overloaded wasele can be faded in or out.

The current view can be saved via context menu item *Set the current view as default for all classes* available in the structure tree root. This setting is applied user-dependent to all following opened classes, as long as no other view is saved.

2.5. Projects and ticket integration

In order to be able to edit the already created knowledge base, it is defined that the knowledge base has to be edited in the frame of the task that has to be fulfilled. In Develop this task that has to be fulfilled is described as **Project**. In our case the project could e.g. be called Training or Carconfigurator.

A **Project** contains any amount of main knowledge bases which have to be edited in the frame of the project. The main knowledge bases and their libraries can be edited in the project. Therefore other knowledge bases and knowledge base versions are protected from unintentional editing.

In addition to the definition of the knowledge bases that have to be edited, a connection to a ticket system is also defined on a project. This allows a complete documentation of the changes, because each edited class is logged on the ticket.

There are two possibilities with the selection of the ticket system. Either an external ticket system is integrated, such as camosTicket (based on camosSalesCenter). The two programs communicate via an interface. If necessary, this interface can also be used for other external ticket systems.

The other possibility is the use of **camosTicketXS** which is already integrated in camos.Develop. Then an external ticket system is not necessary, because the required functions are already integrated in Develop.

camosTicketXS is a “Mini” ticket system that is integrated in camos.Develop and that can be used in Develop.

Therefore every developer himself can create tickets for the forthcoming changes on the knowledge base and can then process these tickets. Each class that is edited in the frame of the ticket is logged on the ticket.

In this training we will use camosTicketXS.



Click on the icon in the toolbar in the section of the class tree.

Click on the icon in order to create a new project and allocate e.g. the name “Training”. The project is created via acknowledging with OK and the ticket interface dialog is opened automatically.

“camosTicketXS” is selected as ticket system and this dialog is also acknowledged via OK. Open the project via double click or via the icon .

The knowledge base/knowledge bases that have to be edited in the frame of the project have to be taken into the project. This is carried out automatically if a knowledge base is created in an opened project or via allocating an already existing knowledge base.



Click on the context menu item *Add knowledge base* in the section of the class tree on the label *Main knowledge bases*. Then you select the knowledge base Training example, work version 1.0.

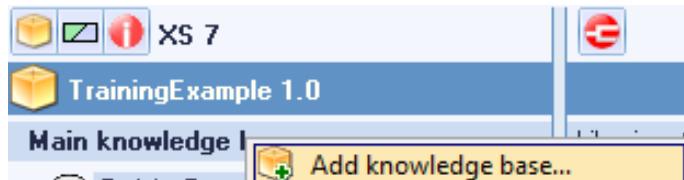


Figure 5: Adding knowledge base to project

In order to edit a knowledge base or the frame it is necessary to create a ticket and to enable this ticket. The ticket gets a meaningful title that describes which actions have to be carried out, e.g. "Change background color of the main form" or "Add product XY".

Open the ticket system via the icon above the class tree.

Create a new ticket via the icon and allocate keyword, e.g. "Create class structure".

The ticket is automatically set as actual ticket if there is currently no other actual ticket.

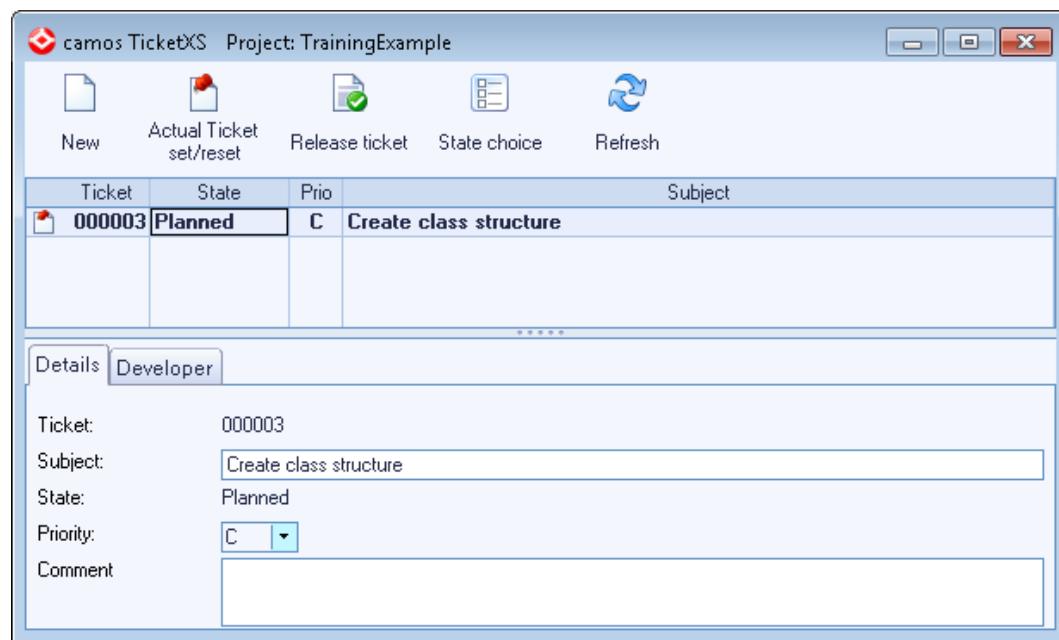


Figure 6: Active ticket in the ticket system (TicketXS)

As from now the frame and the knowledge base properties can be edited and classes can be created.

2.6. Repetition

- A knowledge base contains all information that is required for the solution of the task formulation.
- A -frame has to be allocated to each knowledge base.
- A frame contains global properties such as languages and currencies.
- The surface of camos.Develop consists of the class tree and the workbench as well as of a menubar and toolbar.
- Knowledge bases can only be edited in the frame of a project.
- The connection to a ticket system (camosTicketXS or external ticket system) is defined on the project.
- Classes, the frame, the knowledge base properties etc. can only be edited with an active ticket.

3. Example application “Hello camos Develop”

3.1. Intention

Your first program should generate an output on the screen. As output element a window is used in which the text “Hello camos Develop!” has to be displayed.

To do so, the following four substeps are required:

1. Create start class
2. Create form with label
3. Create constructor
4. Start application

If you closed the development system in the meantime, please restart it.



Open the project *TrainingExample* and then the knowledge base *TrainingExample* which was created in chapter 2.3.2

3.2. Create an application

3.2.1. Create start class

Up to now, the knowledge base *TrainingExample* contains no elements. These will be created in the class tree. The class tree has four views, which can be activated via navigation bar. The navigation bar is located above the class tree.

Program classes

The knowledge base root is the starting point for the creation of first classes. In the context menu of the knowledge base root are e.g. the menu items to create and insert classes, to call the search dialog and the integrity check.

ActiveX classes

Under this node ActiveX-classes can be created. These classes get their properties from interfaces to external programs (*.dll-files). So you can e.g. integrate Microsoft Word or the Acrobat Reader in the own application.

Process classes

Process classes are used in combination with MPF (MultiProcessFacility), so you can use the methods from classes of any knowledge bases.

Value classes

Value classes are used when having values which appear several times. The values can be created once and applied to the features in which they are needed.

In the toolbar above the class tree and the project naming, there are two icons in order to set the display of the debugger.

Debugger

Via click on the icon  the debugger will be opened / activated. The started application can be analysed in the debugger.

If the icon  is activated, the debug studio will be opened in a separate window. If the icon is not activated, the debug studio will be displayed in the area above the class tree.

In the class tree area below the (as applicable) faded in, integrated debugger is shown the area of **opened classes** by default. Here are all currently opened classes displayed in a list which is sorted by names and is independent of class types. The usage of this view accelerates working in the opened classes.

In this default view there is also set that classes are not shown in tab pages. Both settings can be defined via *main menu view -> Options*. The settings can be made for classes as well as for editors within the classes.

Please change the *options in view* as follows so that in the course of the training the settings are the same and you can compare your screen easily with the training material:

Show list of open classes -> deactivated

Display tab pages in workbenches -> activated

Show list of open editors -> deactivated

Display tab pages in editors -> activated

In the first step a new class is created. Since this class starts the application, it is usually called "start".

With the creation of classes you have the choice between two class types: **base class** or **object class**.

The difference is basically that a base class can have **child classes** that inherit its properties. An object class however is the smallest unit and cannot have children.

The name of a class can consist of up to 80 characters, but it has to be unique within the knowledge base. The type (base- or object class) as well as the name of the class can be changed later.

Further information for classes, objects and the object-oriented programming can be found in chapter 4.



Select the view *Program classes* and call the context menu item *New...* at the knowledge base root in order to create a new class. Enter “start” in the field *Name*.

Activate the radiobutton *Object class* and confirm the creation with *OK*.

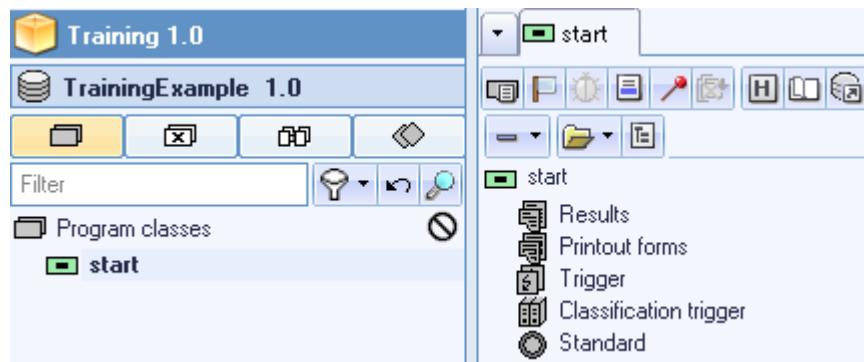
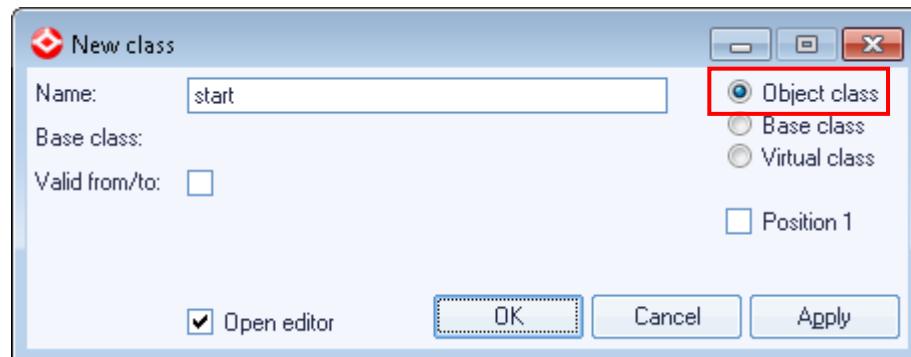


Figure 7: Create start-class

Below the class root *Program classes* you can see the class *start*. Object classes are displayed with the icon

3.2.2. Create a form with a label

In camos Develop windows and dialogs are called **forms**.

Forms allow the user an interaction with the application. On a form diverse form elements can be defined which:

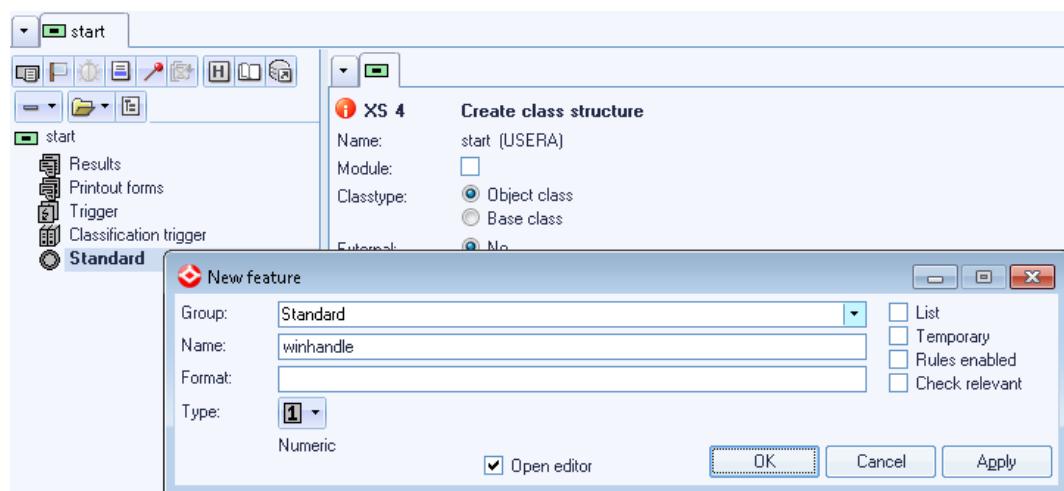
- * display data, e.g. in text fields
- * take up data, e.g. via entry fields
- * process data, e.g. pressing a pushbutton, activating a switchbox

The most part of the form elements combines several of these functions. The form element combobox e.g. displays a list of possible data. The user can select one of these list entries. Due to the new value an action results, e.g. a calculation.

A form is opened via the camos Develop function `WinOpen(<form name>)`. In order to be able to access the form and the contained elements, e.g. the label of a pushbutton, a so-called **form handle** is needed. This is returned by the function `WinOpen`. In order to use it later, it has to be saved in a feature.

3.2.2.1. Practice: Create the form handle

Mark the group **Standard** in the structure tree and call the context menu item **New feature...**. In the dialog **Create feature** you enter “winhandle” in the field **Name** and activate the data type **numeric**. The feature is created with **OK**.

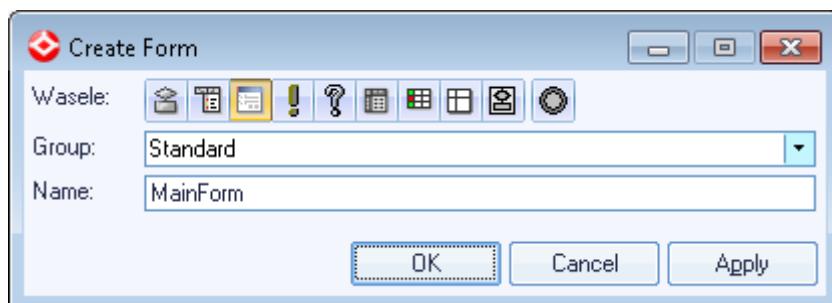


3.2.2.2. Practice: Create the form

In the second step a window is created in which the text “Hello camos Develop!” is displayed. The creating, editing and administrating of forms is carried out in the wasele list, see chapter 2.4.3.

The workbench of the currently opened class *start* (see Figure 3) consists of the structure tree, the wasele list and the class editor.

Click on the icon in the head of the wasele lists. Select in the dialog the icon for forms in the toolbar and enter the name of the form: “MainForm”. Confirm the creation with **OK**.



In the right section you can see now the **form editor** of the *MainForm*.

In the form editor forms are set up. This is carried out in a tree structure. I.e. elements that have to be displayed on the form, e.g. labels or pushbuttons, are created below the form root, see Figure 7.

Via the icon  in the toolbar of the form editor you can open the **form preview**. This preview shows you how the form will look like with the program start.

In order to display a text on a form, the form element **static label** is used.

Select the form root and call *New -> Label static* in the context menu.

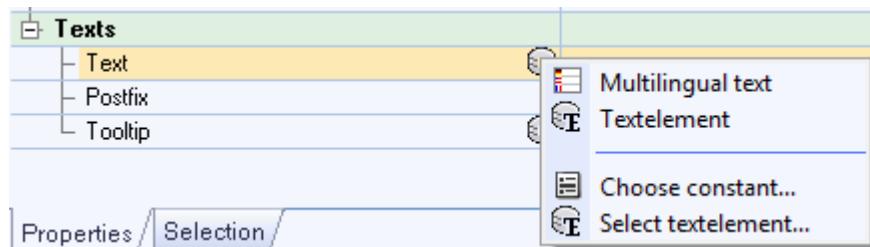


Now you can see the static label you have created under the form root. For each form element that is selected in the form tree certain properties are faded in in the right section. For the static label e.g. the X- and Y-position on the form, height and width, fore- and background color etc. can be set.

Please note that in camos Develop the origin of a form is in the left upper corner.



The text that has to be displayed in the label is entered in the field *Text* below the main node *Texts*. There are three different possibilities that can be set via the context menu of the second column: multilingual text, constant and textelement. The default setting is *Textelement*.



When using the mode *Multilingual text* you can type the text directly into the form element. Constants are Wasele, which are used to maintain static data, e.g. unchangeable texts. The use of constants is covered in chapter 8.6.1 and 14.3.

camos recommends the utilization of textelements, that is why this mode is preset. In chapter 31 you can find a detailed explanation what textelements are and how they are used.

Write the text “Hello camos Develop” (without inverted comma) in the line *Text* and press the **ENTER** Button



With leaving editline the dialog *Create textelement* appears. Keep the default settings and click on the button *Create textelement* to create a new textelement.

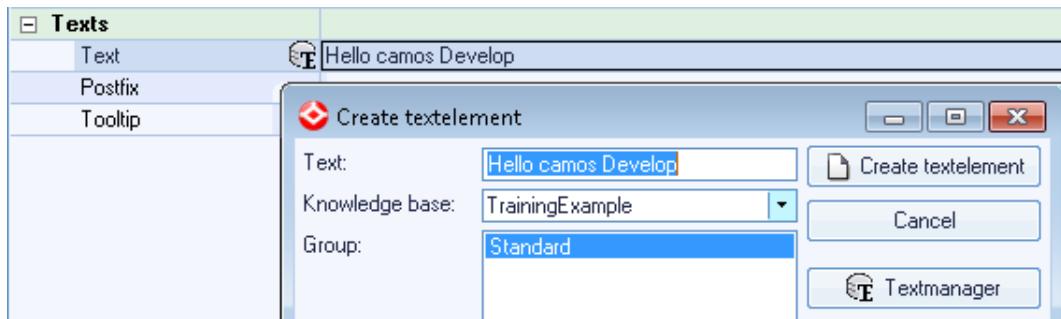
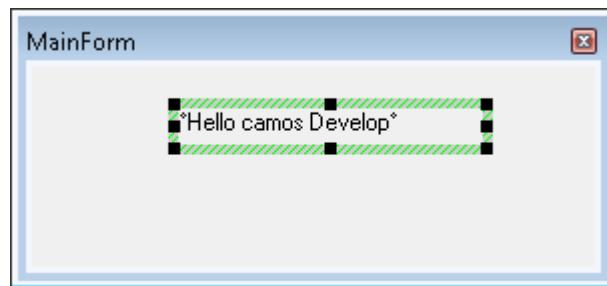


Figure 8: Form Editor – Edit of a static label

Textelements are displayed in the development environment enclosed by two °, at runtime only the content of the Textelement is displayed.

Texts	
Text	°Hello camos Develop°
Postfix	

Check the position and size of the label on the form and the form preview (icon on the toolbar). By D&D, you can move it on the form.



In order to enlarge or reduce the form, you can move the right side of the form. In order to change the size of the label, you select the label in the form preview and drag the frame in the desired size.



In the form editor you can see the field *Name* in the first line of the properties table. This contains by default the description of the form element type, e.g. "Label static" or "Pushbutton". If several elements of the same type exist on a level, they are consecutively numbered, e.g. "Label static_6".



In order to keep the clarity in the form tree, it is recommended to allocate meaningful names, e.g. "Label_Greeting". You can also use the icon to set the naming (property *Text*) or the cause variable of the form element as name.

All special characters (except _) will be converted into the character _ when using the icon to set the naming.

3.2.3. Constructor of the start class

3.2.3.1. What is a constructor?

In the last step has to be defined that the *MainForm* is opened with the start of the class *start*. To do so, the **constructor** of the class *start* is used.

The constructor always runs automatically when an object of a class is created. In camos Develop the constructor is a **method** with the name *new*.

Further information can be found in chapter 11.3.1.

Definition

3.2.3.2. Basics of programming: What is an assignment?

An assignment is used to change the value of a wasele. In camos Develop it has the following syntax:

`Recipient := Value;`

Definition

On the left side of the assignment (character string `:=`) is the recipient. The recipient is principally a wasele that can take up data (writing access to storage space), i.e. a feature or a component.

On the right side of the assignment is the value that has to be assigned to the recipient. The value can

- be individually defined, e.g. `a := 5`; or `Name := „Max Mustermann“`;
- come from a wasele, e.g. a method that returns a value, e.g.
`Sum := Add(2, 4);`
or a constant
`CurImage := !ImageBeetle;`

Frequent error source: The syntax check (see following chapter) does not recognize an error if in the procedure code instead of an assignment `a := b`; a comparison `a = b`; occurs!



All instructions are concluded with a semicolon.

3.2.3.3. Practice: Create a method

Click on the icon in the toolbar of the wasele lists. Select the icon for methods , enter the name “new” and confirm the entry with the Return button.



In the right section of the workbench appears the **procedure editor** of the method *New*. The here entered program code is executed if an object of the class *start* is created.

In order to open a form, the camos Develop function *WinOpen(<form name>)* is used. The form handle that is returned by the function is saved in the feature *winhandle* (see practice 3.2.2.1).

In camos Develop the function call (such as all instructions, assignments etc.) is concluded with a semicolon.

Enter the following program code in the procedure editor of the method *New*:

```
winhandle := WinOpen("MainForm");
```

In order to check the correct syntax, a syntax check is carried out.



Definition

The syntax check checks the elements of a class for validity and syntactic accuracy. It is called via the icon  from the toolbar of any editor or via the shortcut *Ctrl+S*.

Depending on from where the syntax check is called only certain editors of the class are checked. E.g. a syntax check on the root element of the form editor checks the complete form, a syntax check on a form element checks only this form element. If the complete class has to be checked, the syntax check is called on the root of the structure tree. In order to check the complete knowledge base, the context menu item *Integrity check* is carried out on the root of the class tree.

If the syntax check is called in the procedure editor, then in addition to the error check also the program instructions (expressions) are automatically formatted (small/capital letters, blanks between words etc.).

With the closing of the editor of a class a syntax check is automatically carried out in all opened editors. In case of an error a dialog is opened that contains a list of the found errors, see Figure 9.

 *Basics/Syntax check*

Now the syntax check checks if a function (camos Develop function) or method (self-written function) with the name *WinOpen* exists that expects a string parameter.

With *WinOpen* is additionally checked if the transferred form name actually exists: If you made a typing mistake with the name of the form, e.g. "MainFor" instead of " MainForm", the syntax check provides the error message "Unknown form". So you can directly recognize where the error comes from.

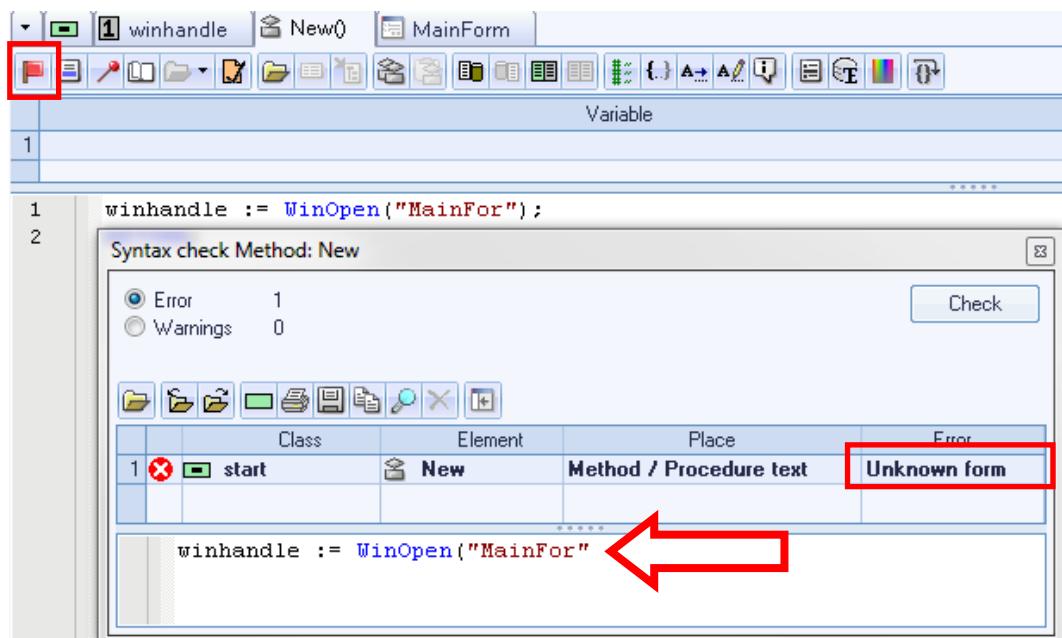


Figure 9: Syntax check in the procedure editor

In general camos Develop is not case-sensitive. I.e. if you typed e.g. “winopen” instead of “WinOpen”, both spellings are correct. The syntax check automatically corrects small and capital letters.



3.2.4. Starting the application

Now the example program is finished and can be executed. To do so, a so-called debug run is started, i.e. the application is executed within the development environment and can be analyzed with different debug functions, if necessary.

Starting the debugger is carried out either via the icon in the toolbar, via the context menu of the class start *Start debugger* or via the hotkey *F5*.

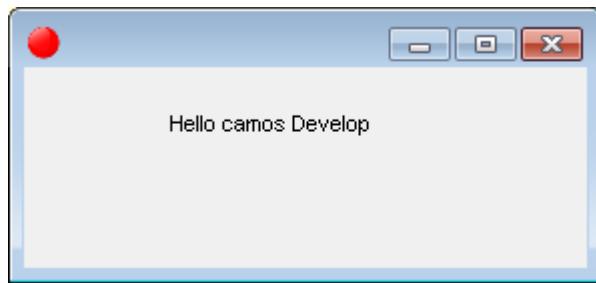
If you prefer the last procedure, please consider that the hotkey always starts the class that is currently marked in the class tree.



Select the class *start* in the class tree and call the context menu item *Start debugger*.



A window is displayed on which you can read “Hello camos Develop”.



In order to terminate the program, you close the form via the X in the titlebar or select the icon from the toolbar of camos Develop.

3.3. Repetition

- The knowledge base root in the class tree is the starting point for the creation of classes.
- In the workbench you can find the information and properties of a class. It is subdivided into structure tree, the wasele lists (knowledge base elements such as methods and forms) and the editor of the opened wasele.
- Texts can be deposited in different ways, it is recommended to use Textelements.
- With the start of a class the constructor is automatically running. In camos Develop the constructor is called *new()*.
- A debug run is the starting of a program in the development environment.

4. Object-oriented programming

4.1. Definitions and basics

In order to map the complex structure of a product to a knowledge base with all necessary elements, rules and connections, camos Develop uses the technology of object oriented programming (OOP). Therefore the backgrounds and the ways of thinking when using OOP should be imparted in this chapter.

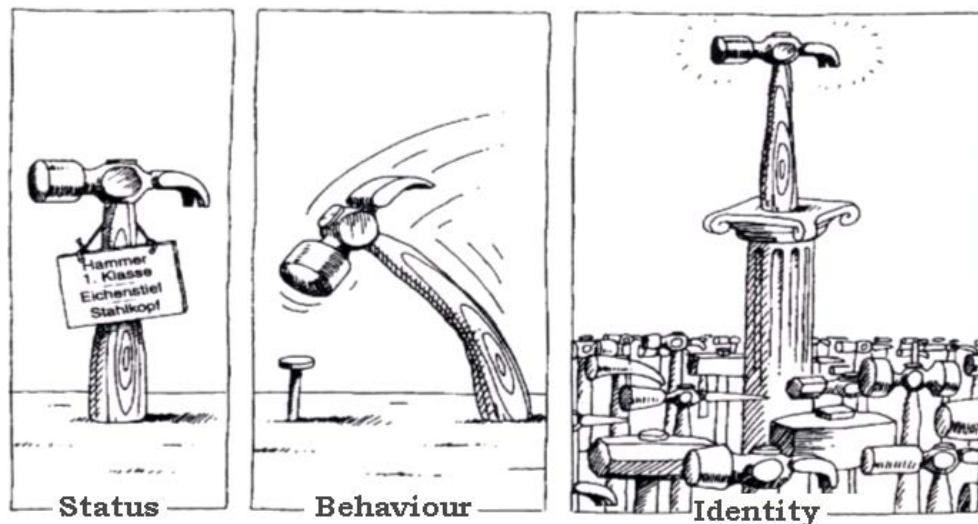
In a simplified way **object orientation** means to orientate on the objects of the reality, to file them in groups (= classes) with similar properties and behaviors and to simplify (= abstraction) them if it is necessary for the type of problem.

4.1.1. Objects and classes

In the reality, an **object** is a recognizable thing or matter that can be differentiated from other things; in the software development an object is also an individual element that mostly reflects an object of the reality.

Similar objects are designed in **classes**. Therefore a class is a (uniquely named) unit that describes a quantity of similar objects. This description comprises the structure (= components), the properties (= features) and the possible behavior (= methods) of later objects of this class.

This can be imagined like the building plan of a machine: with the building plan you can construct any amount of machines that differentiate in details and that can be identified by the article- and serial number. In this connection the building plan corresponds to the class and each constructed machine corresponds to an object.



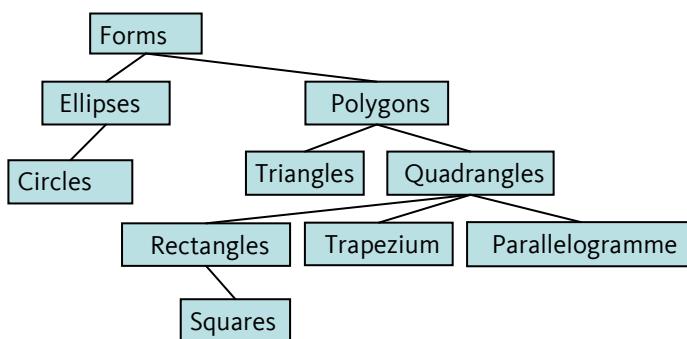
So classes only have a describing and defining function. Objects however exist, have an identity, change their status and have certain behaviors (= methods, rules etc.).

During the development with an object-orientated system such as camos Develop only classes are defined and edited (design time). Objects only result if the program (or the knowledge base) is running (runtime) and therefore objects result from the classes (=**instantiation**).

4.1.2. Inheritance

In order to relate objects not to just one class, but also – just like in reality – to be able to show different detailing levels, you can use the mechanism of inheritance.

I.e. a class is a subclass or **child class** of another class. So e.g. the class *Rectangles* is a child class of the class *Polygons* while the class *Squares* is in turn a child class of *Rectangles*. A child class “inherits” all determinations that were defined in the **parent class**.



In the above example the class *Trapezium* contains - in addition to its own properties - all properties of *Rectangles*, *Quadrangles*, *Polygons* and *Forms*, too. Such inheritances can be displayed perfectly in hierachic tree structures.

During the work on a knowledge base, only classes with their structure and features as well as their operations are defined. Therefore a lot of things can be simplified and combined via a clever inheritance hierarchy.

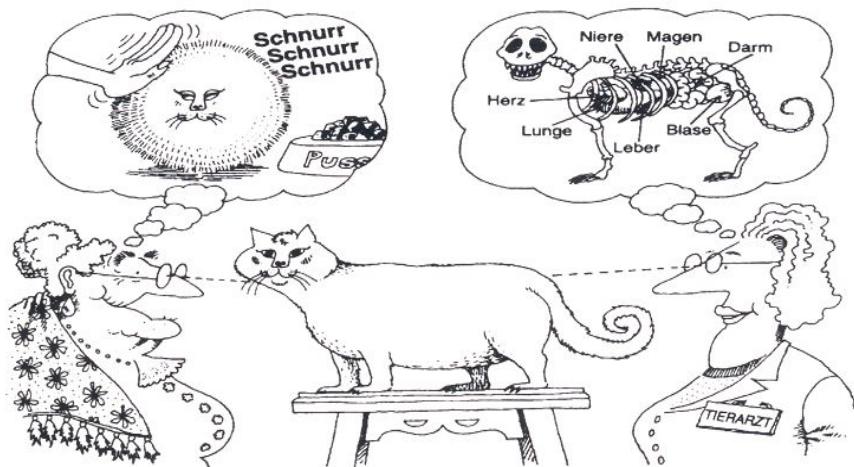
Only during runtime, when the application is started, objects are created according to the classes in the system. The objects behave as it is defined in the respective class.

4.1.3. Abstraction

Each object in camos Develop can be considered as an abstract model of a protagonist that can carry out orders, report and change its status and communicate with other objects in the system.

Now the question is which tasks can be processed by an object of a certain class, which states it can accept and with which objects it can communicate? What has to be specified in the class?

It is useful to keep the type of problem in mind. **Abstraction** means to concentrate on the essential characteristics relative to the problem.

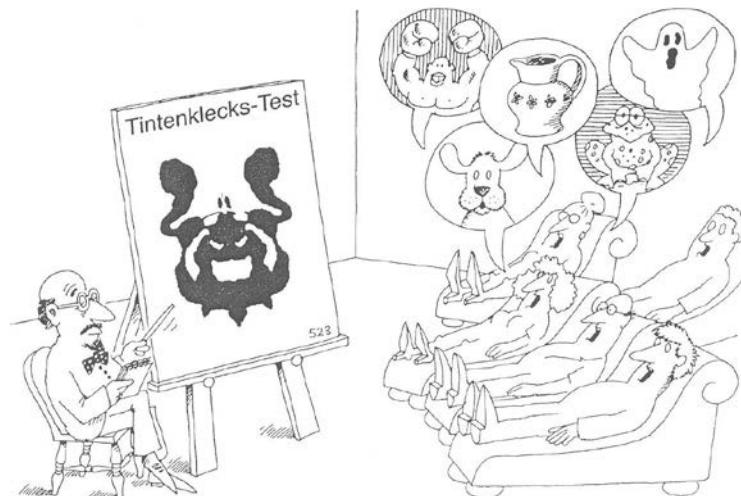


If you want to generate a bill of materials, you need other information than for generating a datasheet. For a price calculation you need other methods and features than for the creation of a drawing. The required level of detailing differentiates strongly from problem. What are the real important facts when the mechanic of a system talks to its operator or when a motor mechanic talks to a car buyer?

It is also important that all developers that work on an application share the same approach to the project. Only then the mechanisms of the object orientation can be used profitable in a team development.

4.1.4. Modularity

Another important aspect is the **modularity**, i.e. the abstractions are packed into independent units that are components of superior elements. Objects in the real world are not only classified, they are also in a relation with each other (**association**). There are two possibilities for relations between objects: aggregation and usage.



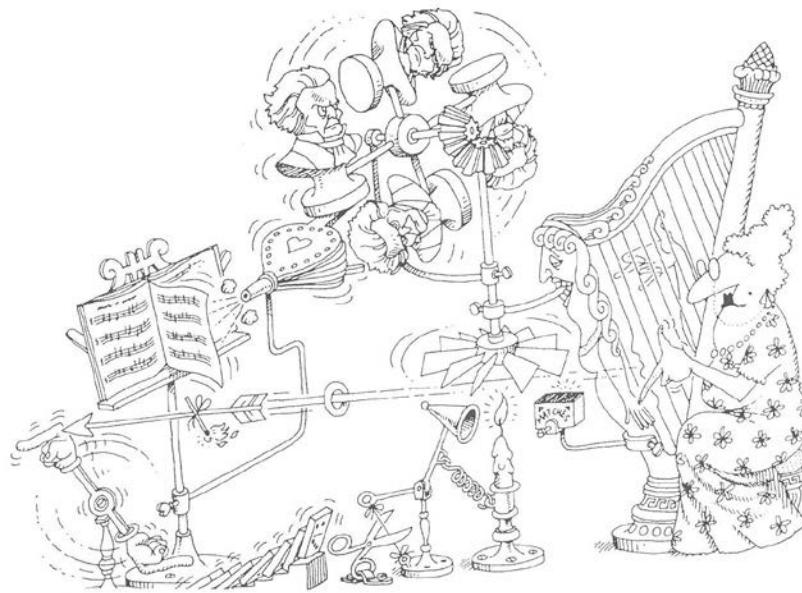
Aggregation means that an object consists of other objects. A car consists e.g. of a chassis, an engine, wheels and many other objects which are in turn constructed of objects. With this we describe the structure of an object. In order to create a structure of objects, it has to be defined inside the classes – in the example it would mean that objects of the class *Car* consist of objects of the classes *Engine*, *Wheels* and *Chassis*.



Usage means that an object uses queries or changes the status or an operation of another object. It has to be considered that also the different usages of objects have to be defined in the classes. Via this cooperation of aggregated objects the most complex systems can be created.

4.1.5. Constructors and destructors

In an object-oriented environment two predefined behaviors for objects are available: constructors and destructors.



The **constructor** of a class is automatically carried out as soon as an object is created from this class. In camos Develop constructors are created via a method named `new()`. By filling the `new` method with source code the object is able to react to its creation, e.g. by setting up its components.

As a counterpart to this there is the **destructor** of a class. This method is automatically triggered when an object is destroyed again. The destructor is often used to start automatic storage routines or security queries. In camos Develop this method is named `delete()`.

4.2. Object Orientation in camos Develop

4.2.1. General

camos Develop is a strictly object-oriented system. Everything that is created in camos Develop (method, form, feature or rule) has to be created in a class.

An application is started with instantiating an object from a class. This start object contains all further objects with their states and behaviors. Via constructors (`new()` method) in the object and its components, then defined program flows, database queries and form displays can result with the start.

This instantiation of an object from a class is carried out via the function `Start debugger`. This function can be accessed via the context menu of a class, via the main toolbar and via the main menu item `Debugger -> Start`.

4.2.2. Types of classes

4.2.2.1. Base classes

Symbol of a base class

Base classes are classes from which child classes can be derived. Base classes are used to group child classes and to avoid multiple definitions of similar properties and behaviors.

It is generally recommended e.g. to create a base class *Modules* above all single modules of a machine that has to be configured. This is done in order to inherit a property or method that is required by all modules. The needed property would have to be defined only once inside the base class and would then be inherited to all modules. This is a lot easier and clearer than defining the same property for every single class separately.

4.2.2.2. Object classes

■ Symbol of an object class

In camos Develop, object classes are used to easily create objects from these classes. Contrary to base classes you cannot create further child classes from an object class.

Object classes have the advantage that the respective object can be selected with very simple mechanisms in a form element *configuration box*. Object classes are useful if you have for a certain component (e.g. the engine of a car) different precise selection possibilities (e.g. 90 PS Diesel engine or 110 PS Otto engine). In this case, the object classes *Diesel90* and *Otto110* are defined as child classes under the base class *Engine*.

A special form of object classes are the *article classes* (icon ■■■■) which are mentioned here only to complete the listing. They have the same behavior as object classes; they are however not manually created but automatically generated via a database link.

?

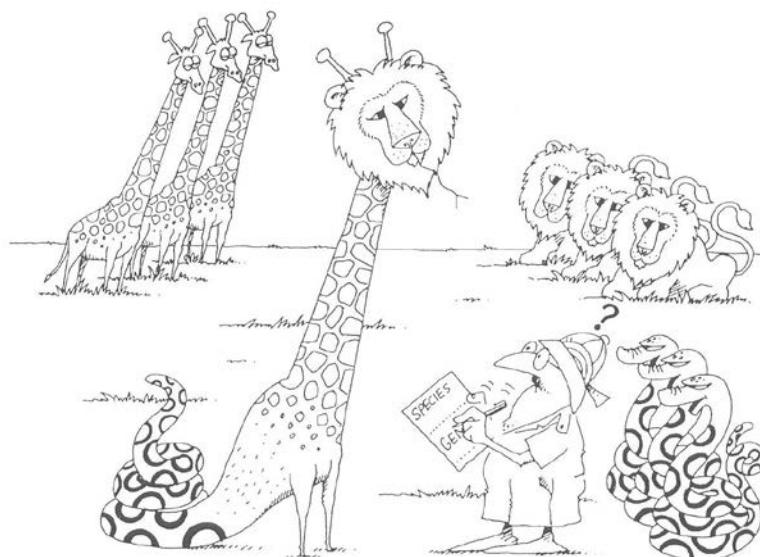
Basics/Class types/Article class

4.2.2.3. Virtual classes

■ Symbol of a virtual class

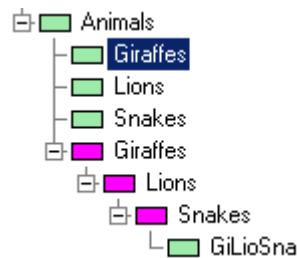
In camos Develop a so-called multiple inheritance is represented via virtual classes. **Multiple inheritance** means that a class inherits properties and methods from several parent classes.

The problem of our animal researcher in the picture could be imaged in camos Develop as follows:



There are classes for giraffes, lions and snakes in which the corresponding properties are contained. Then a virtual class for giraffes is created (the symbol for virtual classes is colored in purple). This just means that possible child classes of the virtual class *Giraffes* will inherit all properties from the giraffe.

If a further virtual class is created under a virtual class, then children inherit properties from the virtual class and from the class above.

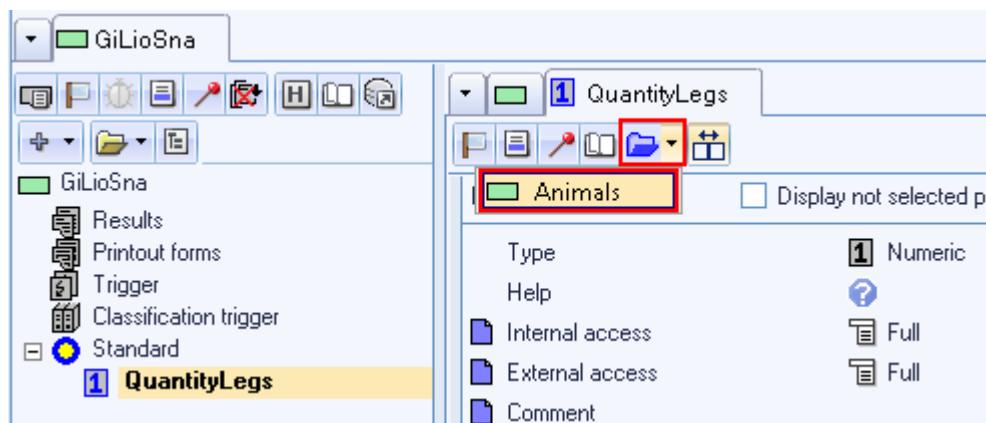


In this example the class *GiLioSna* inherits all properties from the classes *Giraffes*, *Lions* and *Snakes*. If an element exists in several parent classes, the definition of the parent class which is closest to the child class is considered – in this case the class *Snakes*.

4.2.3. Inheritance and overload

In camos Develop inherited elements are always displayed with a blue colored icon. The color blue signals that this element was not defined at this position, but that it was inherited from a superior class.

If in the animal example of 4.2.2.3 a feature *QuantityLegs* was defined in the class *Animals*, the feature is displayed in the class *GiLioSna* as follows:

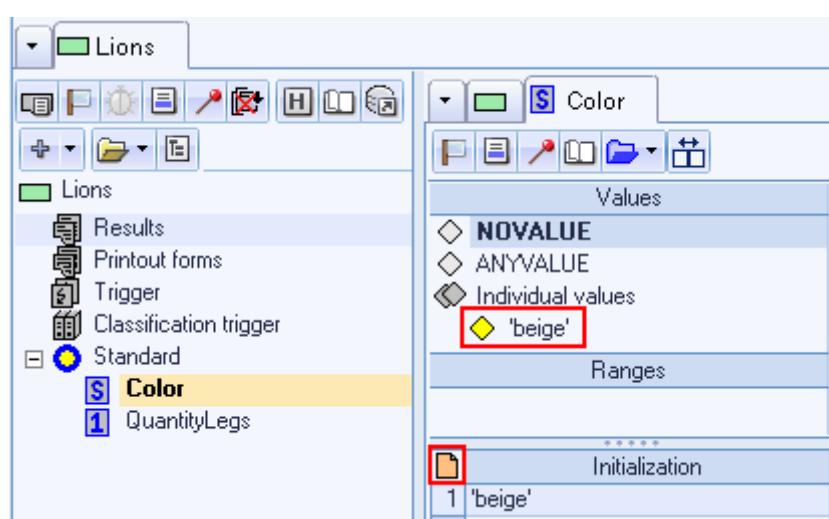


The blue symbol shows that it is an inherited numerical feature. In the dropdown-list the class where this feature is defined is displayed, in this case it is the the class *Animals*. Via a click on the blue open icon you can jump to the original definition.

Sometimes it could happen that you want to define an exception or a further specialization of a property in a child class. Therefore many inherited elements can be **overloaded** in the child class. I.e. the inherited value is locally covered by different value. This can be methods, forms, settings or initialization values. Overloads are displayed with orange-colored symbols.

Back to the animal example in chapter 4.2.2.3: In the class *Animals* the feature *Color* was defined. In the child class *Lions* the init value of this feature was overloaded with the value 'beige' which can be recognized from the orange-colored symbol.

In order to make an overload undone, you have to click on the orange-colored sheet symbol. For overloaded elements without the sheet symbol (e.g. forms or methods) you just delete the overloaded element and get back the inherited definition.

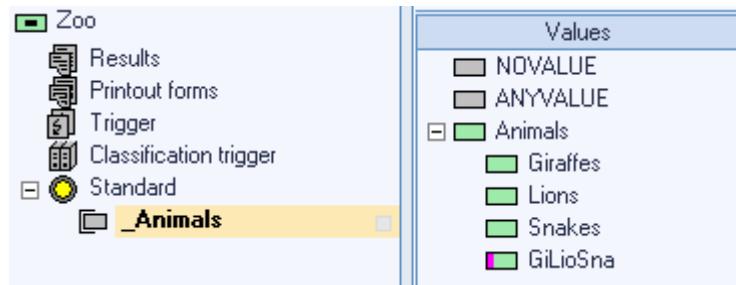


4.2.4. Components – Aggregation

In camos Develop the aggregation is imaged via so-called **components**. If an object has a sub-object, a component is created in the class; the component is typified by the assignment of the class of the sub-object.

The sub-object is only created (= instantiated) through an init value, an assignment in a method or a selection of the affiliated object class in a form.

In extension to the class structure of chapter 4.2.2.3 an object class *Zoo* was created. This class contains a component *_Animals* of the class *Animals* (in camos Develop component names always begin with an underscore).



In the area *Values* in the component editor the potential values of the component are displayed. Potential values are all classes which are derived from the component class (*Animals*). The icon informs you that the class *GiLioSna* is derived from (at least) one virtual class.

With this an object was not yet instantiated, but only a base was created that can take up an object that can be formed by the class *Animals* or one of its child classes.

The allocation of a component with an object is acted out via an init value or in a method via an assignment. A method of the class *Zoo* could e.g. contain:

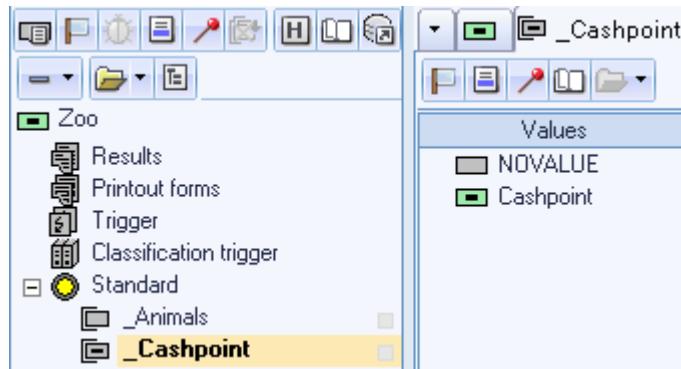
```
_Animals := „GiLioSna“;
```

With this the component *_Animals* is allocated with a new object of the class *GiLioSna*. If the class *GiLioSna* contains a constructor, it would run immediately because a new object is instantiated.

If the component-class is an object, the object can only be instantiated by this class.

If our *Zoo* contains e.g. an object class *Cashpoint* and from this the resulting component *_Cashpoint*, the assignment or the instantiation of the cash point could look like this:

```
_Cashpoint := 1;
```



If an object has to be deleted, you just assign the value *NOVALUE* to the component:

```
_Cashpoint := NOVALUE;
```

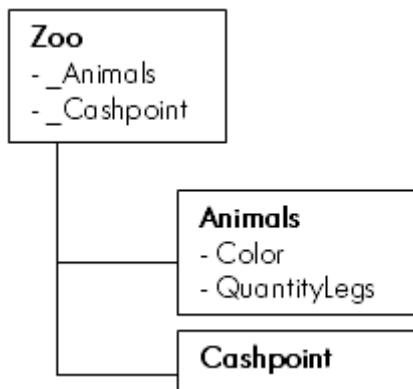
This syntax is valid independently of the origin of the object (object- or base class component).

Alternatively you can use the camos Develop function `ObjDelete()` as well. In order to delete the object `_Cashpoint`, the function call could look like this:

```
ObjDelete(_Cashpoint);
```

4.2.5. Components – Mutual use

If you would now start the application, the definitions above would result in the following object tree:

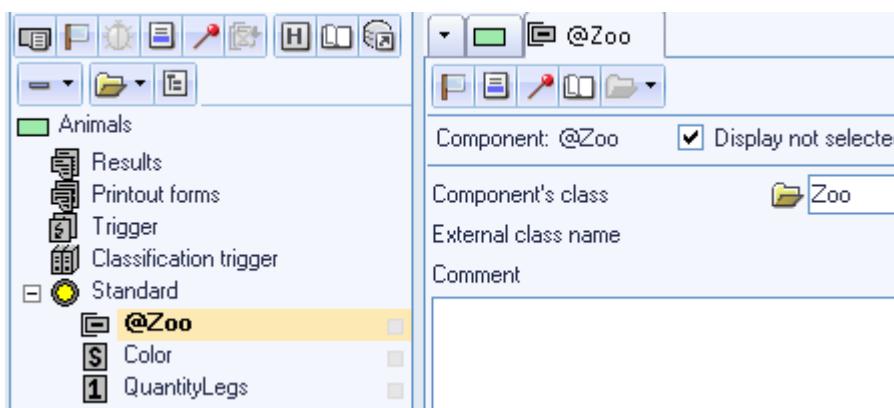


The object `Zoo` contains objects of the classes `Animals` and `Cashpoint`.

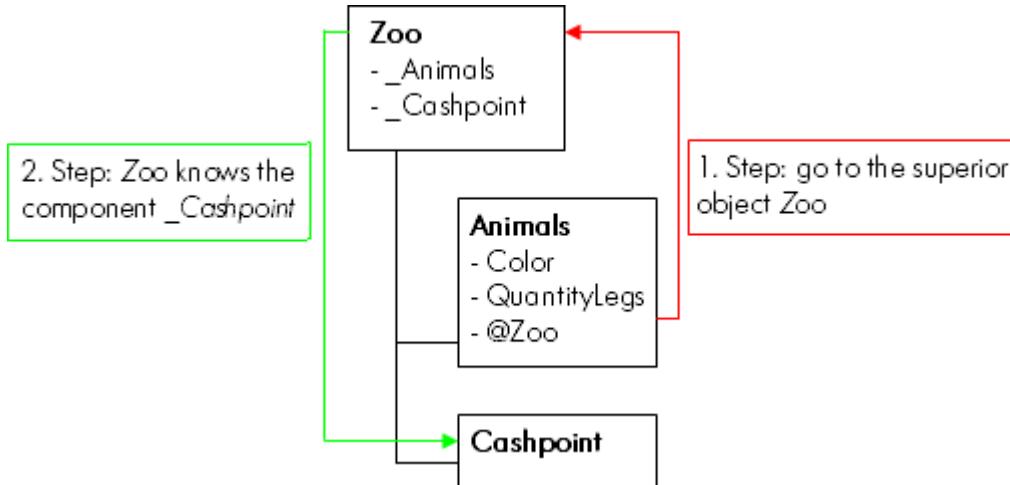
The different existing objects of an application are usually interacting – mutually calling methods or reading values from features. Any object knows the components it contains. But it does not know whether it is an independent object or if it is a child object of a superior component itself.

In order to access the properties of a component, you specify the component name followed by a dot and the name of the element. So the query of the feature `Color` of an `Animal` (from inside the class `Zoo`) would look like as follows: `_Animals.Color`

In order to “grab” upwards in the object tree, a so-called **predecessor component** is needed. This is also typified with the specification of a class. The name of a predecessor component always starts with an @.



Via the predecessor component `@Zoo` inside the class `Animals`, you are able to access from the component `_Animals` via the superior Object Zoo to the neighboring component `_Cashpoint` with `@Zoo._Cashpoint`.

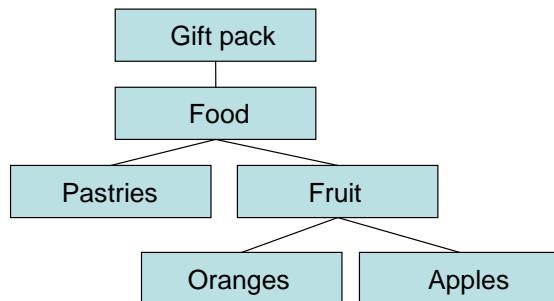


A predecessor component `@Zoo` always queries internally the next higher-order object if it comes from the class `Zoo` until a matching object is found. If no object is found, the predecessor component remains blank (NOVALUE).

4.3. Common traps of object-oriented programming

4.3.1. Mixing inheritance and association

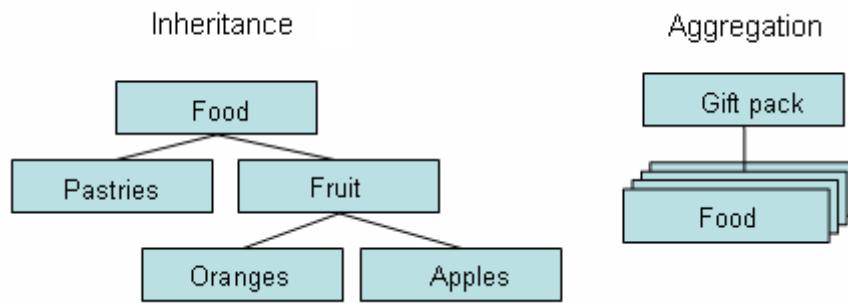
Principally you can display the class hierarchy with its inheritance structure very nicely in tree structures. Just imagine a structure that deals with food. We want to configure a gift pack that consists of fruit and pastries. In broad outline it doesn't take a long time to look like as displayed below. But what does this graphic say?



In the class hierarchy derivations can also be expressed with “.. is a ..”. “Oranges are fruit, fruit is food”. So far okay, but is food a gift pack?

No, food is a component in the gift pack. Inheritance relations (“.. is a ..”) were mixed with aggregation (“.. consists of ..”).

It would be correct to create 2 graphics, one for the inheritance and one for the aggregation:



4.3.2. Wrongly used inheritance

Inheritance is not suited to carry out fast extensions of the system via derivations and extensions or overload if the principles of the specializations from top to bottom are violated.

If in a system a class *Square* exists that has the operations *Shift*, *Scale*, *Rotate* and *Draw* and a class *Rectangle* is quickly needed, you could have the idea to derive a class *Rectangle* from *Square* and extend it with the operations *ScaleWidth* and *ScaleHeight*.

This does not guarantee anymore that all objects of the class *Square* are really squares!

Through the inheritance an object of the class *Rectangle* is also an object of the class *Square*. If another method assumes that all squares have the same height and width, the errors are pre-programmed.

4.3.3. Correct abstraction grade

With the creation of an application with which a house has to be configured, you cannot just say how fine the granularity of the components has to be.

We want to demonstrate this problem with the question how the property windows could be mapped in the application:

- the complete number of the windows is a feature of the house for the calculation of the required work routine to install the windows
- the window area in square meter is required as feature for the calculation of the cost of materials
- the windows are created as individual components of the house; therefore each window can be entered with its measures
- the house contains rooms as components which in turn have the windows as components – so the windows can in addition to their measures also contain the installation situation in the room

- the house has apartments as components; these have rooms which have the windows as components – or the number of windows or the window area as feature

This listing could be continued forever. It just cannot be said that any definition is wrong or false. It always depends on the type of problem.

However, you should check if the required element

- is only entered without own features and methods; then you can image the element in a feature or
- has own features and methods that are required; then it has to be created as a class.

For the creation of features and components has to be principally considered in which class they are created.

In the object orientation you talk in this context also about the responsibility principle. Each class is responsible for the handling with its data and the class which contains the essential data for a task is also responsible for the task.

5. Structure of the product

5.1. Intention

As already mentioned in the introduction, an application for the configuration of a car should be created during this training course. The user of the configurator has the choice between three car models: *VW Beetle Convertible*, *VW Golf Individual* and *VW Passat Variant*. The selection and later configuration of the individual models is carried out on a form.

In this chapter you learn how the three car models are imaged in the class tree and how they are displayed on a form.

5.2. Set up the class structure

5.2.1. Structure considerations

The product that has to be configured with the application is a car. For the product "Car", there are three variants: Beetle, Golf and Passat. In general, the three models have common properties, e.g. all car models have wheels and an engine. But they are also different: the Beetle e.g. has a soft top while the Golf and the Passat have a hard top.

According to the OOP, objects with the same properties should be combined under base classes. The car models themselves are defined as object classes, because further classes cannot be derived from a model.

The following structure results from the considerations and prerequisites above:

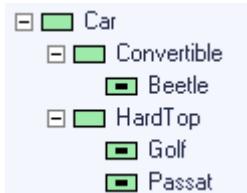


Figure 10: Class structure of the car

Properties that all models have in common are defined in the class *Car* and therefore inherited to all its child classes. Properties that are valid for just one particular model are defined in the respective child class. In the class *Convertible* is e.g. defined that the Beetle has a soft top. For the time being, this property is only handed down to the Beetle.

Principle of the expandability: If now another convertible model is added, it would be derived from the class *Convertible* and therefore inherit the property *Soft top*.

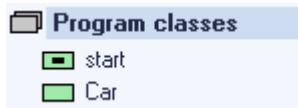
5.2.2. Practice: Create classes

Create the class structure according to Figure 10.



Mark the class tree root *Program classes* and select the context menu item *New...*. Enter “Car” in the field *Name* in the dialog *New Class*; leave the radio button *Base class* untouched and confirm with *OK*.

Interim result:

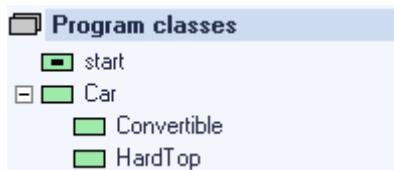


In order to create the base classes *Convertible* and *HardTop* that are derived from *Car*, you select the class *Car* in the class tree and call the context menu item *New...*.

In the dialog *Create class* you enter “Convertible” in the field *Name* and confirm with *Apply*.

Then you enter “HardTop” in the field *Name* and confirm with *OK*.

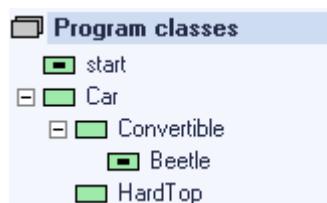
Interim result:



In order to create the object class *Beetle*, you select the menu item *New...* of the context menu of the class *Convertible*.

In the dialog *Create class* you enter “Beetle” in the field *Name*, select the radiobutton *Object class* and confirm with *OK*.

Interim result:



The models *Golf* and *Passat* are created below the class *HardTop*.



Select the context menu item *New...* of the class *HardTop* and enter in the dialog *Create class* “Golf” or “Passat” in the field *Name*. Select each time the radio button *Object class*.

Final result:



5.3. Create components

5.3.1. What is a component?

With the creation of object classes for the car models Beetle, Golf and Passat the basis to create objects is made. Only with the instantiation of a class the definition in the class tree becomes a "living" object with an own identity.

How and where is the car object instantiated? And how can it be displayed on a form?

The class *start* starts the application. On the *MainForm* that was created in chapter 3.2.2.2 the three car models should be offered for a selection. To do so, first a connection between the class *start* and the object classes *Beetle*, *Golf* and *Passat* has to be established.

This is carried out via a so-called **component**.

A component is used in order to integrate a class B in another class A. According to OOP this means that "class B is component of class A" or "class A contains class B".

In camos Develop components have always the prefix "_" (underscore), e.g. *_Component*.

If the class *start* has to contain the product "Car" that has to be configured, a component of the class *Car* has to be created in the class *start*.

Open the class *start* via double click in the class tree.

Definition



5.3.1.1. Groups in the structure tree

In the workbench of the class *start* you can see the structure tree with the nodes "Results", "Trigger" and "Standard" as well as the wasele lists with the method *New()* and the class editor of the class *start*.

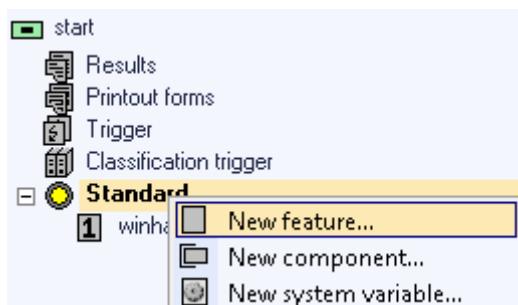
The node "Standard" in the structure tree is a so-called **group**.

Groups serve to structure wasele in the structure tree. It can be very confusing if a class contains a lot of wasele (features and components). Therefore groups are used in order to structure wasele, e.g. according to type (feature, component) or use (help features, configuration components).

A group that contains wasele is displayed with the icon . If the group is empty it is displayed with the icon . If the group was defined in a higher-ranking class, that is it concerns an inherited group, the group is displayed with a blue frame, e.g. .

The group *Standard* is automatically created. Further groups can be created via the context menu item *New group...* of the root of the structure tree.

Groups are the starting point to create features and components. In the context menu of a group you can find the menu items *New feature...* and *New component...*



5.3.2. Practice: Create component of class Car

Components can be created on a group in the structure tree via the context menu item *New component...*. In this case the name of the class from which the component has to come from has to be manually entered (see Figure 11)

 This procedure holds a certain risk for beginners because often only the field *Name* is adapted, the field *Class* however is filled with a different value than originally planned. Because always the currently marked class in the class tree is set as *Class-* and this does not necessarily need to be the class, which should be applied as a component.

The safer way is to drag the class that has to be integrated as component via D&D directly from the class tree into the structure tree. In this case the property *Class* in the dialog *Create component* is initialized with the correct class.

 Select the class *Car* in the class tree and drag it via D&D into the workbench of the class *start*. Release the mouse button directly above the group *Standard*.

Then the dialog *Create component* opens. The field *Class* already contains the name of the class that has to be integrated as component in the class *start*: *Car*. The field *Name* is also initialized with the prefix (underscore) and the name of the class: *_Car*.

Except the prefix “_” you are free to choose the name of the component, it does not mandatorily have to be the class name. You can name the component e.g. _Products as well.

For reasons of clarity, all components you create during the training should have the same namings like the classes from which they are created. Therefore please leave the name proposal _Car unchanged.

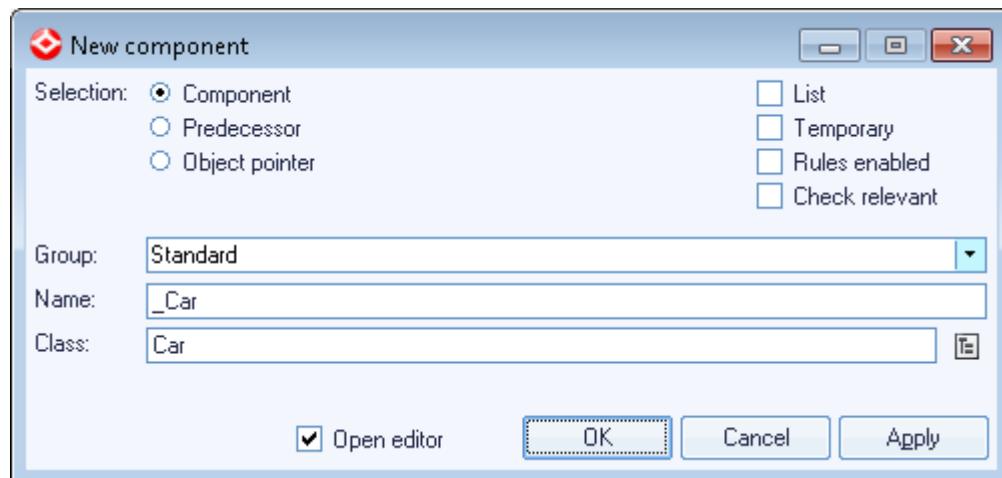
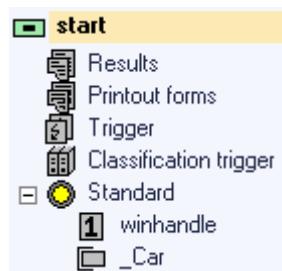


Figure 11: Creating a new component

Activate the switchbox *Open editor* and confirm the creation of the component with *OK*.



Now you can see the component *_Car* below the group *Standard*.



Components always begin with the prefix “_” (underscore), also called “underline”. This is a convention in order to differentiate components from other class properties such as e.g. features. If you didn't enter the prefix manually, it is automatically added with the creation of the component.

The class *start* now contains (via the component) the class *Car* and can therefore directly access the properties of the car models.

5.4. Selecting a model on the form

5.4.1. Use of configuration boxes

The car models have to be displayed on the form *MainForm* of the class *start*. In camos Develop the form element **configurationbox component**, short: *configbox component* is used.

In a configbox component the possible object classes of a component are displayed and can be selected.

5.4.2. Practice: Create a configurationbox for Car

 Open the *MainForm* of the class *start*.

 To this, change the *wasele* type in the *wasele* list to forms and double click the *MainForm*.

The static label that was created in chapter 3.2.2 can be furthermore used as a label for the configbox.

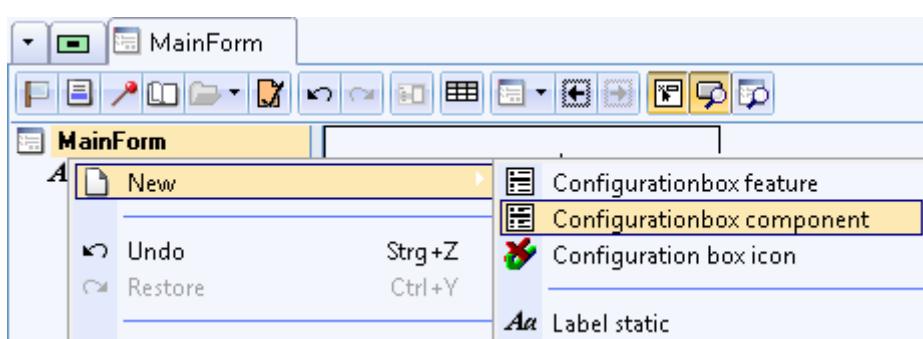
 Select the static label in the tree of the form editor, click into the line *Text* and delete the textelement °Hello camos.Develop° with the delete key. Now enter the text "Model selection" and leave the input field. Thereby a new textelement will be created.

Below the field *Text* there is the property *Postfix*. The character you define in this line will automatically be added to the content of the label at runtime. Type a colon as Postfix, so that at runtime the text "Model selection:" is displayed.

 Open the form preview via the icon  in the toolbar of the form editor and check the position and size of the label.

On the right side next to the label the configbox component has to be displayed in which the desired car model is selected.

 Select the form root and select *New -> Configurationbox component* in the context menu.



Now you have to tell the configbox what it has to display.

The wasele that provides the values or the contents for a form element is in general described with the term **cause variable** (or *EFG*).

The cause variable of a configbox component is (as the name says) a component, in this case *_Car*.

Select the configbox in the tree of the form editor. In the properties of the configbox (right editor) you see the line *Cause variable* under the main node *Special*.

Click on the icon  of the line *Cause variable*. This opens the **item selector**. Here you can see all wasele of the class *start* that you can use as EFG of a configbox component. Select the component *_Car* and confirm with **OK**.

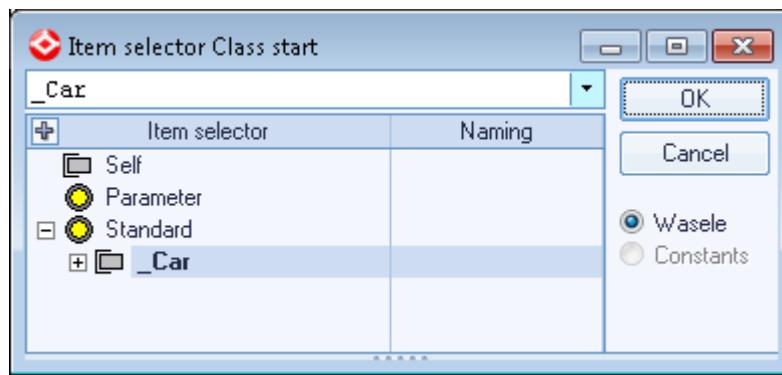
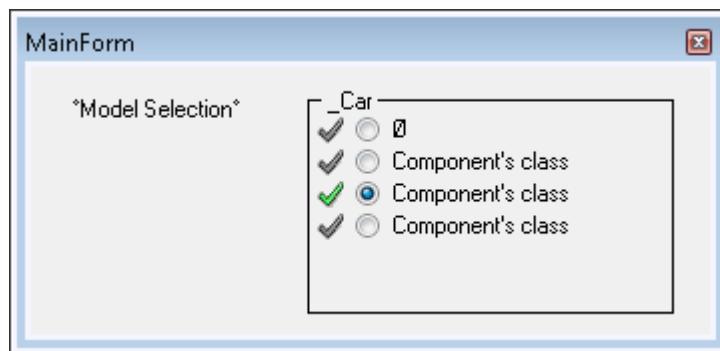


Figure 12: Item selector dialog

Move the configurationbox via D&D in the form preview, right besides the label "model selection" and adjust the size.



5.4.2.1. Component values

The adaptations on the form are now concluded. Now the configbox knows that it has to display the possible objects of the component *_Car*.

It is not yet defined which classes are possible objects. This allocation is carried out on the component *_Car* itself.



Open the editor of the component *_Car* via double click on the component in the structure tree of the class *start*. In the left section of the component editor you will find the *Value list*.

Definition

The term **value** can be described as “possible value”. Values are needed if a possible value of a component or a feature has to be displayed in a configbox or if this possible value has to be ruled (see chapter 17.4).

Values of components are always the object classes that are derived from the component class.

The values of features have to be manually defined – except NOVALUE and ANYVALUE. This is carried out via the context menu item *New...* in the value list, see chapter 10.3.2.1.

Possible values of *_Car* are therefore the object classes *Beetle*, *Golf* and *Passat*.

Since a configbox component displays the assigned values of a component, the possible values *Beetle*, *Golf* and *Passat* have to be assigned as values. This is carried out via a double click on the desired object class in the value list.



Double click the classes *Beetle*, *Golf* and *Passat* in the value list. The assigned values are displayed in the structure tree with the icon below the component.

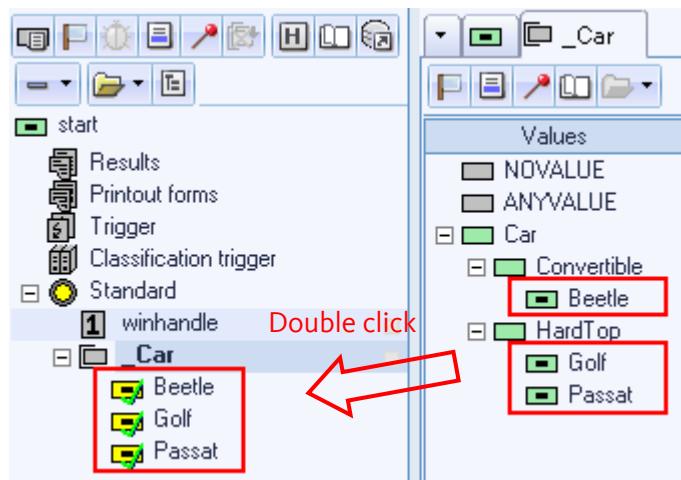


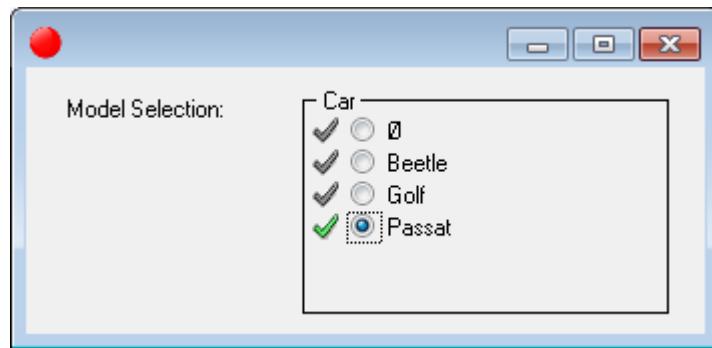
Figure 13: Apply values to the structure tree



After the assignment of the object classes as values, you start the application. To do so, you select the class *start* in the class tree and select the context menu item *Start debugger*.

On the form the three car models Beetle, Golf and Passat are displayed in a configbox and can be selected. With the selection of a model (by clicking) an object of the clicked value (object class) is created automatically.

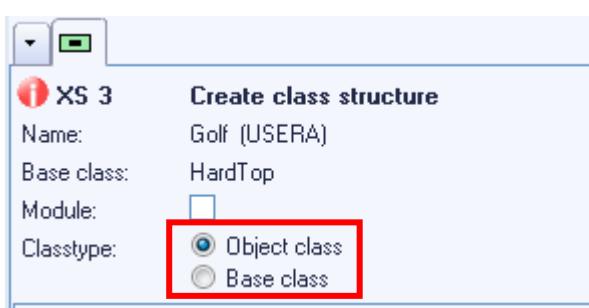
Existing objects are indicated by a green flag. Values that are possible but not yet selected, have a grey flag.



5.5. Tips & Tricks

5.5.1. Creating classes

- If you made a spelling mistake with the name of a class, you can change the name via the context menu item *Rename...* of the class.
- If you selected a wrong class type, you can correct this via opening this class and changing the type in the class editor:



- In the dialog *Create class* you will find the switchbox *Position 1*. If this is enabled, the newly created class is always placed at the uppermost position in the class tree, otherwise at the last position. In order to keep the sequence that exists in the figures, you should disable the switchbox.

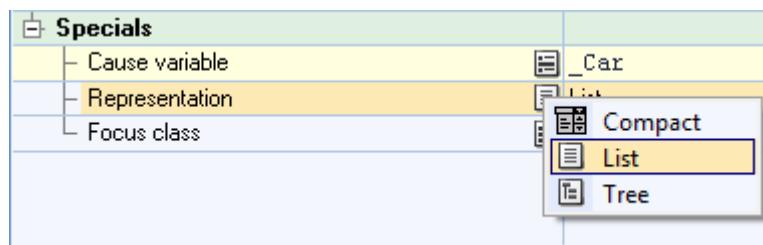
5.5.2. The NOVALUE symbol

The character \emptyset symbolizes the value NOVALUE. In this case NOVALUE means that the value "nothing" (no car model) is selected.

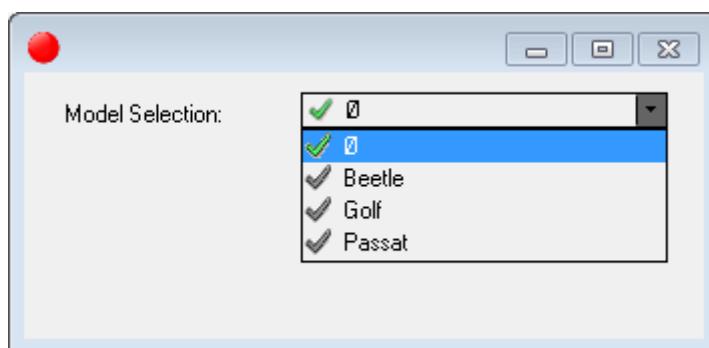
An instruction to replace this symbol by an expression such as e.g. "no model" you can find in chapter 8.5.

5.5.3. Representation of configboxes

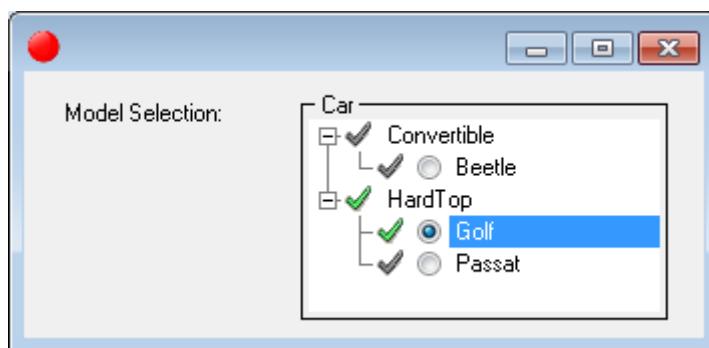
The *Representation* with which the configbox and the values are displayed on the form can be set in the form editor of the configbox.



In the current representation (list) the values are displayed in a list with radio boxes. If you change the mode to *Compact*, you get the following representation (also called combobox representation):



In the third possible representation (tree) the configbox is displayed as follows:



Via the option *Expand tree* you can define whether the tree should be displayed with all nodes opened or all nodes closed.

5.6. Repetition

- Classes that have common properties are combined under *base classes*.
- Classes from which no further classes are derived have to be defined as *object classes*.
- A *component* is a class that is part of another class.
- Components and features are created below *groups*.
- Component names always begin with the prefix underscore, e.g. “_Car”.
- In a *configurationbox component* the possible values of a component can be displayed and selected.
- For form elements that have to display the value of a wasele, a *cause variable* (EFG) has to be specified.
- The icon  opens the *Item selector*. This dialog contains all wasele of the current class that are allowed as cause variables in the concrete context.
- “Possible values” are called *values*. The applying of a possible value (e.g. object class to a component) as an assigned value is carried out via double click in the value list of the component editor.

6. Defining components of the car

6.1. Target

Each car model is set up of a series of components, e.g. chassis, engine, gear, in which several variants are possible for each component; the engines differ in the type, the required fuel type and power.

After the selection of the desired model by the user, he should be able to configure these components. To do so, the possible components are shown as classes and allocated as parts to the car.

In this chapter you repeat the creating of classes and components as well as the creating of values and configuration boxes. Furthermore you will learn how components can be initially valued in order to initialize them with a certain value.

6.2. Set up of the product structure

In the first step the components Engines and Wheels are integrated in the application. The variants Otto engine (model O50 and O120) and diesel engines (model D70 and D110) are available for the component Engine.

The wheels can be selected with steel- or aluminum rims in the wheel sizes 155, 175, 185 and 205.

6.2.1. Exercise: Show components as classes

How can these components be shown as class structure? First we have to consider which components can be combined in base classes because they have equal properties.

1. The engines as well as the wheels are components and therefore they can be combined under a base class *Modules*:



Actually the class *Modules* is not necessary, because modules are not components that can be selected by the user. In the OOP however often so-called “Container classes” are defined that are solely used for the passing on of properties to all child classes. All components can e.g. be allocated conveniently with the property “Article number”. 

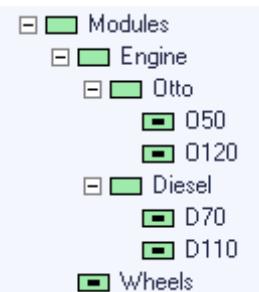
2. The engines differ in the function principle (Otto- or diesel engine) and therefore the base class *Engine* is subdivided into two further base classes *Otto* and *Diesel*:



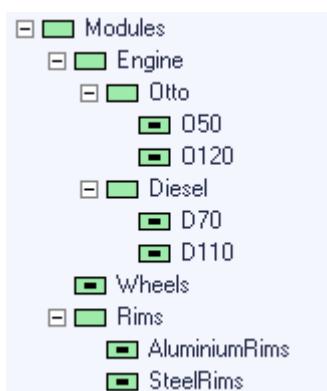
3. The figure of the module *Wheels* which is from a technical view composed of the modules Tyres and Rims is somewhat more complicated.

There are however different variants of wheels, but each of these variants results from a combination of tyre and rim that is desired by the user.

Therefore the module *Wheels* is defined as object class that contains the modules Tyres and Rims as parts:



4. The steel- and aluminum rims are combined under the module *Rims*:



5. The tyres only differ in their size and cannot be further subdivided:



Add the just created class structure to your knowledge base. Please consider that the class *Modules* is created on first level in the class tree and not as child class of *Car*!



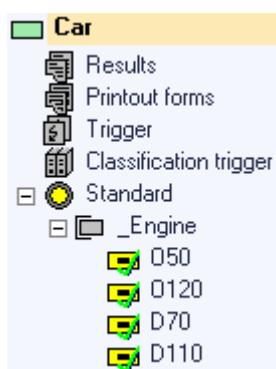
If several classes have to be created one after the other on the same level, then it is recommended to acknowledge the creating via the button *Apply*. In this case the dialog stays open and does not have to be reopened.



6.2.2. Exercise: Allocate components and init values

Now the previously defined modules have to be allocated as parts (= components) to the car.

Open the class *Car* and define a component of the class *Engine* in the structure tree of *Car*. Apply all engine variants (= object classes) as value in order to be able to display the engine in a configurationbox Component.



According to the structure considerations in chapter 6.2, step 3, a car consists of wheels while the wheels in turn consist of tyres and rims.



Add a component of the object class *Wheels* to the class *Car*.

Since the user cannot configure the wheels (possible values would be "Wheels yes" and "Wheels no) and therefore no configuration box for the wheels is needed, no values have to be applied.

The component *Wheels* is initially valued due to the missing selection possibility for wheels and the condition that a car always must have wheels.

An **Init value** is a value that is assigned automatically to the Wasele with the instantiating of the object.

Init values are defined in the feature- or component editor. To do so, a table *Init value* is available below the value list. Existing values, e.g. freely selectable values (only with features), can be deposited as init value in this table.

Init values are used for the definition of standard values. All car components can e.g. be initially valued with a value so that a finished (pre-) configuration exists immediately with the selection of a vehicle model.

Definition



Open the editor of the component *_Wheels* and drag the value *Wheels* via D&D to the field *Init value*.

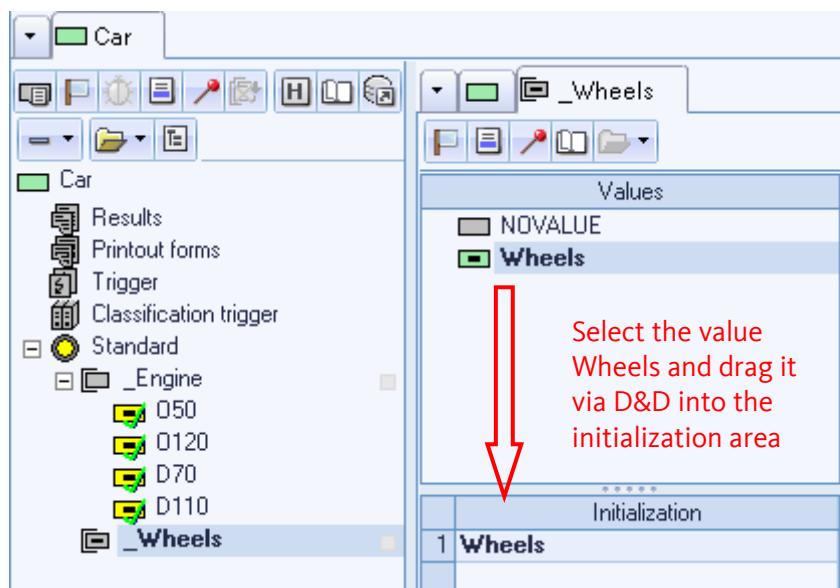
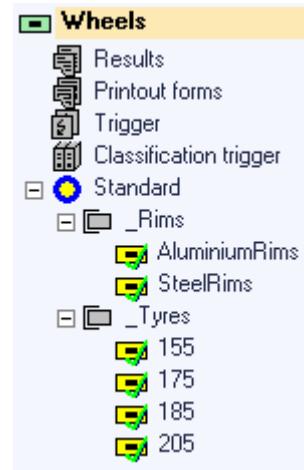


Figure 14: Defining init values

With the creating of a car object the component *_Wheels* is now always initially valued automatically with the value *Wheels*.

Momentarily is only defined that a car consists of an engine with the possible types O50, O120 etc. and wheels. Since the wheels in turn are composed of tyres and rims, this allocation has to be carried out in the class *Wheels*.

Open the class *Wheels* in the class tree and in the structure tree you add components for the modules *Rims* and *Tyres*. Apply all possible object classes as values.



6.3. Displaying components on form

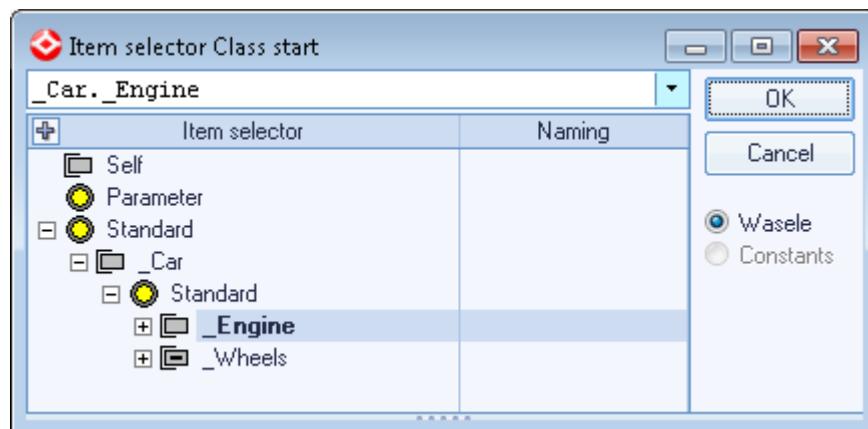
The selection of the engine as well as of the tyres and rims is carried out via configboxes (see selection of the vehicle model, chapter 5.4.2).

6.3.1. Exercise: Create configboxes for components

Open the form *Main form* in the class *start* and add a new configurationbox Component for the engine. Open the form preview and set position and size.



In order to assign the cause variable, you open the Wasele selection dialog of the property *Cause variable* and select the component *_Engine* of the class *Car*.



Proceed the same way to create the configboxes for the tyres and rims. Experiment with the display types and options of configboxes.



Please consider with the selection of the cause variable that the corresponding components are in the class *Wheels* which is in turn a component of *Car*. Therefore the path from the class *start* to the component *_Rims* would be *_Car_Wheels_Rims*.

Example for the result of this exercise:

Form tree	Interpreter

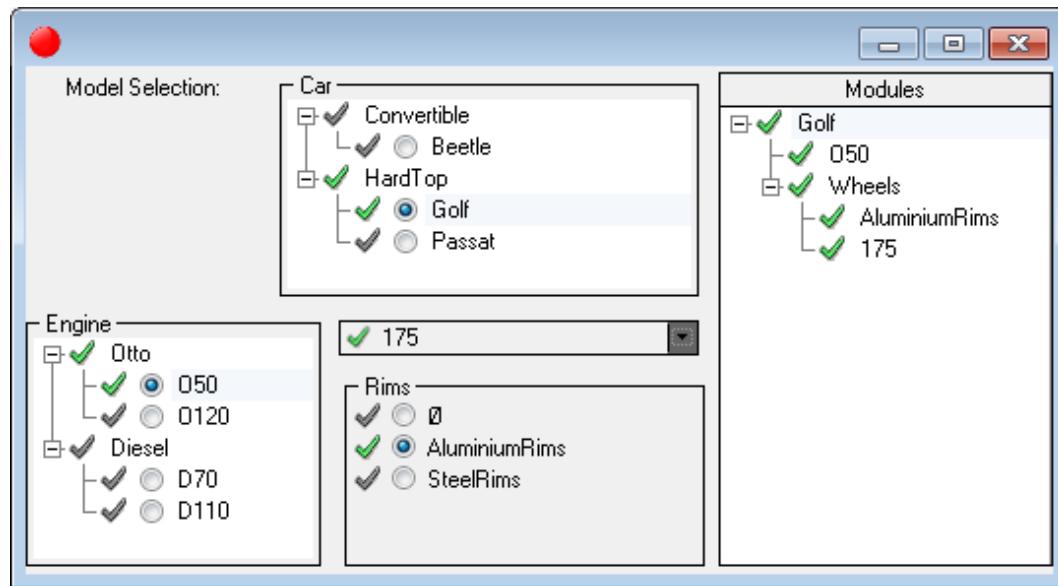
6.3.2. Exercise: Form element Component tree

The form element Component tree is used to display the current configuration of the module components – the so-called object structure.

Hardly any configurator application can do without the component tree. It is a browser for the complete or partial object structure. Due to its many functions it is the ideal basis for the navigation through the product that has to be configured.



Add a component tree to the main form. In the field *Root* you enter the component *_Car*. Therefore the component tree displays the object structure as from the object *Car*. Optionally you create a textelement as heading (e.g. "Modules") on the automatically created column 1. *Component tree column*.



In the above figure you recognize a multilevel object structure in the form element Component tree:

```

Car
  Wheels
    Tyres
    Rims

```

If you try to express this structure verbally, you get the following statements:

A	Car
has	Wheels.
Wheels have	Tyres.
Wheels have	Rims.

If these statements concur logically with the knowledge about the structure of the product that has to be configured, you can assume that no errors slipped in with the definition of the class- and component structure.

More information for the options and the usage of the component tree will be given in chapter 13.

6.4. Repetition

- An init value is used to allocate a Wasele automatically with a value with the instantiating of the object.
- Possible init values of a component are the classes that are present in the value list. The value is dragged via Drag & Drop from the value list to the init value section.
- The access to properties of components is carried out via the syntax *_Component.Property*.
- According to the OOP it is sometimes useful to define container classes whose sole meaning is to pass on properties to the child classes below.

7. Model selection via the product image

7.1. Target

The selection of the vehicle that has to be configured should be carried out via graphics for an optical improvement of the application surface. The user does no longer select the model in the configbox Component but he clicks on the product image of the desired model. The image files that are needed in the following exercise can be found in the folder *Graphics* on the training CD.

In this chapter you will learn how image files are imported to the knowledge base, how graphics are integrated in a form and how an object is instantiated via source code.

7.2. Importing external graphic files

7.2.1. Use of constants

First the premade product images that have to be displayed on the form later have to be read in to the knowledge base. The Wasele type Constant is used especially for graphics.

Constants are used to deposit persistent data in the knowledge base, e.g. menu item icons or texts for message dialogs. The prefix ! (exclamation mark) is always in front of the name of a constant, e.g. `!IconNew`.

The value of a constant is static while the value of a feature or a component can change during runtime, i.e.

- * constants can only be accessed reading
- * the value of a constant is not determined during runtime but already defined with the creating and then it can only be changed manually in the development system
- * constants can also be accessed without a present object

Constants can be of the data type numerical, String, RTF, Date, Currency, Graphic, or HTML. Furthermore String-, RTF- and HTML-constants can be defined multilingual (see chapter 8.6.1).

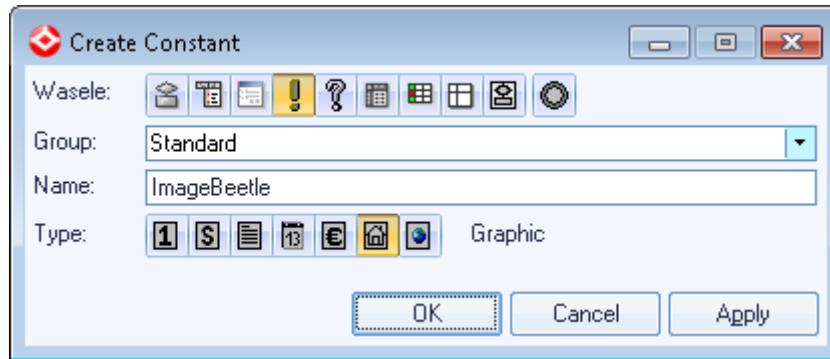
Definition

7.2.1.1. Exercise: Create graphic constants

Constants are Wasele and therefore they are administrated analog to methods and forms in the Wasele list.

Open the class *start* and select the New icon from the toolbar of the Wasele list. In the appearing dialog you click on the icon  for the Wasele type Constant and then on the icon  in order to determine the data type Graphic. A corresponding description is specified in the field *Name*, e.g. `!ImageBeetle`, because the constant has to contain the product image of the model Beetle.





The graphic constant editor is opened after a click on OK.

7.2.2. Graphic constant editor

The editor of a newly created graphic constant first contains a white 16x16 pixel big graphic. Basic image editing functions can be executed via the toolbar on the left border of the editor, e.g. pen, filling bucket, pipette, rotate and reflect image, swap colors etc.

In the upper toolbar of the editor you will find functions such as Create new, Open, Save, Copy and Insert graphics.

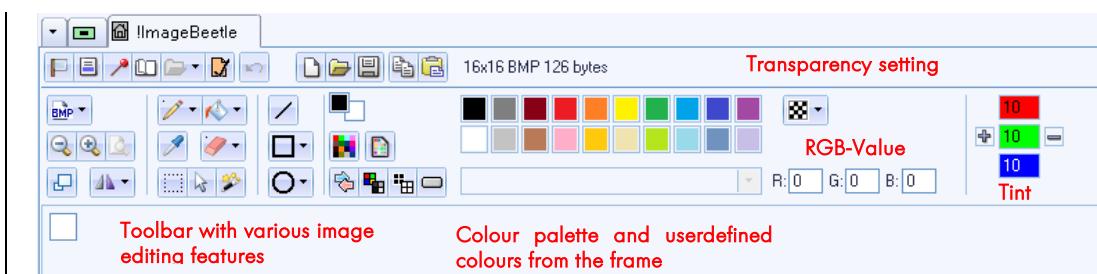


Figure 15: Graphic constant editor

7.2.2.1. Exercise: Load graphics

Select the icon from the upper toolbar in order to read in an external graphic file. In the file selection dialog you open the folder *Data\Graphics* on the training CD.



If you work via an application server, please ensure that the dialog accesses the CLIENT computer! By default the directory structure of the application server would be displayed in this case. The desired computer is set via the icon in the dialog on the right top.

By default the file selection dialog displays only bitmap graphics. The data type that has to be loaded can be set or changed in the combobox *File type*. Since the product images are in the JPEG-format, you select the entry *JPEG* in the combobox.

Now you flag the file *PicBeetle.jpg* and acknowledge the dialog with *Open*.

Now the constant `!ImageBeetle` contains the product image of the model Beetle. You proceed analog with the product images for the models Golf (file name: `PicGolf.jpg`) and Passat (file name: `PicPassat.jpg`).

7.3. Display of graphics on a form

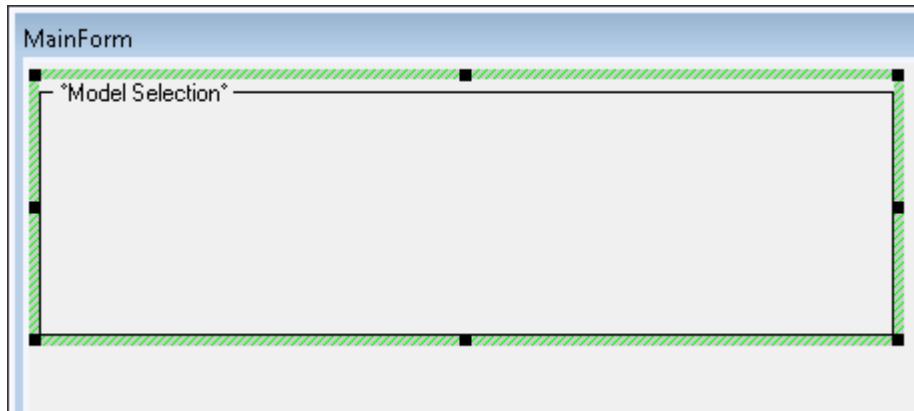
The **Form element Graphic** is used to display a graphic on the form. Since altogether three of these graphic form elements are required, these are embedded in the form element **Groupbox** for reasons of the surface design. A groupbox has no operational function but it is used to group several form elements optically and/or thematically.

7.3.1. Exercise: Form element Graphic

Open the *Main form* of the class `start` and delete the configbox Component for the selection of the car and the label with the naming *Model selection*.

Then you create a new groupbox. Change the position and size so that three graphics can be displayed in the groupbox.

In the property table of the groupbox you enter “Model selection” in the field *Heading*. Since a Textelement with this content already exists, it is recognized and applied.

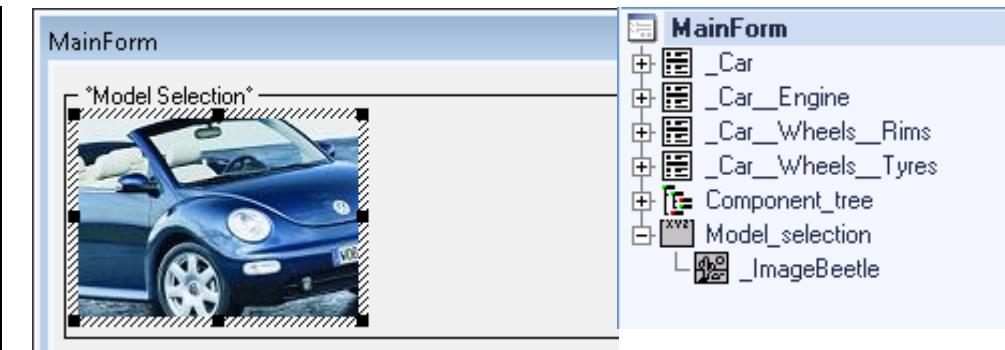


The form elements that should be displayed within the groupbox have to be created in the form tree below the groupbox. The X- and Y-position of these elements refers to the left top corner of the groupbox.

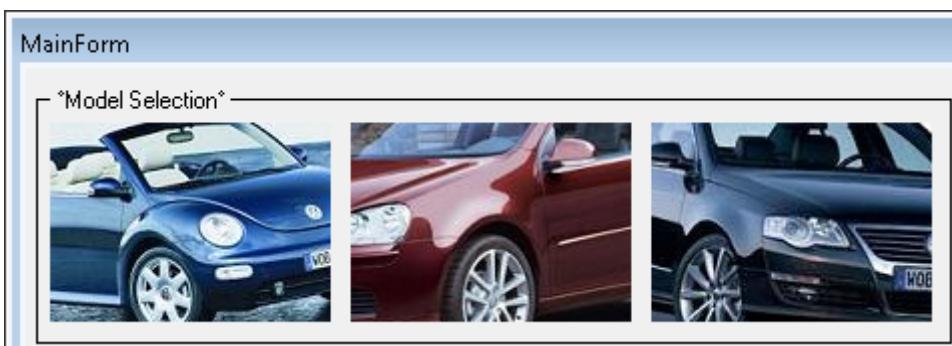
Flag the groupbox in the form tree or in the form preview. Create a new form element of the type Graphic via the context menu and position it in the left third of the groupbox.

Assign the graphic that has to be displayed: `!ImageBeetle`.





You proceed analog with the form elements for the graphics `!ImageGolf` and `!ImagePassat`.



7.4. Manual object instantiation

7.4.1. Selection trigger of a form element

While the configbox Component carries out the creating of the car object automatically with the clicking on a value, this has to be implemented manually with the selection of the car via a graphic.

Since the object has to be created if the user clicks on a product image, the procedure code is deposited in the selection trigger of the corresponding graphic form element.

The procedure code that is deposited in the **Selection trigger** is always executed if the user clicks (= selected) on a form element, e.g. a pushbutton, a graphic or a switchbox.

The selection trigger is filled on the tab page *Selection* in the lower part of the form editor of the form element:

[Properties](#) / [Selection](#) / [Double click](#) / [Move](#) / [Drag&Drop](#) / [Hover](#) / [Hotkeys](#) /

7.4.2. Exercise: Create car object

In the selection trigger of the graphic of each model a procedure code is deposited that instantiates an object of the type *Car* with the value (name of the object class that is derived from *Car*) *Beetle*, *Golf* or *Passat*.

Open the selection trigger of the graphic form element for *!ImageBeetle* and deposit the following procedure code:

```
_Car := NOVALUE;
_Car := "Beetle";
```



The first line deletes the possibly existing configuration of another model, the second line creates an object of the model "Beetle".

For the graphics of the Golf and of the Passat you also deposit the corresponding code for the instantiation of the car object.

Pay attention to the correct spelling of the class name! If you named a car class e.g. "Passant" instead of "Passat", you have to specify the name that exists in your class tree in inverted commas. Start the application and check if the models are created correctly.



7.5. Tips & Tricks

7.5.1. Display of graphics

By default a graphic is displayed on the form in the actual size of the graphic cause variable. In order to adapt the graphic to the size of the form element, the form element options *Proportional* and/or *Size adjust* have to be enabled.



Representation	
Visible	<input checked="" type="checkbox"/>
Frame	<input type="checkbox"/> No border
Dynamic	<input type="checkbox"/>
Proportional	<input checked="" type="checkbox"/>
Size adjust	<input checked="" type="checkbox"/>
Antialiasing	<input type="checkbox"/>

The option *Dynamic* controls if the form element is updated automatically or not if the graphic cause variable gets a new value. Since the graphic cause variable is a constant in this case, i.e. its value cannot change, the option stays disabled.

The enabling of the option *Dynamic* would have no effect – as to this state of the development. With regard to the creation of applications with a more complex surface, the thoughtless enabling of *Dynamic* however would affect the performance, because with each internal form update a check is carried out if the graphic cause variable has changed.



7.5.2. Changing several form elements simultaneously

In order to change the properties of several form elements simultaneously, e.g. all graphics and the groupbox should be displayed with a dark blue line border, you flag all affected form elements, set the new property and click on the icon in order to apply the current value of the property for all flagged form elements.

If the icon is clicked without specifying a new value for a property, the value of the last selected element is applied.



7.5.3. Undoing actions

The **Restore memory** is used to make an action undone. This memory is in the toolbar of the form editor. The last instruction is made undone via a click on the icon . The last action that was made undone is restored via the icon .

In the knowledge base options (see chapter 16.4) can be set under *Undo depth* how many conditions can be saved or restored. By default 10 conditions are saved.

7.5.4. Tooltips

At many places it is useful to define so-called **Tooltips** that give useful notes and explanations for handling the application. Tooltips can be defined on almost all form elements.

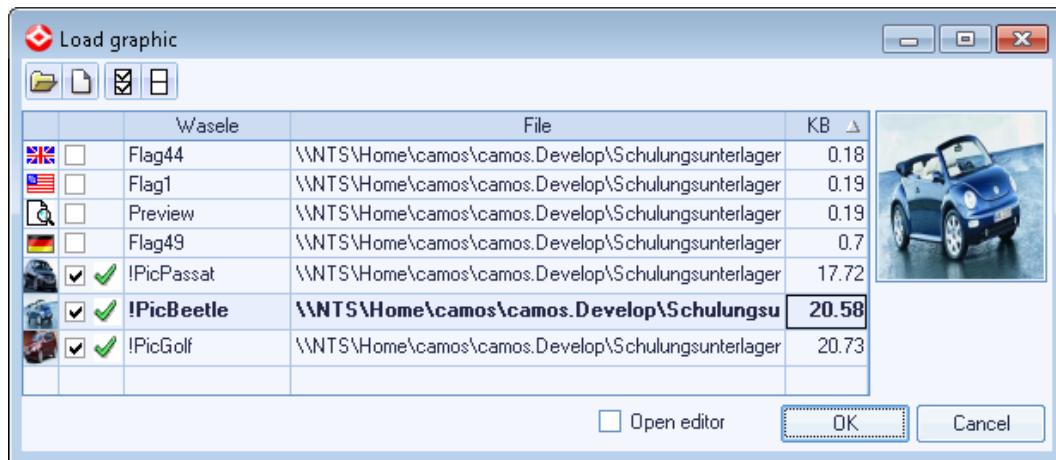
Name	_ImageBeetle
Type	Graphic
Help	
Stylesheet	
+ Geometry	
+ Representation	
+ Handling	
- Texts	
Tooltip	*Please click here to start the configuration for the Beetle.*
+ Specials	
Graphic	!ImageBeetle

The text is displayed automatically for approx. 5 seconds in a bright yellow box if the mouse is moved over the form element.



7.5.5. Load graphic

Via the context menu item *Load graphic* in a Wasele list several graphics from the file system can be created simultaneously as graphic constants.



A folder can be selected via the icon . Then the contained graphic files are displayed in the dialog. The file name is initially valued as Wasele name, it can however be changed via entering a different value in the column *Wasele*.

After acknowledging this dialog with OK, the selected files are created as graphic constants in the Wasele list.

7.6. Repetition

- Data that does not change during runtime can be deposited as constants, e.g. product images or error messages.
- Constants have an exclamation mark as prefix, e.g. !Constant.
- The value of a constant cannot be changed during runtime.
- The form element Graphic is used to display graphics on a form.
- In the selection trigger of a form element a procedure code can be deposited that is always executed if the user clicks on the form element.
- The tooltip of a form element is a text that is displayed automatically if the mouse is over this form element.

8. Basics of multilingualism

8.1. Target

The layout of the autoconfigurator should be bilingual in order to be able to use it in the international sales. Therefore it should be possible to migrate the language. In order not having to create the application separately for each dialog language, you have the possibility to deposit labels, message texts, product names etc. in any amount of languages.

In the following chapter you will learn how to add a new dialog language to the knowledge base and how to deposit multilingual namings.

8.2. Creating a new language in the frame

8.2.1. Main- and secondary languages

The languages that have to be available in a knowledge base are defined in the frame. In addition to the main language (in this case English) any amount of secondary languages can be defined.

Languages are always coupled to a country. English is e.g. spoken in Great Britain as well as in the USA. Therefore the identification of the languages is carried out via the country code of the country.

The country code is decisive with the use of functions for the migration of the language, the currency etc. It corresponds to the international telephone area code.

Definition

The presetting of the main language depends on the language with which camos.Develop is started (menu *View -> System language*). Any language can however be changed later to the main language.

8.2.2. Exercise: Create new language

Open the *Training frame* via the *Frame maintenance* or via the icon  of the toolbar. Flag and lock the section *Languages*.



Momentarily only the main language English (United States) is present:



The screenshot shows the 'Language' section of the 'Frame TrainingFrame Working version 1.0' interface. On the left, there is a tree view with 'Language' selected. Under 'Language', 'United States' is listed. On the right, there are several input fields and dropdown menus:

Code:	001
Icon:	 
Language ID:	
Country ISO:	US
Country:	 United States



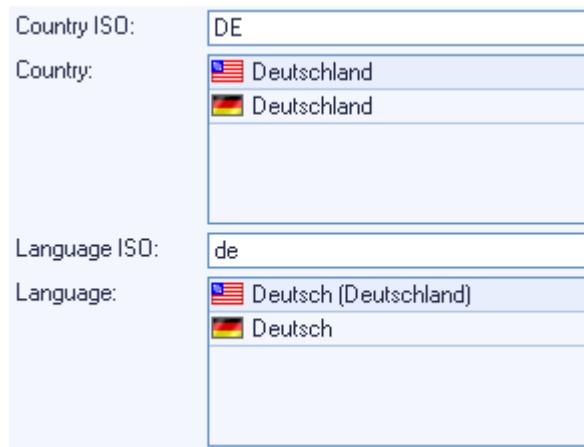
Select the context menu item *New* in the upper left table in order to create a new language. The dialog *Create language* is opened. In this dialog you can select from all languages that are known by the operating system. Select Germany. After confirming with *OK*, click on the pushbutton *Init value from system* in order to read in the *System init values*, e.g. date formats and separators.

If several countries exist for a country code, e.g. France and Monaco have the same country code 33, a separate window appears in which you can select the desired country.



Figure 16: Creating new dialog language

After a click on *OK* the new language is displayed in the left upper table. In the right section you can see the properties of the language. Here you can e.g. enter the translations for the country name and the language description.



The translations and abbreviations of the months and weekdays are displayed in the two right upper tables and here they can be edited.

The **Fontaliases** for the current language are displayed in the lower right table. Momentarily only the fontalias *Standard* which was created automatically with the creation of the frame is available; the fontalias was allocated with the font Microsoft Sans Serif 8.

A fontalias can have a different font or formatting for each language. If no specifications for a secondary language were made, then the settings of the main language are valid. In this case the font definition is displayed bright gray and next to it the flag of the main language is displayed, cp. following figure.

Since a special font for the fontalias *Standard* in the language Germany was not yet defined, the USA-flag is displayed:

	Fontalias	Font	
1	Standard	Microsoft Sans Serif 8	USA

Figure 17: Language-specific fonts

In the next step you will create an own font definition for the language Germany .

Change to the section *Fonts* and lock it.



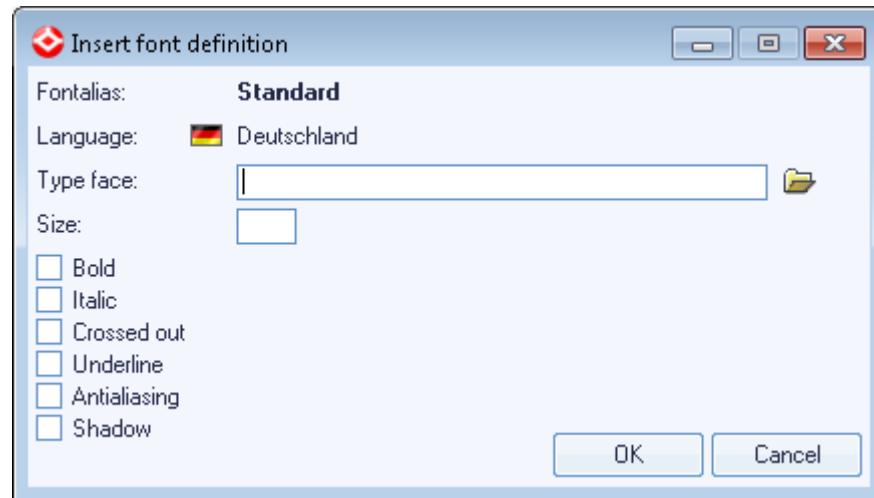
The screenshot shows the 'Fontalias' configuration dialog. On the left, there's a tree view with 'Fontalias' expanded, showing 'Standard' selected. The main area has a table with columns: Language, Currency, Font, and Size. There are two rows: Row 1 shows 'United States' with 'USD' currency, 'Microsoft Sans Serif' font, and size '8'; Row 2 shows 'Germany' with the same currency and font, but size '8' is highlighted with a red box. The 'Name:' field above the table contains 'Standard'.

The icon of the USA-flag and the gray font indicate that no own font definition for the fontalias *Standard* for the language Germany is present.

Flag the line for the language Germany and select the menu item *New...* from the context menu.



The dialog *Create allocation font to fontalias* is opened.



Click on the icon of the field *Type face* in order to select any font. Keep the presettings for *Font* and *Font size* and set the *Font style* to *Italic* (possibly also called *Oblique*) and acknowledge your selection with *OK*.

The screenshot displays two windows side-by-side. On the left is the 'Font' dialog, which includes:

- Font dropdown: Microsoft Sans Serif (selected)
- Font style dropdown: Oblique (selected)
- Size dropdown: 8 (selected)

On the right is the 'Insert font definition' dialog box, which includes:

- Fontalias: Standard
- Language: Deutschland
- Type face: Microsoft Sans Serif (selected, matching the 'Font' dialog)
- Size: 8 (selected, matching the 'Font' dialog)
- Font style checkboxes:
 - Italic (selected)
 - Bold
 - Crossed out
 - Underline
 - Antialiasing
 - Shadow
- Buttons: OK and Cancel

Figure 18: Extending fontalias by secondary language

Close the dialog *Create allocation font to fontalias* via a click on OK.



The USA-flag is no longer displayed. Through this can be seen that an own font definition for Germany is present.

		Language	Currency	Font	Size	Styles
1		1 United States		Microsoft Sans Serif	8	
2		49 Deutschland		Microsoft Sans Serif	8	Italic

Close the frame via the button *Release all and close*.



Therefore the main language United States and the secondary language Germany are available in all knowledge bases that use the *Training frame* (momentarily only the knowledge base *Training example*).

8.3. Exercise: Migrating the language

In order to make the effects of the previous changes visible, the language switching is implemented in the next step.

To do so, two pushbuttons are needed, one for German and one for English.

Open the class *start*. Open the *Main form* of the class *start*. Create a pushbutton via the context menu item *New -> Pushbutton*. Open the form preview via the icon and position the pushbutton on the form.



The naming of the pushbutton is defined in the line *Text* below the node *Texts*.

Enter "English" in the field *Text* and exit the entry field (tab, return, click) in order to create a Textelement with this text.

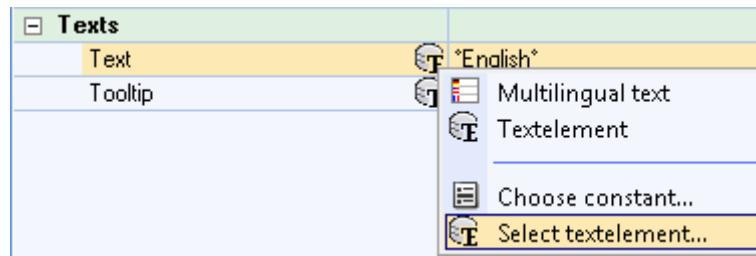


Name	Pushbutton
Type	Pushbutton
Help	
Stylesheet	
Geometry	
Representation	
Handling	
Texts	
Text	^English^
Tooltip	

In order to translate the English contents of the Textelement into the German language, the Textmanager has to be opened. The Textmanager is the administration environment for the Textelements.



Click on the icon in the second column of the line *Text* and select the entry *Select Textelements...* from the appearing menu.



The Textmanager opens. The table in the middle shows all previously created Textelements. The Textelement "English" is flagged. In the right section you can see the contents of the flagged Textelement and here you can also edit it.

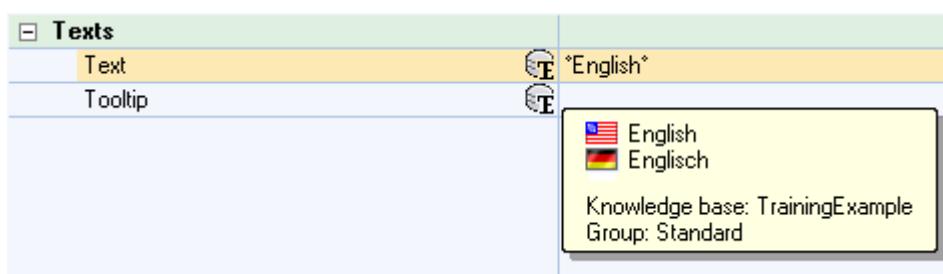


Click in the line with the German flag and enter the translation: Englisch.



Figure 19: Translating Textelement in the Textmanager

Close the Textmanager via double click on the Textelement, via Escape or via the X in the title bar. If the mouse is moved over a Textelement, the tooltip displays the contents of the Textelement in all languages.



If the user clicks on the pushbutton, the language has to be changed. You already got acquainted with the result “Click” (= selection) in chapter 7.4.1 in which a car object was created via a click on a graphic.

In the current context the implementation of the action (= language migration) is also carried out in the selection trigger of the form element. To do so, the call of the function *LanguageDialogSet()* is deposited on the tab page *Selection* of the form element Pushbutton.

The dialog language can be changed during runtime via the function *LanguageDialogSet(Country code)*. The parameter *Country code* defines the language that has to be used.

Write the following function call to the editor of the tab page *Selection* of the pushbutton:

```
LanguageDialogSet(1);
```



With this the dialog language is set to English via a click on the pushbutton.

Create a further pushbutton and enter “German” in the column *Text*. Then you open the Textmanager and translate the just created Textelement.

On the tab page *Selection* you deposit the procedure code

```
LanguageDialogSet(49);
```



Start your application and test the language migration. The change can be seen on the setting of the italic fontaslias.

English

German

Englisch

Deutsch

A translation however is only carried out for the pushbuttons. The reason is that no namings in German were deposited for the other objects, components etc.

Generally valid: If the naming is not defined in the currently active dialog language, only the naming of the main language is displayed.



The translation of the classes, components etc. is carried out in the next step.

8.4. What is a naming and what is a name?

The name of an element serves for the unique identification. The relevant element can be addressed via this name.

Namings however only serve for the display, e.g. on the form. The same namings can be used as often as you like.

Definition

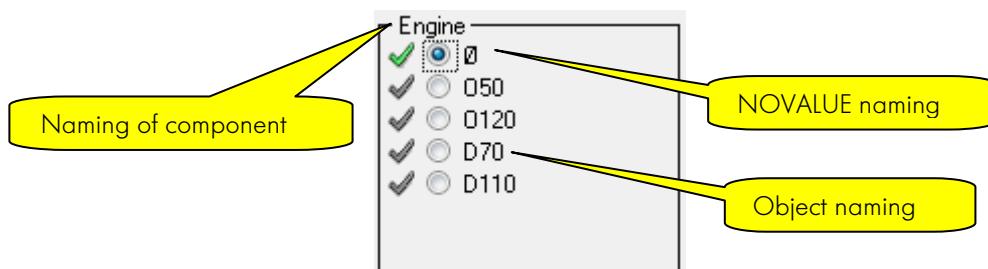
Name	MainForm	Name for unique identification
Type	Form	
Help		
Stylesheet		
+ Geometry		
+ Representation		Header of the form
+ Handling		
- Texts		
Header	Car Configuration	

Figure 20: Difference between name and naming

Example: The class name has to be unique in the knowledge base. The class however can be allocated with a naming, because often class names are not meaningful and especially not multilingual.

8.5. Maintaining namings

On the form are configuration boxes that represent different components (= modules) of the car. Let us look at the configbox (here in the list display) for the engine. Is the representation satisfactory? How can be proceeded in order to translate the individual namings?



8.5.1. Object naming / Namings on classes

Often the names of classes are not suitable for the display on a form, e.g. the user cannot recognize that "O50" is a 50 HP Otto engine. Therefore it is useful to define a meaningful (multilingual) naming.

The object naming is defined in the class editor of the corresponding class.

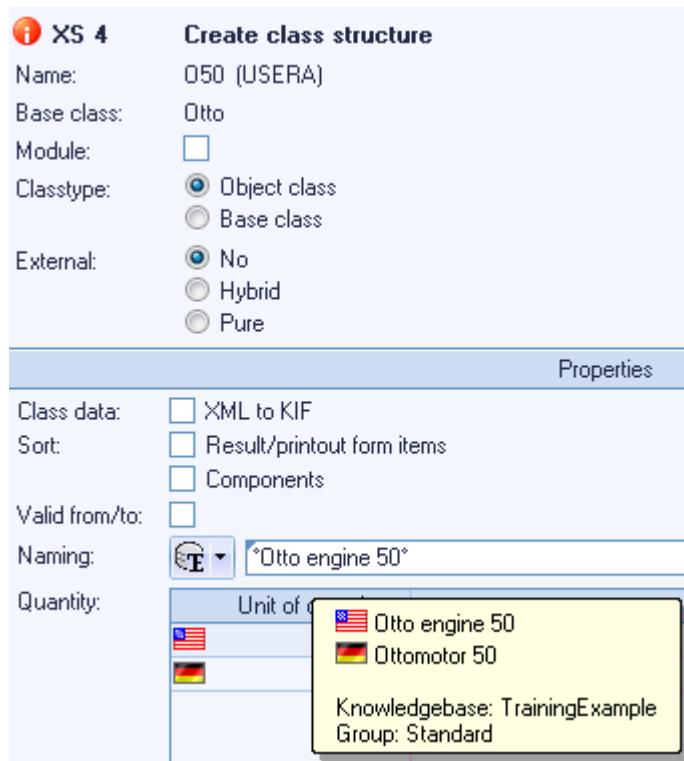
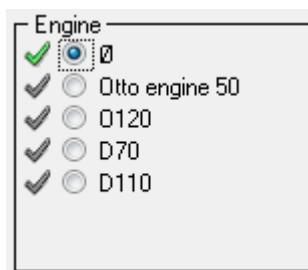


Figure 21: Definition of the object naming in the class editor

Now the naming instead of the class name is displayed during runtime:



8.5.2. NOVALUE naming on Components / Features

If a component or a feature is not allocated with a value, the value *NOVALUE* is displayed with the symbol \emptyset . The symbol stands for „no selection“. The symbol \emptyset can be replaced by a meaningful text via the definition of a *NOVALUE* naming.

The *NOVALUE* naming is defined in the editor of the component or of the feature that is displayed in the configbox. In order to deposit namings, the Wasele first has to be indicated as translation-relevant!



In order to change the NOVALUE naming of the component `_Engine`, you open the class `Car`. In this class you open the component `_Engine` via a double click in the structure tree.

Enable the option *Translate relevant*. Through this the fields *Naming* and *NOVALUE* become visible. In the field *NOVALUE* you enter the text that has to be displayed instead of the `\` symbol, e.g. „please select...“. Exit the entry field via the Tab key in order to create the Textelement. Then you open the Textmanager and enter the translation of the NOVALUE naming.

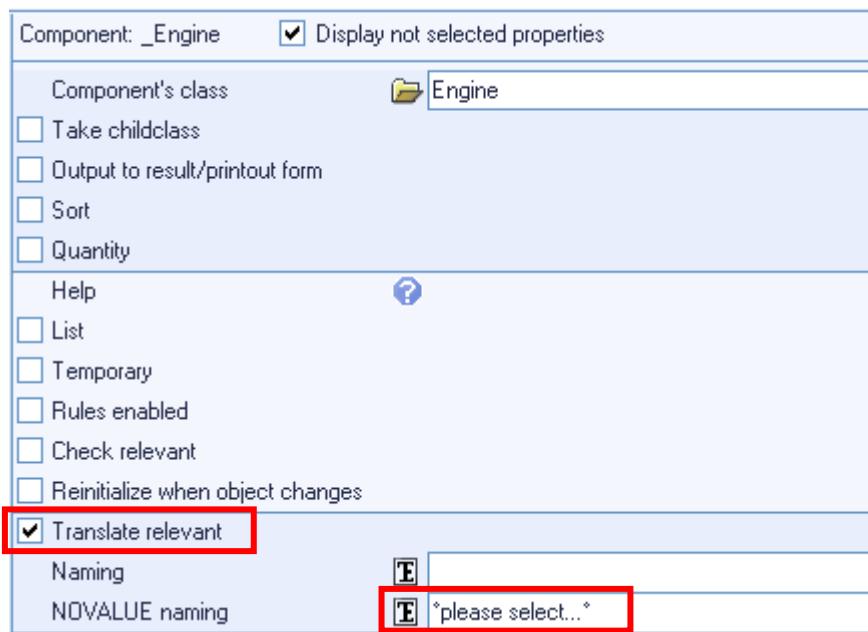
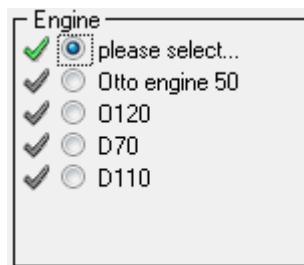


Figure 22: Changing the NOVALUE symbol

The change has the following effect in the interpreter:



8.5.3. Namings on components and features

By default the name of the cause variable (feature or component) is displayed in the title of a configbox. In order to change the heading of a configbox, a different naming can be deposited.

Namings on components and features are edited at the same place as the NOVALUE naming. Prerequisite is also that the switch *Translation-relevant* is enabled in the editor of the component or of the feature.

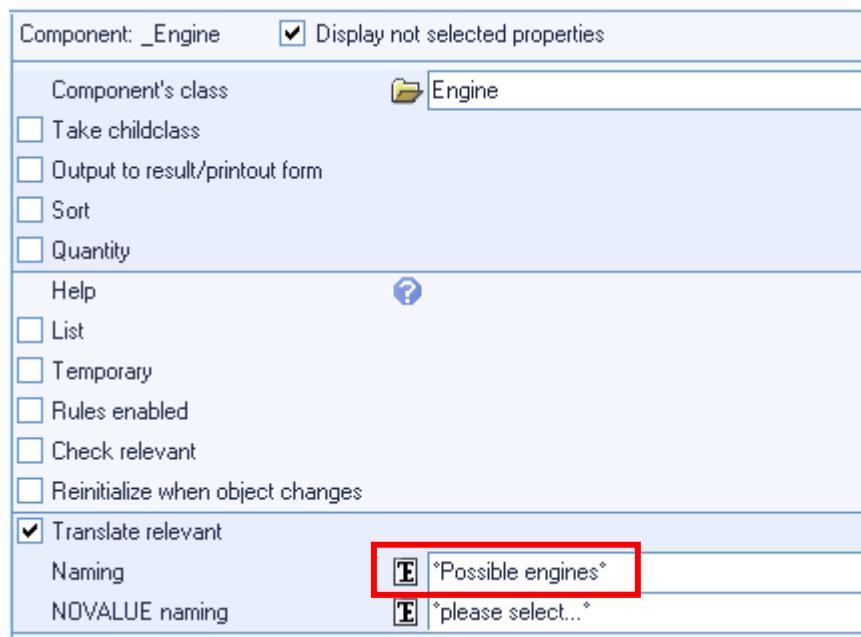
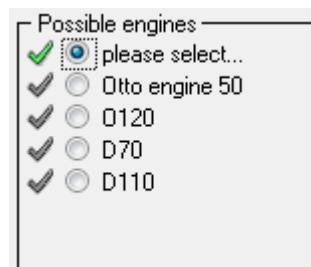


Figure 23: Defining component naming

This has the following effect in the interpreter:



Create object namings for all object classes (car models and modules) and translate the namings.



Define the Textelement °please select...° as NOVALUE naming for all components that are displayed on the form.

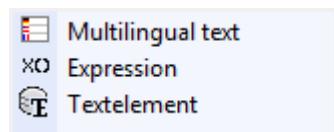
8.6. Tips & Tricks

8.6.1. Definition of multilingual texts

With the set up of a knowledge base should be planned from the start that the application could become multilingual at a later time.

The consistent use of Textelements for namings, error messages, offer texts etc. has the advantage that anytime and without big effort languages can be added later in the frame and that the Textelements can be translated conveniently in the Textmanager.

Altogether three possibilities are available to deposit texts. The following selection menu can be found at each place in camos.Develop where namings can be defined:



 In the mode *Expression* the text that has to be displayed can be composed dynamically via simple expressions and functions. At some places the mode *Expression* is not supported and instead only the mode *Constant* is offered.

The mode *Expression* also comprises the support of constants.

It is principally possible to work with a mixture of multilingual constants, Textelements and free text entry in a knowledge base. For reasons of conformity you should however decide on one procedure.

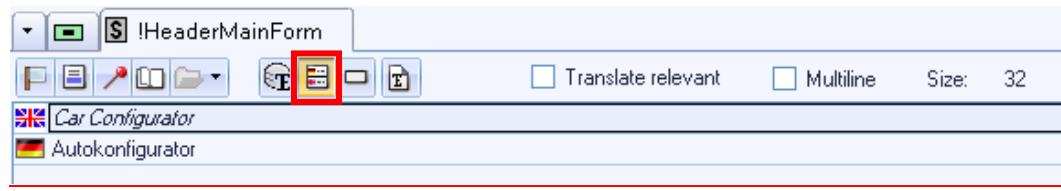
Constant

The naming is deposited in a string constant. The text that the string constant should get can be defined in three different ways.

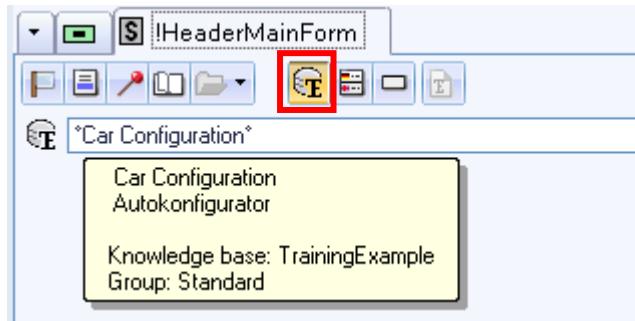
- monolingual



- multilingual



- Textelement



Then the constant is used as the naming. If the text that has to be displayed changes or if a new language was added that has to be translated, this adaptation is carried out in the constant editor.

Texts	
	!HeaderMainForm
Header	!

Multilingual text

The definition of the (multilingual) naming is carried out in a table. The table contains a line for each language that is defined in the frame.

Texts	
Header	Car configurator
Translate relevant	
United States	Car configurator
Germany	Autokonfigurator

This procedure to deposit texts is the most unfavorable and error-prone procedure. In the first place the text cannot be reused but is only valid at the place where it was entered.

If a language is added, the table is actually extended by a corresponding line. The developer however has to find, open and adapt all places on which the text has to be changed. Here it can happen very fast that a few places are overlooked or typing errors slip in with entering the new text.

Due to this and other disadvantages camos recommends the use of Textelements. More information for the usage possibilities of the Textmanager and the Textelements can be found in chapter 31.

8.6.2. Display options in the form element Component tree

The form element Component tree offers several display possibilities for namings. If a new component tree is created, by default the class name of the objects that exist in the component tree is displayed because this name is present in any case.

There are two further display options in addition to the option *Class name*. These options are defined in the properties of the *First. component tree column*.

- *Component naming* shows the naming of the component from which the respective object was created (see chapter 8.5.3).
- *Object naming* shows the object naming that is deposited in the class editor (see chapter 8.5.1).

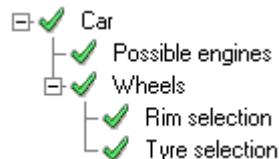
The three options can be combined with each other as you like. The further selection is added in the display to the previous name in the form “Component=Class Object naming”.

Texts	
Class name	<input checked="" type="checkbox"/>
Component naming	<input type="checkbox"/>
Object naming	<input type="checkbox"/>
Naming expression	<input type="checkbox"/>
Naming class	<input type="checkbox"/>
Header	<input type="checkbox"/> *Modules*
Header connect with the next column	<input type="checkbox"/>
Tooltip	<input type="checkbox"/>

Object naming:



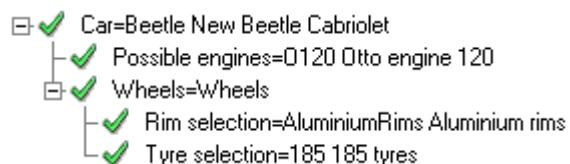
Component naming:



Class name:



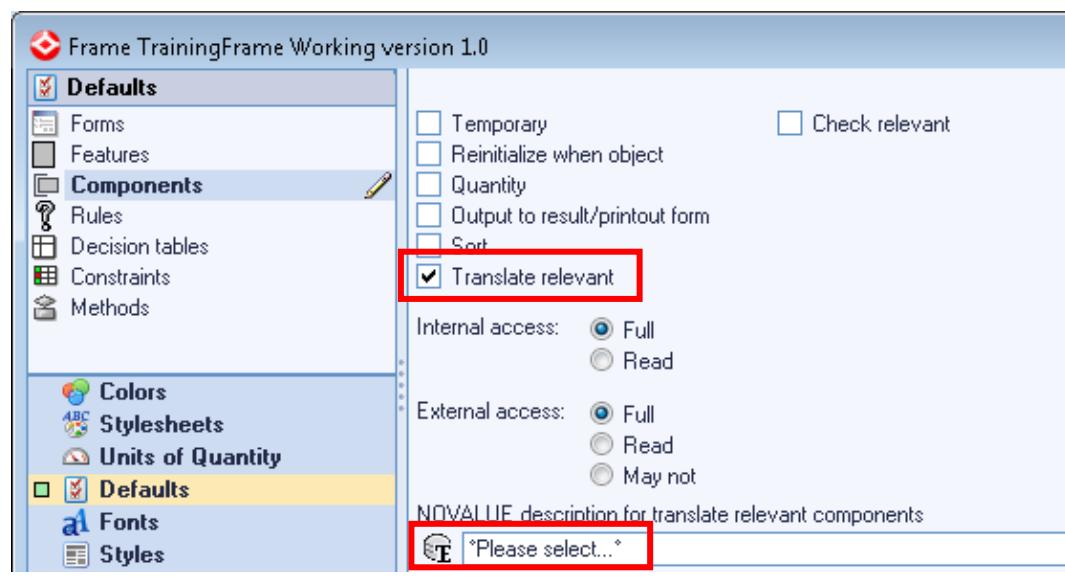
All three options enabled:



8.6.3. Presetting of the NOVALUE naming

For components and features an init value of the NOVALUE naming can be made in the frame in the section *Presetting*. This naming is allocated for all newly created components or features, it can however be changed manually.

The naming can be deposited as multilingual text or as Textelement. The migration of the mode is carried out via the context menu of the table.



The presetting for the NOVALUE naming is only effective if the option *Translation-relevant* is enabled!



If a Textelement has to be used as NOVALUE naming, this Textelement has to be allocated to the section *Public* in the Textmanager. Knowledge base-specific Textelements are not allowed, because the frame can also be used for other knowledge bases than the *Training example* and in this case the access to Textelements of the knowledge base *Training example* is not possible.



The section *Public* contains Textelements that are used knowledge base-overlapping, e.g. New, Delete, OK, Cancel, etc. The section *Training example 1.0* however contains only texts that are needed in connection with the autoconfigurator.

The separation of the sections for knowledge base-specific and generally valid texts should offer the developer a better overview and it should also prevent the creating of several Textelements with identical contents.

In order to use the already existing Textelement °please select...° as NOVALUE naming for future components and features, it first has to be moved from the section *Training example 1.0* to the section *Public*.



Open the Textmanager via the icon in the toolbar of camos.Develop and flag the Textelement °please select...° in the middle table. Call the context menu item *Move*. In the dialog *Move* you select the group *Standard* in *Public* and click on *OK*.

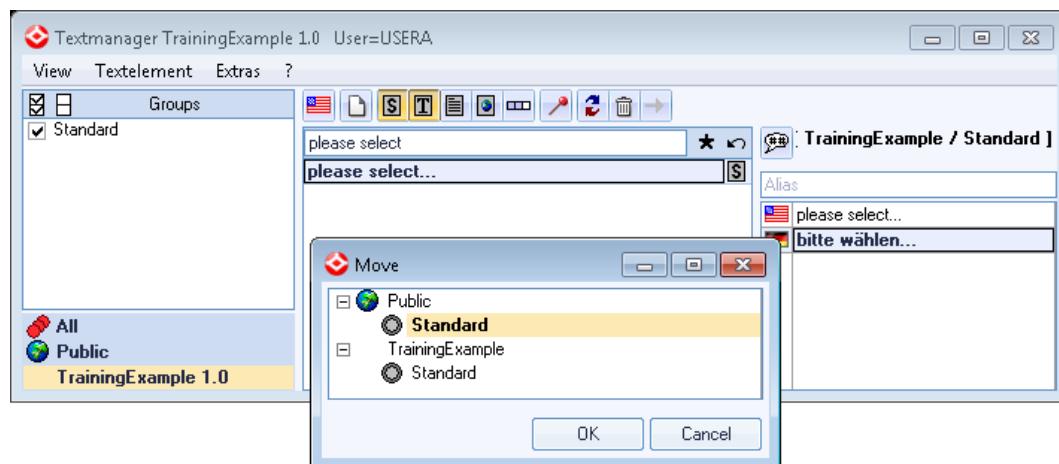


Figure 24: Moving Textelements to other sections

The Textelement °please select...° is now displayed in the section *Public*. It can now be selected as init value for the NOVALUE naming.

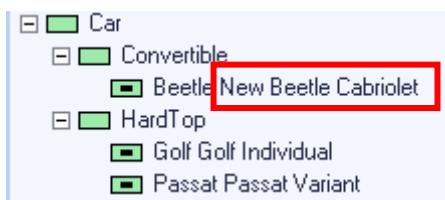


Open the *Training frame*, lock the section *Presetting* and flag the element *Components*. From the context menu of the language table you select the entry *Select Textelement...* and assign the Textelement °please select...°.

Don't forget to enable the property *Translation-relevant*.

8.6.4. Display of class- and component namings

Via the function *Display namings* in the context menu of the knowledge base root the namings of classes that are defined in the class editor can be added directly to the display in the class tree.



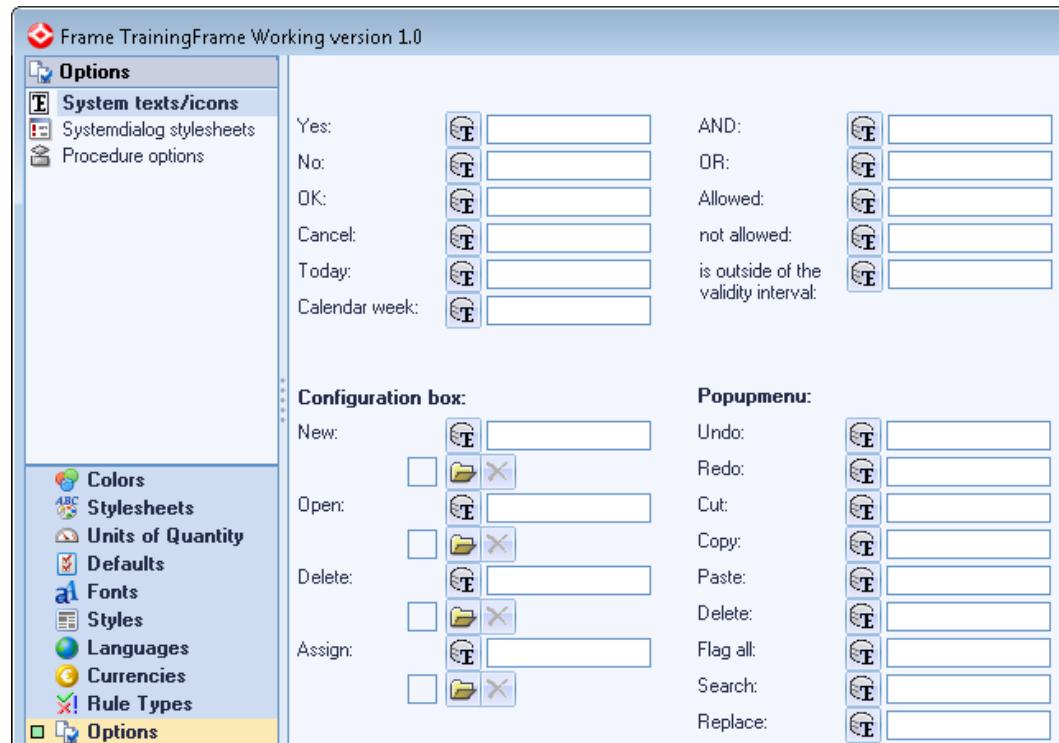
The same is valid for the structure tree. Here the component-, feature- and object namings can be displayed via the menu item *Show namings* in the context menu of the structure tree root. An additional column for the display of the init value can also be faded in. To do so, the menu item *Show init value* in the same menu is enabled.

Car		Naming	Initialization
<input type="checkbox"/>	Results		
<input type="checkbox"/>	Printout forms		
<input type="checkbox"/>	Trigger		
<input type="checkbox"/>	Classification trigger		
<input checked="" type="checkbox"/>	Standard		
<input type="checkbox"/>	_Engine	Possible engines	NOVALUE
<input checked="" type="checkbox"/>	050	Otto engine 50	
<input checked="" type="checkbox"/>	0120	Otto engine 120	
<input checked="" type="checkbox"/>	D70	Diesel engine 70	
<input checked="" type="checkbox"/>	D110	Diesel engine 110	
<input type="checkbox"/>	_Wheels	Wheels	'Wheels'

These settings are only valid for the currently opened class. If you want to display the namings and/or contents of an init value permanently, you can save this view. To do so, you will find the menu item *Set current view for all classes with the opening* in the context menu of the structure tree root.

8.6.5. System texts for system dialogs and form elements

For system dialogs and certain form elements Textelements can be deposited in the frame in the section *Options* on the page *System texts*. So e.g. the text of the pushbuttons Yes, No, OK and Cancel in WinMessages can be affected. This can further be used in the form elements Calendar and Rule texts.



8.7. Repetition

- Languages are administrated in the frame.
- The fontalias *Standard* has to be defined for each language.
- The object naming is deposited in the class editor.
- The component naming is deposited in the component editor.
- The multilingualism on components and features is determined by the option *Translation-relevant*.
- The NOVALUE naming is deposited in the editor of the component or of the feature.

9. Switching the language via the menu bar

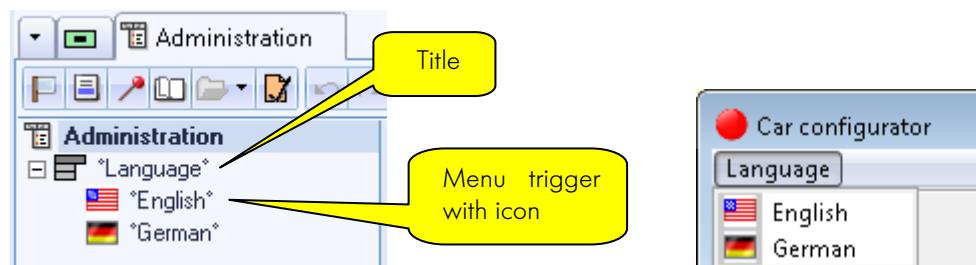
9.1. Intention

In order to optimize the surface of the form, the pushbuttons for switching the language will now be replaced by a **menu** with two menu items.

Definition

A menu is a control element that can be allocated to a form (so-called form- or main menu) or a form element (as popup- or context menu).

Menus are wasele and therefore they can be found in the wasele list with the icon  A menu can consist of several menu elements. The most frequent items are titles, separators and menu trigger.

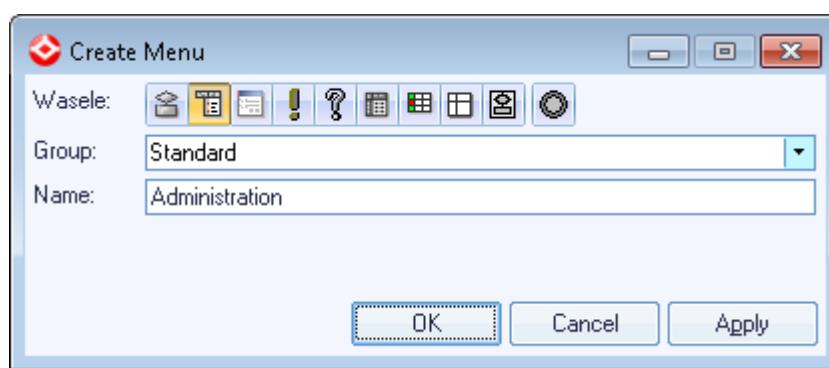


Menus are – analog to forms – built up in a tree structure.

9.2. Practice: Create a menu

Open the class start. Click on the icon  in the toolbar of the wasele list.

Select the wasele type Menu, enter the name “Administration” and confirm with OK.



A new menu element can be created via the context menu of the menu root.

Open the context menu of the menu root of *Administration* and select *New title*.



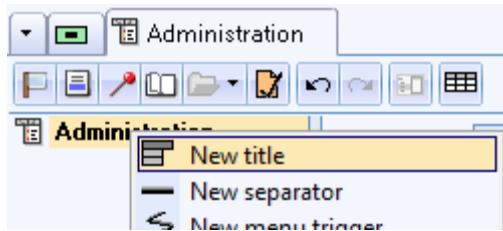
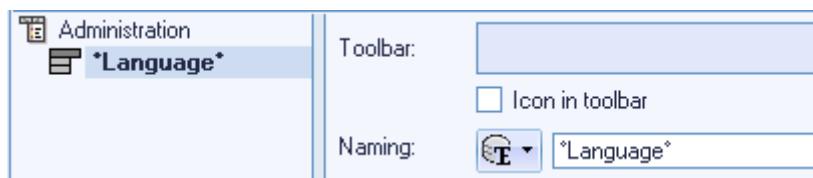


Figure 25: Menu editor - Create new elements

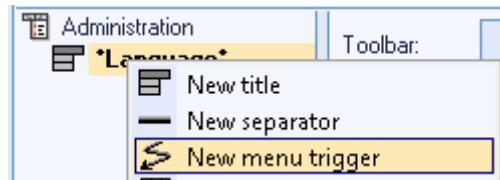
Allocate the English naming “Language” to the menu title. Define “Sprache” as German translation for the Textelement.



The menu element *Title* serves as to structure functions similar to the standard menu titles *File*, *Edit* or *Extras* of every Windows application under which menu items for different actions (*New*, *Open*, *Save* etc.) can be found.

Under a title you can create any amount of further menu elements.

In order to create a menu item under the title, you select the title *Language* and carry out the context menu item *New menu trigger*. A new menu item is created.



The naming of the menu should be “English”. For this text, you already have applied a Textelement in Chapter 8.3. To use that textelement, open the Textmanager, typing “English” in the line *naming* and assign the desired textelement via double click or via the icon → from the toolbar.

Or type “English” in the line *naming*, leave the field via TAB and the Textelement °English° is automatically recognized and assumed.

Tip: If more than one Textelement begins with "English", all matching Textelements are shown and you can choose the correct Textelement by double click.

Create the second menu item German in the same manner.



9.2.1. Integrate graphics as menu icons

Menu triggers can be furnished with icons which are displayed left next to the name. For the languages *English* and *German* the respectively matching country flag has to be displayed.

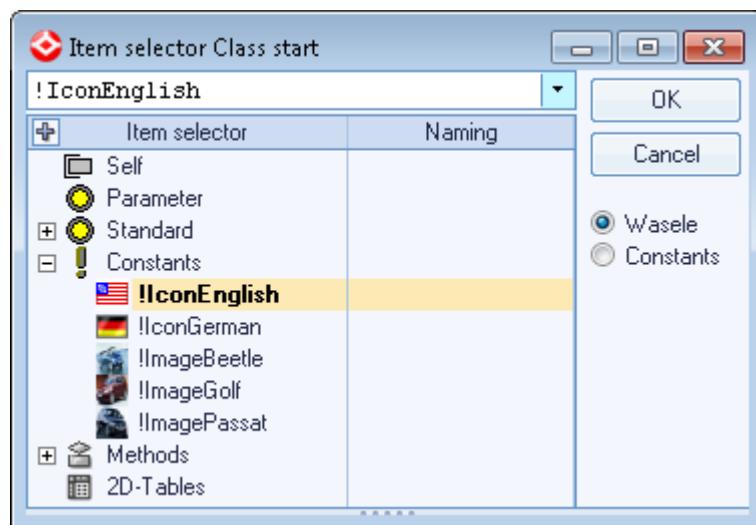
To do so, two graphic constants are created. This is carried out analog to the graphics of the product images, see practice in chapter 7.2.2.1. On the training CD you will find the graphics *Flag49.bmp* for Germany and *Flag01.bmp* for United States in the folder *Graphics*.

Create a new constant of the type Graphic with the name *!IconEnglish* and load the graphic *Flag1.bmp*. Create a further constant with the name *!IconGerman* and insert the image *Flag49.bmp*.



The created graphic constants can now be assigned to the menu items. To do so, the menu *Administration* and the menu item *English* is opened.

Open the menu *Administration* and assign the graphic constants to the field *Icon* of the menu items *German* and *English*. Drag the constant per D&D from the Wasele-list into the field *Icon* or call the Wasele selection dialog from the context menu of the field *Icon* and select here the constant.



Via setting the option *Icon in toolbar* the country icon is additionally displayed in a toolbar below the menubar. This option is only relevant if the menu has to be used as form menu or in the form element *Toolbar*. If the menu is used as context menu, the icons of menu items are always displayed.

Activate the option *Icon in toolbar* for both menu items.



9.2.2. Assign actions to the menu items

The action that should be carried out when the user clicks on the menu item is entered in the procedure editor of the menu trigger: *LanguageDialogSet(49)* changes the current dialog language to German.



Add the procedure code to the menu items *German* (country code 49) and *English* (country code 1).

Now the menu *Administration* looks like this:

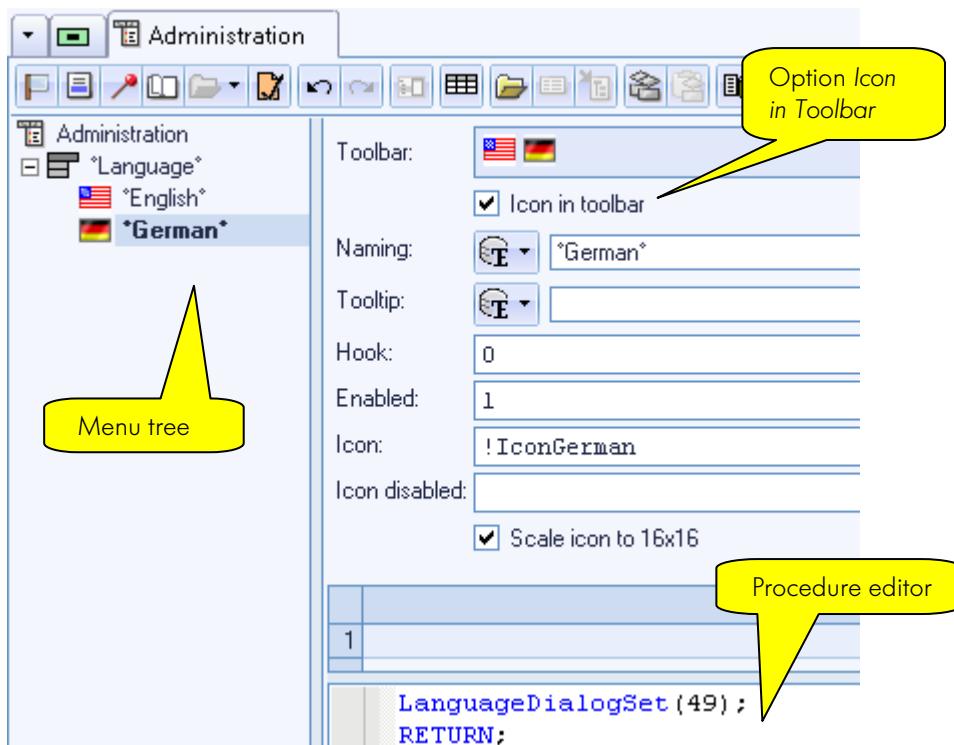


Figure 26: Menu editor – Properties of a menu item

9.2.3. Allocate the menu to the form

Since the menu has to be used as main menu on the form, the menu *Administration* has to be allocated to the form *MainForm*.

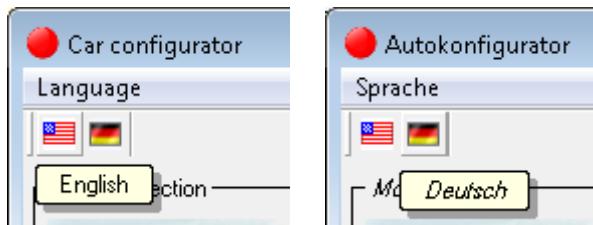


To do so, you open the class *start* and the *MainForm* from the wasele list. Select the form root.

Under the structure item *Handling* you will find the field *Menu*. Via a click on the icon the item selector opens. Select the menu *Administration* and confirm the selection with *OK*.

Name	MainForm
Type	Form
Help	
Stylesheet	
+ Geometry	
+ Representation	
- Handling	
Readonly	<input type="checkbox"/>
In taskbar	<input checked="" type="checkbox"/>
Menu	Administration
Toolbar	<input checked="" type="checkbox"/>
Large buttons	<input type="checkbox"/>
ViewShot relevant	<input checked="" type="checkbox"/>

Start your application and test the switching of the language.



Delete the two pushbuttons from practice 8.3 because they are now replaced by the menu.



9.3. Tips & Tricks

9.3.1. How to activate and deactivate menu items

In the field *Enabled* can be set via an expression if/when the menu element is enabled or not. By default a trigger is always enabled; the field contains the value 1.

Enabled:

Example: The menu item for the dialog language *German* should be enabled only if the current dialog language is not German. The same thing applies to the menu item *English*, it should only be enabled if the dialog language is not yet English.

The following expression automatically activates or deactivates the menu item *German* according to the current language:

LanguageDialogGet() <> 49

Correspondingly on the menu item *English*:

LanguageDialogGet() <> 1

LanguageDialogGet() provides the country code of the current dialog language. If this code is different to 49, the menu item *German* is enabled; otherwise (the current dialog language is already German) German cannot be selected again.



The expression may not be concluded with a semicolon! This is valid for all positions in camos Develop at which expressions can be entered.

Always carry out a syntax check for control reasons!



9.3.2. How to indicate a menu item as being selected

The entry field *Hook* indicates if a selected menu item is marked with a hook symbol. So a Yes/No condition can be visualized.

Hook:

In the field an expression, a function etc. can be entered that returns 0 or 1. 0 means “no hook”, 1 means “display hook”.

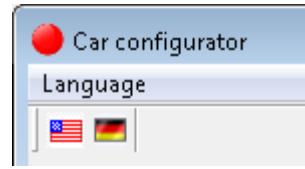
Example: The active dialog language has to be displayed via a hook. To do so, the following expression is entered in the field *Hook*:

`LanguageDialogGet() = 1`

If the dialog language is English, the icon of the menu item *English* is changed to a hook:



If the menu item has an icon that is displayed in the toolbar, it is displayed as follows if the hook expression has the result 1:



9.3.3. Icon for a deactivated menu item

Icons that represent the inactive condition can be optionally entered on titles and menu items. To do so, you enter a graphic constant in the field *Icon disabled*.

Icon disabled: `!IconEnglishDisabled`

If no graphic is entered in the field *Icon disabled*, the icon is automatically greyed out.

9.4. Repetition

- Menus are arranged in a tree structure.
- Icons of menu items can be additionally displayed in a toolbar.
- Menus need other forms or form elements to which they are allocated.
- Different icons can be entered for activated and deactivated titles and menu items.
- The setting or query the current dialog language is done with the function *LanguageDialogSet(CountryCode)* or *LanguageDialogGet()*.

10. Pricing

10.1. Intention

Currently we differentiate the individual modules only via their names, e.g. *SteelRims* or *Alloy-Rims*. In reality, a rim is specified by a variety of properties, it is e.g. differentiated from other rims: material, size and color.

Not all of these properties are of interest for the user. A potential buyer is e.g. barely interested in the manufacturer number and how many screws were needed to construct the rim. The price for the modules, however, should be directly shown in the application without the condition that the component has to be previously selected.

In the following chapter you will learn how to quickly allocate all modules with an individual price. For that purpose you will be taught what features are, where they are defined and how they are used. Since this is price information, you will also learn how to add a currency to the knowledge base and how the prices of the individual components are displayed on the form.

10.2. Main- and secondary currencies

Before you can enter the price information in the knowledge base, you have to define on which currency unit the amounts refer to. To do so, you enter the currency US Dollars in the frame.

Currencies are created in the frame. Analog to languages they refer to a certain country (country code). However, language and currency don't depend on each other, i.e. the language Swedish can be displayed and used together with the currency Russian Ruble.

A main currency and any other secondary currencies can be defined in the same way as main- or secondary languages. The first created currency is always the main currency, but later it can be replaced by a secondary currency.

By default, the main currency is used for the calculation and display of all currency features. Therefore at least one currency has to exist in the frame if you want to work with currency features and/or constants in the knowledge base.

Definition

10.2.1. Practice: Create the currency US Dollar in the frame

Open the frame *Training_Frame* via the icon  from the toolbar above the class tree. Then you switch to the area *Currencies* and reserve it. Via the context menu item *New* in the upper left table the dialog *New Currency* is opened.



First the referring country of the currency has to be selected from the combobox *Countries selection*.

Select the entry *United States of America / Vereinigte Staaten von Amerika* in the field *Countries selection*. After the country code 1 was automatically entered in the field *Code*, confirm with *OK* and then apply the currency *USD* by confirming the dialog *New Currency* with *OK*.





 Click on the pushbutton  *Init value from system* in order to read in the init value of the icon, the currency, the ISO description, the currency symbol and the decimal places.

Code:	001			
Icon:	 			
Currency:	<table border="1" style="width: 100px; height: 100px;"> <tr><td> USD</td></tr> <tr><td></td></tr> <tr><td> </td></tr> </table>	 USD		
 USD				
				
International currency description				
ISO:	USD			
Currency symbol:	\$			
Decimal digits:	2			
Decimal separator:	.			
1000 separator:	<input checked="" type="radio"/>  <input type="radio"/>  <input type="radio"/> Space			
Factor:	1			
Currency symbol for the display on the form				
Number of digits after the comma				
Conversion factor with regard to the main currency				

Figure 27: Creating a new currency in the frame

During runtime the currency can be switched via the function `SetCurrency(CountryCode)`. Via a `factor` that is entered for secondary currencies an automatic exchange conversion with regard to the main currency is carried out.

 Confirm the changes in the frame with *Release all and close*.

10.3. Features

In camos Develop properties of objects are displayed via **features**.

A feature is used to describe an object in detail. Features can contain different data types: Numerical, string, RTF, date, currency, graphic, binary, object pointer, ActiveX-pointer, MPF-Pointer, HTML and bit.

Contrary to constants (see chapter 7.2.1) the value of a feature can be changed during runtime.

Features are created in the structure tree. Certain settings can be made in the editor of an opened feature, see chapter 10.3.1.

Workbench/Wasele/Features/Properties

10.3.1. Practice: Create the feature Price

All modules should have the property *Price*. Here the inheritance hierarchy can be used with creating the feature in the class *Modules*; the feature will be inherited to all derived classes.

Open the class *Modules* in the knowledge base *TrainingExample*



Features are created in the structure tree via the context menu of a group.

Select the group *Standard* in the structure tree and carry out the context menu item -> *New feature...*. The dialog *New feature* is opened.

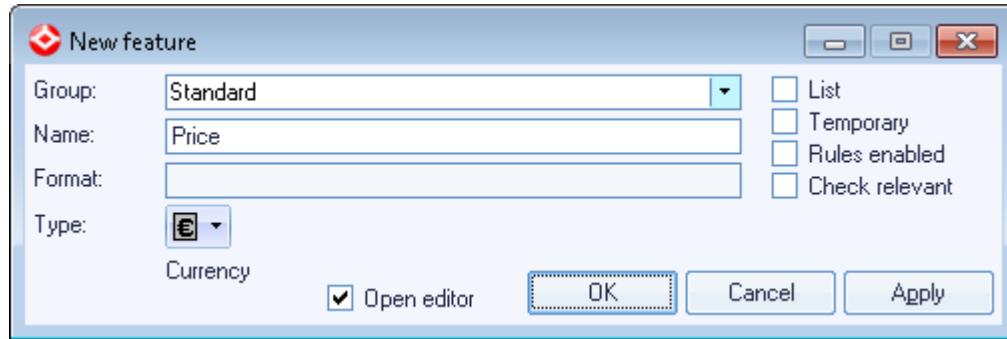


Figure 28: Creating a new feature

Beside the name and the data type, features have a lot more properties and options. Four of them are offered in the dialog to create a feature; further properties can be found in the editor of the feature.

- *List* specifies whether the feature can take up only one value or several values. If *List* is enabled, the name of the feature is automatically supplemented by the list brackets [], e.g. list feature *Username[]*. Features that are not list features are also called “scalars”. Attention: Lists begin with index 1 in camos Develop!
- The switchbox *Temporary* has to be set for features with a temporary character. It is e.g. useful to define count variables, form handles etc. as temporary features.

- The option *Rules enabled* is to be set if the feature should be included in the rule processing (see chapter 15). Ruled features are imaged via a turquoise icon in the structure tree, e.g. .
- The option *Check relevant* is needed, when a feature is used in a condition of a rule. In order to increase the performance of the application, this option should be set explicitly, so that the rule validation checks only the marked features.

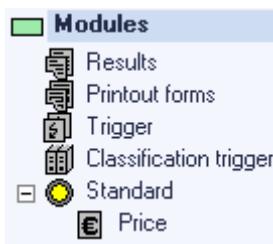
 [Workbench/Structure Tree/Features/Create feature](#)

 [Workbench/Structure Tree/Features/Properties](#)



Allocate the name “Price” and activate the type *Currency*. If you enable the switchbox *Open editor*, the feature editor is directly opened with the creation. Otherwise you have to double click the feature in order to open it. Click on *OK* to create the feature.

The currency feature *Price* is added under the group *Standard* in the structure tree.



10.3.2. The feature editor

The editor of a feature consists of three sections: the list of the possible values, the *init value* field, and a set of tab pages, e.g. *Properties*.

10.3.2.1. Feature values

A **feature value** is a possible value which can be allocated to the feature. Contrary to components whose values are predefined (= child classes of the component class), feature values have to be defined manually. For each feature any amount of values can be defined.

Definition

Values that have to be ruled or offered as a possible selection in a configuration box, have to be applied from the list of the possible values to the structure tree via a double click. There they are displayed with the icon .

Two special values are predefined: NOVALUE and ANYVALUE.

NOVALUE means that the feature has no value. ANYVALUE means that the feature has any value. These values are used with the ruling of features. So the value NOVALUE can be e.g. forbidden for a feature for which always a selection has to be made.

10.3.2.2. Init values of features

An **init value** is a value that is automatically allocated to the feature with the instantiation of the object, even before the method *New* is carried out.

For scalar features only one init value can be defined. If the feature is a list any amount of init values can be allocated; the same value can occur several times, too..

There are two ways to define an init value:

1. The context menu item *Set as initial value* is carried out on an existing value in the *value list*.
2. The context menu item *New* is called in the *Init value* field. The here defined value is automatically applied as a possible feature value in the value list.

With the creation of an init value you have the following types:

Constant

Via *Constant* you can define the content of a constant as init value.

"ab" Static value

Static value is any value. It has to match the data type of the feature.

XO Expression

Via *Expression* you can determine the value of the init value e.g. via methods. If e.g. the expression $5+4$ is defined, this expression is calculated and the result is assigned as init value.

The new initialization is only written in the list of values if the option *Add to values* in the dialog *Input of value* is set.

The option can be disabled if you don't need the initial value in the value list



10.3.3. Practice: Set the init value

Since the price feature is inherited to all child classes of *Modules*, it is useful to define an init value. So the later added components inherit a valid price value.

Here *valid* means that addition errors can occur within the calculation of the total price (see chapter 11) if a component has to be added whose price is set to NOVALUE.

Assign the init value 0 (= 0.00 USD) to the feature Price.

To do so, open the editor of the feature *Price* via a double click. In the section *Initialization* you carry out the context menu item *New*.

Activate the mode *Static value* and enter 0 as init value; during runtime the decimal places are automatically filled according to the format definition of the currency USD. Disable the check mark in the option *Add to values* and confirm with *OK*.



With this init value in all module classes the feature *Price* gets the value 0 if an object is created from them.

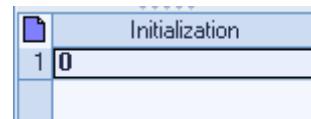
10.3.4. Practice: Overloading the feature Price

The init value of a feature can be changed in derived classes via a so-called **overload**.



Open the class *O50* in the knowledge base *TrainingExample*. Open the feature *Price* via a double click.

All inherited properties that can be overloaded are indicated by a blue sheet symbol.



The inherited init value is 0. The field for the init value cannot be edited. To do so, you first have to overload the init value. This is carried out via click on the icon or on the context menu item *Overload*. Overloads are displayed with an orange-colored sheet symbol .



Click on the blue sheet symbol of the Init value section and select the context menu item *Open* to change the init value 0.

Enter any price for the engine O50, e.g. 350, and confirm the dialog *Input of Value* with *OK*.

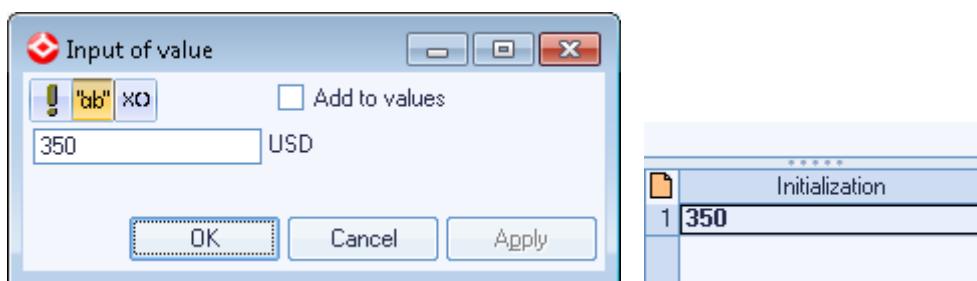
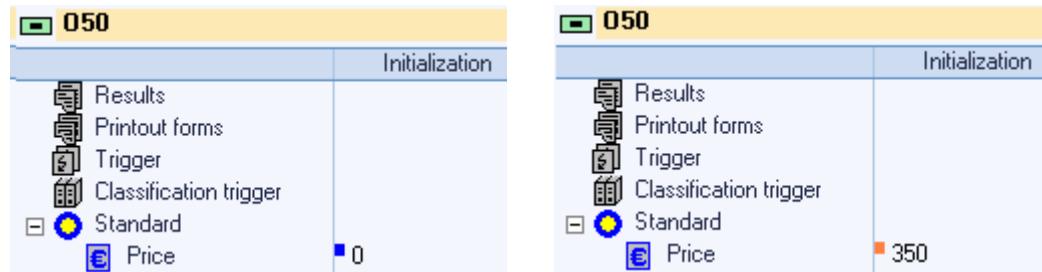


Figure 29: Feature editor - Overloading the init value

If you have activated the display option *display initialization* in the structure tree (see chapter 8.6.4), you can set the prices directly in the structure tree without opening the feature editor.

Click in the line with the inherited 0 and type the desired price. The blue icon changes the color into orange to show the overload.



Proceed in the same way for all object classes (O120, D70, steel rims, etc.) and allocate prices for the components.



10.3.5. Practice: Configurationbox columns to display the Price

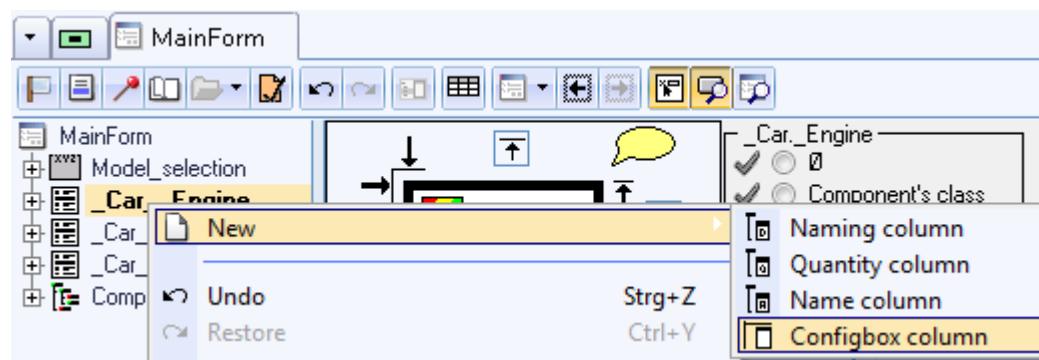
In the already existing configboxes for the modules *engines*, *rims* and *tyres* the just defined amounts have to be displayed.

To do so, you have the possibility to create additional columns under a configbox. The so-called **configurationbox columns** are used to display features of the configbox component values (object classes).

The advantage of the configurationbox column is that it can access the init value of features without an existing object of the class. Therefore the user can see the initialized price without having to previously select the module.

Open the class start and the contained form *MainForm*.

In the form tree you select the configurationbox component which is responsible for the display of the engine. Now you create a new column via the context menu item *New -> Configurationbox column*.

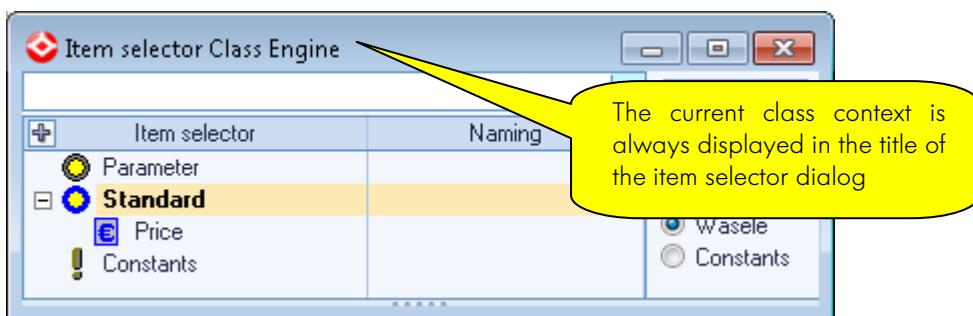




In order to assign the feature which should be displayed to the column, you click on the icon in the field *Cause variable*, select the feature *Price* and confirm with *OK*.



Attention: The cause variable of a column that is subordinated to a configbox is not allocated with an absolute path specification, but it applies the main path of the superior form element. I.e. only the feature *Price* is specified as *EFG* for the price column in the configbox of the engine while the path *_Car._Engine* is already defined for the configbox.



Open the form preview and adjust the width of the configbox in a way that the additional column is completely visible. If necessary, you adjust the width of the price column in the form editor of the column.

A configurationbox column has also the option *Horizontal alignment*. Here you can set how the content is adjusted to the width of the column.



In order to display the price feature right justified, you change the *Horizontal alignment* to *Right*.

A heading for the new column can also be specified in the field *Header*:

Name	<input type="text"/>	Price
Type	<input type="checkbox"/>	Configbox column
Stylesheet	<input type="button" value="ABC"/>	
+ Geometry		
- Representation		
Visible	<input checked="" type="checkbox"/>	
Header alignment	<input type="radio"/>	Centered
Horizontal alignment	<input type="radio"/>	Left
Column width	<input type="radio"/>	Manually
+ Handling		
- Texts		
Header	<input type="text"/> *Price*	
Tooltip	<input type="text"/>	
- Specials		
Cause variable	<input type="text"/> Price	
Attribute of value	<input type="text"/>	

Create a Textelement for the header of the price column.

Proceed in the same way for the other configboxes on the form according to the same scheme and supplement the price column.

Then you start the application and check how the form and the display of the module prices look like.



Example for the display of the engine prices:

Possible engines		Price
<input type="checkbox"/>	✓ Otto	
	<input type="radio"/>	Otto engine 50 \$350.00
	<input type="radio"/>	Otto engine 120 \$800.00
<input type="checkbox"/>	✓ Diesel	
	<input type="radio"/>	Diesel engine 70 \$330.00
	<input type="radio"/>	Diesel engine 110 \$750.00

With the option *Header Alignment* the alignment of column-naming can be set.



10.4. Option “Reinitialize when object changes”

With the testing you notice that the value in the price column changes if you switch between the different engine types.

Possible engines		Price
<input checked="" type="checkbox"/>	Otto	
<input checked="" type="checkbox"/>	<input type="radio"/> Otto engine 50	\$350.00
<input checked="" type="checkbox"/>	<input checked="" type="radio"/> Otto engine 120	\$350.00
<input checked="" type="checkbox"/>	Diesel	
<input checked="" type="checkbox"/>	<input type="radio"/> Diesel engine 70	\$330.00
<input checked="" type="checkbox"/>	<input type="radio"/> Diesel engine 110	\$750.00

After a closer analysis the following phenomenon can be determined: on switching between different engines, the price of the first selected engine is transferred to any later selections! Therefore the current and the last selected engine always have the same price.

The reason is that the feature *Price* is not reset with the switching between the values (object classes of *_Engine*) and not initialized with the amount that is defined as init value. In order to recover the problem, the option *Reinitialize when object changes* is activated in the feature editor of *Price*.

Since this behavior always occurs in all modules, the option is activated on the feature *Price* in the class *Modules* and therefore it is inherited to all module classes.



Open the editor of the feature *Price* in the class *Modules* and activate the switchbox *Reinitialize when object changes*.

Feature: Price	<input checked="" type="checkbox"/> Display not selected properties
Type	<input type="button" value="€"/> Currency
Structureclass	<input type="button" value=""/>
Help	<input type="button" value="?"/>
<input type="checkbox"/> List	
<input type="checkbox"/> Temporary	
<input type="checkbox"/> Rules enabled	
<input type="checkbox"/> Check relevant	
<input checked="" type="checkbox"/> Reinitialize when object changes	
<input checked="" type="checkbox"/> Translate relevant	
Naming	<input type="button" value="E"/>
NOVALUE naming	<input type="button" value="E"/> *please select...*
<input type="checkbox"/> Procedural feature	
<input type="checkbox"/> Init values if NOVALUE	

If features of component- or feature values do not accept the desired values, then this switchbox was not activated (this is frequently forgotten).

The property *Reinitialize when object changes* can be turned on and off, because in some cases a certain configuration has to be kept with switching the model / object.

- ⌚ [Workbench/Structure Tree/Components/Properties/Reinitialize when object changes](#)
- ⌚ [Workbench/Structure Tree/Features/Properties/Reinitialize when object changes](#)

10.5. Tips & Tricks

10.5.1. Columns of a configuration box

As you might have already noticed, a further element is always created automatically with the creation of a new configurationbox component, the *configurationbox naming column*.

This is needed to display the name or (if existing) the naming (see chapter 8.5.1) of the value classes. If you delete the naming column, the namings of the values are missing, e.g.:

With naming column	Without naming column																																	
<table border="1"> <thead> <tr> <th colspan="2">Possible engines</th> <th>Price</th> </tr> </thead> <tbody> <tr> <td><input checked="" type="checkbox"/></td> <td><input checked="" type="radio"/></td> <td>Please select...</td> </tr> <tr> <td><input checked="" type="checkbox"/></td> <td><input type="radio"/></td> <td>Otto engine 50 \$350.00</td> </tr> <tr> <td><input checked="" type="checkbox"/></td> <td><input type="radio"/></td> <td>Otto engine 120 \$800.00</td> </tr> <tr> <td><input checked="" type="checkbox"/></td> <td><input type="radio"/></td> <td>Diesel engine 70 \$330.00</td> </tr> <tr> <td><input checked="" type="checkbox"/></td> <td><input type="radio"/></td> <td>Diesel engine 110 \$750.00</td> </tr> </tbody> </table>	Possible engines		Price	<input checked="" type="checkbox"/>	<input checked="" type="radio"/>	Please select...	<input checked="" type="checkbox"/>	<input type="radio"/>	Otto engine 50 \$350.00	<input checked="" type="checkbox"/>	<input type="radio"/>	Otto engine 120 \$800.00	<input checked="" type="checkbox"/>	<input type="radio"/>	Diesel engine 70 \$330.00	<input checked="" type="checkbox"/>	<input type="radio"/>	Diesel engine 110 \$750.00	<table border="1"> <thead> <tr> <th colspan="2">Possible engines</th> <th>Price</th> </tr> </thead> <tbody> <tr> <td><input checked="" type="checkbox"/></td> <td><input checked="" type="radio"/></td> <td>\$350.00</td> </tr> <tr> <td><input checked="" type="checkbox"/></td> <td><input type="radio"/></td> <td>\$800.00</td> </tr> <tr> <td><input checked="" type="checkbox"/></td> <td><input type="radio"/></td> <td>\$330.00</td> </tr> <tr> <td><input checked="" type="checkbox"/></td> <td><input type="radio"/></td> <td>\$750.00</td> </tr> </tbody> </table>	Possible engines		Price	<input checked="" type="checkbox"/>	<input checked="" type="radio"/>	\$350.00	<input checked="" type="checkbox"/>	<input type="radio"/>	\$800.00	<input checked="" type="checkbox"/>	<input type="radio"/>	\$330.00	<input checked="" type="checkbox"/>	<input type="radio"/>	\$750.00
Possible engines		Price																																
<input checked="" type="checkbox"/>	<input checked="" type="radio"/>	Please select...																																
<input checked="" type="checkbox"/>	<input type="radio"/>	Otto engine 50 \$350.00																																
<input checked="" type="checkbox"/>	<input type="radio"/>	Otto engine 120 \$800.00																																
<input checked="" type="checkbox"/>	<input type="radio"/>	Diesel engine 70 \$330.00																																
<input checked="" type="checkbox"/>	<input type="radio"/>	Diesel engine 110 \$750.00																																
Possible engines		Price																																
<input checked="" type="checkbox"/>	<input checked="" type="radio"/>	\$350.00																																
<input checked="" type="checkbox"/>	<input type="radio"/>	\$800.00																																
<input checked="" type="checkbox"/>	<input type="radio"/>	\$330.00																																
<input checked="" type="checkbox"/>	<input type="radio"/>	\$750.00																																

The naming column disposes of the property Header as well as the configurationbox column.

Possible engines		
	Engines	Price
<input checked="" type="checkbox"/>	Otto	
<input checked="" type="checkbox"/>	<input checked="" type="radio"/>	Otto engine 50 \$350.00
<input checked="" type="checkbox"/>	<input type="radio"/>	Otto engine 120 \$800.00
<input checked="" type="checkbox"/>	Diesel	
<input checked="" type="checkbox"/>	<input checked="" type="radio"/>	Diesel engine 70 \$330.00
<input checked="" type="checkbox"/>	<input type="radio"/>	Diesel engine 110 \$750.00

In addition to the naming column the following three column types are available:

Name column

The name column is used if the class names have to be displayed. Possibly existing object namings are ignored by this column.

Possible engines		
	Engines	Price
<input checked="" type="checkbox"/>	Please select...	
<input checked="" type="checkbox"/>	Otto engine 50	\$350.00
<input checked="" type="checkbox"/>	Otto engine 120	\$800.00
<input checked="" type="checkbox"/>	Diesel engine 70	\$330.00
<input checked="" type="checkbox"/>	Diesel engine 110	\$750.00

Quantity column

The quantity column is used to change the quantity of an object. You can find more details in chapter 26.

Keys		
	Quantity	Price per pcs.
<input checked="" type="checkbox"/>	Emergency key	3 pcs. \$20.00
<input checked="" type="checkbox"/>	Remote control key	1 pcs. \$60.00

Configurationbox column

As already described in practice 10.3.5, configurationbox columns are used to display properties of component values. Basically only features or constants can be displayed. The form element cannot be used to change values.

With the allocation of the cause variable of the column has to be considered that only the name of the feature is specified. The path to the object in which the feature is located is already allocated by the cause variable on the configbox.

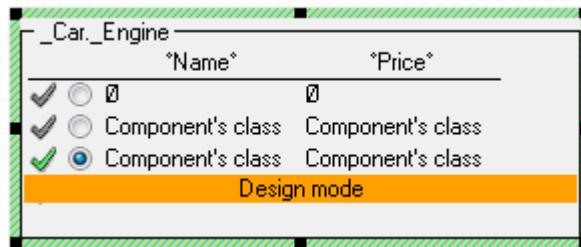
This can also be seen in the item selector dialog. If this is opened in the line *Cause variable* of a configurationbox column, it is immediately started in the class that was specified as cause variable of the configbox.

10.5.2. Change the column width in the form preview

In the form editor the width of a column can be edited in the field *Width*. The form preview offers another possibility. Here the width of a column can be changed via shifting the column separator with the mouse.

To do so, first the *Design mode* has to be activated; otherwise columns in the form preview cannot be selected. The *Design mode* is activated via selecting the form element in the preview and via selecting the context menu item *Design mode*.

While the mode is active, an orange-colored label is displayed on the preview. As soon as another form element is selected, the *Design mode* is automatically terminated.



10.5.3. Lines for the tree representation of a configbox

As from a certain number of columns the display of a configbox can be confusing. In order to improve the display you can activate horizontal and vertical lines.

Form editor		Interpreter																									
Frame	<input type="checkbox"/> Line border	<table border="1"> <thead> <tr> <th colspan="3">Possible engines</th> </tr> <tr> <th></th> <th>Name</th> <th>Price</th> </tr> </thead> <tbody> <tr> <td><input type="checkbox"/></td> <td>Otto</td> <td></td> </tr> <tr> <td><input checked="" type="checkbox"/></td> <td>Otto engine 50</td> <td>\$350.00</td> </tr> <tr> <td><input checked="" type="checkbox"/></td> <td>Otto engine 120</td> <td>\$800.00</td> </tr> <tr> <td><input type="checkbox"/></td> <td>Diesel</td> <td></td> </tr> <tr> <td><input checked="" type="checkbox"/></td> <td>Diesel engine 70</td> <td>\$330.00</td> </tr> <tr> <td><input checked="" type="checkbox"/></td> <td>Diesel engine 110</td> <td>\$750.00</td> </tr> </tbody> </table>		Possible engines				Name	Price	<input type="checkbox"/>	Otto		<input checked="" type="checkbox"/>	Otto engine 50	\$350.00	<input checked="" type="checkbox"/>	Otto engine 120	\$800.00	<input type="checkbox"/>	Diesel		<input checked="" type="checkbox"/>	Diesel engine 70	\$330.00	<input checked="" type="checkbox"/>	Diesel engine 110	\$750.00
Possible engines																											
	Name	Price																									
<input type="checkbox"/>	Otto																										
<input checked="" type="checkbox"/>	Otto engine 50	\$350.00																									
<input checked="" type="checkbox"/>	Otto engine 120	\$800.00																									
<input type="checkbox"/>	Diesel																										
<input checked="" type="checkbox"/>	Diesel engine 70	\$330.00																									
<input checked="" type="checkbox"/>	Diesel engine 110	\$750.00																									
Border color	<input type="color" value="black"/> (0,0,0)																										
Display style	<input type="color" value="lightblue"/> 2007																										
Horizontal lines	<input checked="" type="checkbox"/>																										
Vertical lines	<input checked="" type="checkbox"/>																										
Break in cells	<input type="checkbox"/>																										

If necessary, a color for the lines can be additionally set on the individual columns.

10.6. Repetition

- Features are used to describe an object in detail.

- Features are defined in the structure tree of a class. In derived classes many properties of the feature can be overloaded. Inherited properties are indicated by a blue sheet , overloaded properties are indicated by an orange-colored symbol 
- A feature can have any amount of values.
- Features can get a “Default” value via an init value.
- On the form element configbox component you can display additional columns in order to display features within the Object classes of the value.

11. Price calculation

11.1. Intention

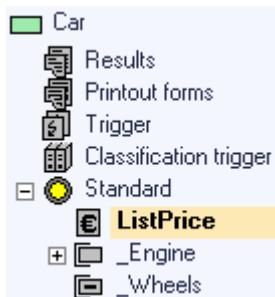
Now the prices for the individual components that are allocated in chapter 10 have to be added to a total price and displayed on the form. To do so, a certain list price for each model is estimated and the individual prices of the selected modules are added to this list price.

In this chapter you will learn how to carry out a simple **price calculation via constructors and destructors**. Furthermore you will get to know the component type predecessor and the form element **Currency**.

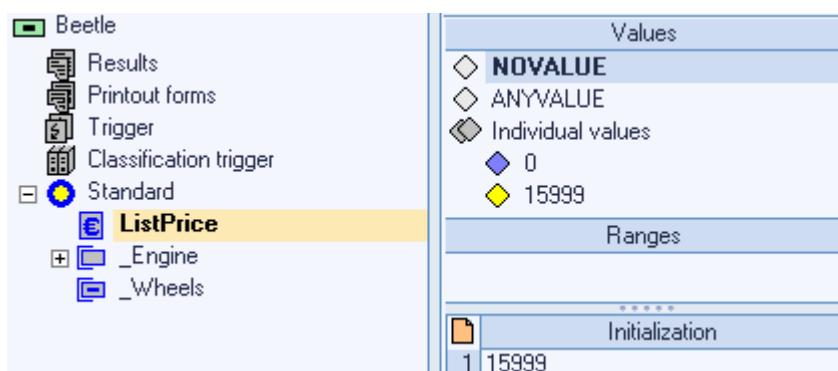
11.2. Practice: Defining list prices

First the list price for the models is determined and displayed on the form. As with the price of the modules a currency feature is used.

Open the class *Car* and create a new feature with the name “*ListPrice*” and the data type *Currency*.



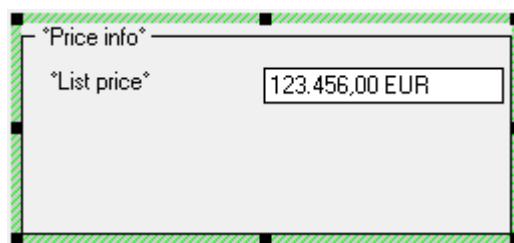
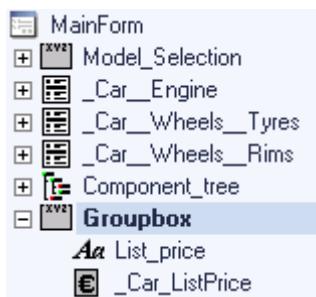
Open the classes *Beetle*, *Golf* and *Passat* and overload the init value of the *ListPrice* with any values.



In order to display the list price on the form, the *MainForm* has to be extended.

 Open the form *MainForm* in the class *start*. Create a new groupbox "Price info" analog to the groupbox "Model selection" in which the product images are displayed, see chapter 7.3.1.

 In this groupbox you create a new currency form element. As cause variable you define the feature *ListPrice* of the class *Car*. Additionally you create a static label with the naming "List price:". Enter ":" in the line Postfix.



 Start the application and check the correct display of the list price for each model.

 Since the currency element is not supposed to be edited, you activate the option *Readonly* in the form editor. Additionally you can set the *Horizontal alignment* of the form element to *Right*.

Representation	
Visible	<input checked="" type="checkbox"/>
Font	 Standard
Foreground color	 COLOR_SYSTEM
BG color	 COLOR_SYSTEM
Focus FG color	 COLOR_SYSTEM
Focus BG color	 COLOR_SYSTEM
Frame	<input type="checkbox"/> Line border
Border color	 {0,0,0}
Horizontal alignment	 Right
Handling	
Enabled	<input checked="" type="checkbox"/>
Readonly	<input checked="" type="checkbox"/>
Assign value with Enter	<input type="checkbox"/>
Tabstop	<input checked="" type="checkbox"/>
Popupmenu	
ViewShot relevant	<input checked="" type="checkbox"/>
Texts	
Specials	
Cause variable	 _Car.ListPrice

11.3. Preliminary considerations with regard to the price calculation

11.3.1. When is the price calculation carried out?

Because of the init value of the *ListPrice* the list price already exists with the instantiation of the car object. But when should the addition of the individual prices be carried out?

One possibility would be to leave the decision to the user. He presses e.g. a pushbutton that gives him the current total price in a message.

However, in order to be able to always display the current price on the form, the total price has to be always recalculated if the user selects or deselects a new component.

On the event “an object is created” you can always react with the system method *New()*, the so-called constructor. You have already heard about the constructor in chapter 3.2.3.1 and used it to open the *MainForm*.

With the event “the object is deleted”, however, the destructor is automatically carried out.

With the deletion of an object automatically its **destructor** is carried out, a method with the name *Delete()*.

Therefore you can use the methods *New* and *Delete* to trigger the price calculation. The calculation is carried out in the *Car* object. When creating a *Module* object, a method in the *Car* is called and the price of the *Module* is transferred to add its price to the list price. When deleting a *Module* object, another method is called to subtract the price.

Since all components are combined under the base class *Modules*, the required methods *New* and *Delete* have to be created only once in the class *Modules*. Via the inheritance each module object knows what to do with the instantiation/deletion.

11.3.2. Saving the calculation result

The addition/subtraction of an individual price is always carried out if a module object is created/deleted. In this case the module object has to call the corresponding method in the car object to add or subtract its own price.

The result of the calculation is then saved in the *ListPrice* of the car via an assignment. You already know the assignment from chapter 3.2.3.3 when the return value of the method *WinOpen* was saved in the feature *winhandle*.

The same syntax is also used with the saving of the result of list price + individual price.

List price of the car := list price of the car + individual price of the component.

Above code verbatim: The new list price of the car results from the current list price of the car plus the individual price of the just selected component.

Alternatively this code can be shortened like this:

List price of the car += individual price of the component.

For the subtraction (method `Delete()`) the same syntax is valid:

List price of the car := list price of the car – individual price of the component.

Verbatim: The new list price of the car results from the current list price of the car minus the individual price of the just selected component.

Or again in short form:

List price of the car -= individual price of the component.

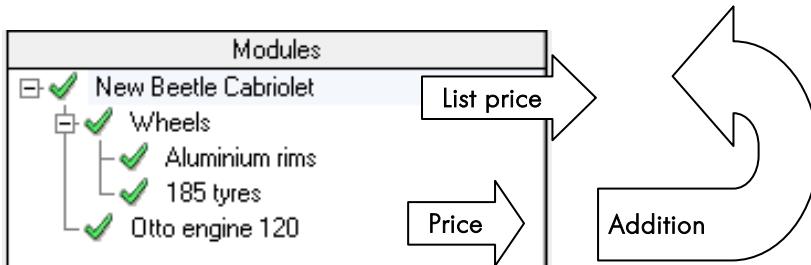
11.3.3. Access to properties of a superior object

There is a new problem arising from the preliminary considerations of the previous chapter 11.3.2: being “inside” a module object, how can you access the addition and subtraction method of the car?

The syntax `_Component.Property` or `_Component.Method()` (see practice chapter 6.3.1) cannot be used here because with this “path specification” only the properties of child objects can be accessed. Out of an object `Car` you can e.g. access the feature `Price` of the object `Engine` via the path `_Engine.Price`.

But with the price calculation the access has to be carried out “from bottom to top”: from the child object of the car to the list price of the car, e.g. the price of the engine has to be added to the list price of the car.

The component tree clarifies the object structure:



In order to access a superior object via a child object, a so-called predecessor component is needed, see chapter 4.2.5.

Predecessor components are created analog to “normal” components, i.e. they are typified by a class that was selected with the creation. However, predecessor components do not have an underscore as prefix, but an `@`. They also have no values or an init value, but they always automatically contain the name of the “upwardly” next object that matches the class specification of the predecessor component.

So in this case we need a predecessor of the class `Car` within the module objects in order to be able to access the feature `ListPrice` of the car via this component.

11.4. Practice: Implementing the price calculation

The following work steps result of the preliminary considerations:

Open the class *Car* and create the method *PriceAddition*. Define the parameter *Price_* (data type currency).



Deposit the following procedure code:

```
ListPrice += Price_;
```

This method adds the transferred module price to the list price. In the next step, a method to subtract the price is created.

Create the method *PriceSubtraction*. Define the parameter *Price_* (data type currency).

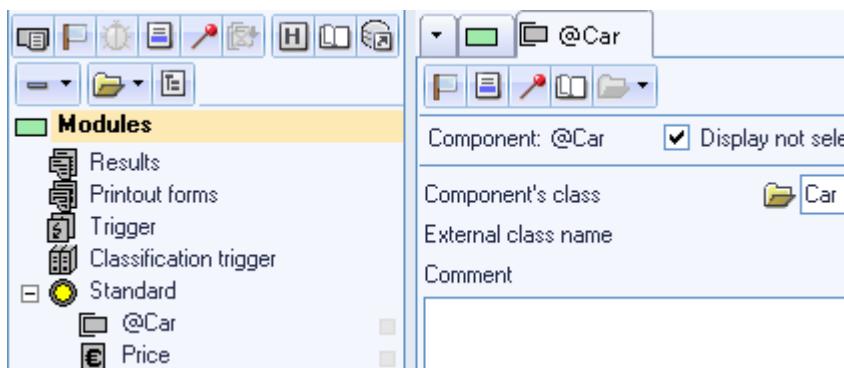


Deposit the following procedure code:

```
ListPrice -= Price_;
```

Next, the predecessor component is created, so that the methods *PriceAddition* and *PriceSubtraction* can be called from the module objects.

Open the class *Modules* and drag the class *Car* via D&D from the class tree in the structure tree of *Modules*. In the dialog *New component* you activate the option *Predecessor* and confirm the creation with *OK*.



Create a new method with the name “new” in the wasele lists of the class *Modules*. There you enter the following procedure code:

```
@Car.PriceAddition(Price);
```

Create a further method with the name “Delete” and there you enter the following procedure code:

```
@Car.PriceSubtraction(Price);
```

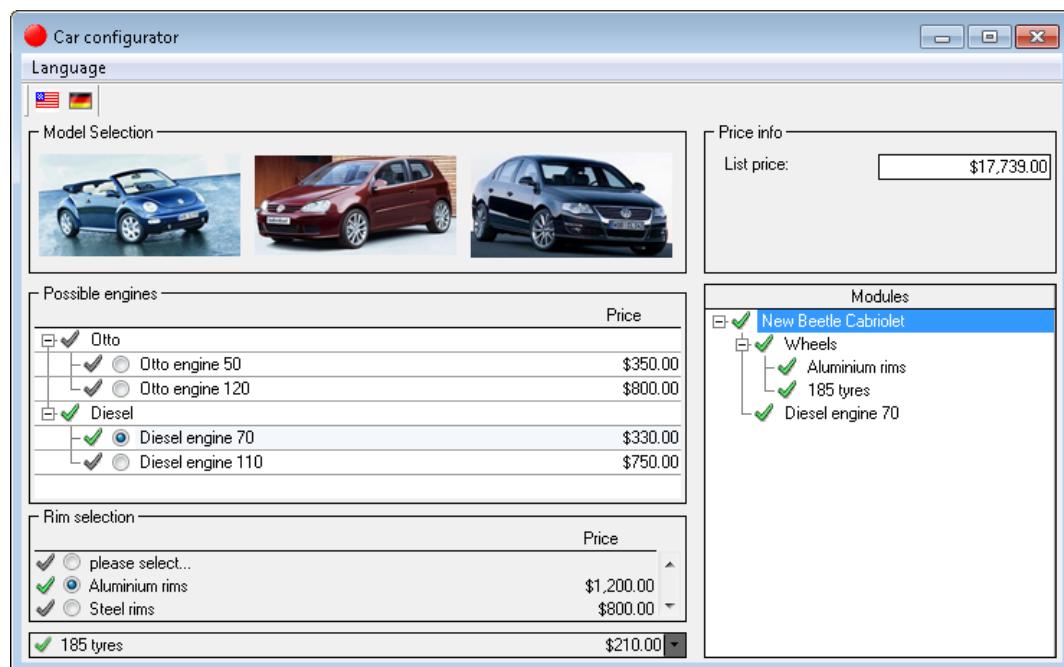
Carry out a syntax check in both methods (Ctrl + S or icon).



Attention: If you should have named your EFGs differently, you adjust this correspondingly in the procedure code. Or you use the item selector dialog from the start.

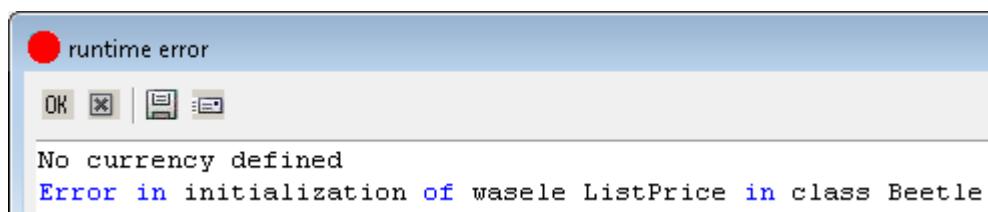


Now you start the configurator and test the price calculation. Observe the value of the list price while you select and deselect various modules.



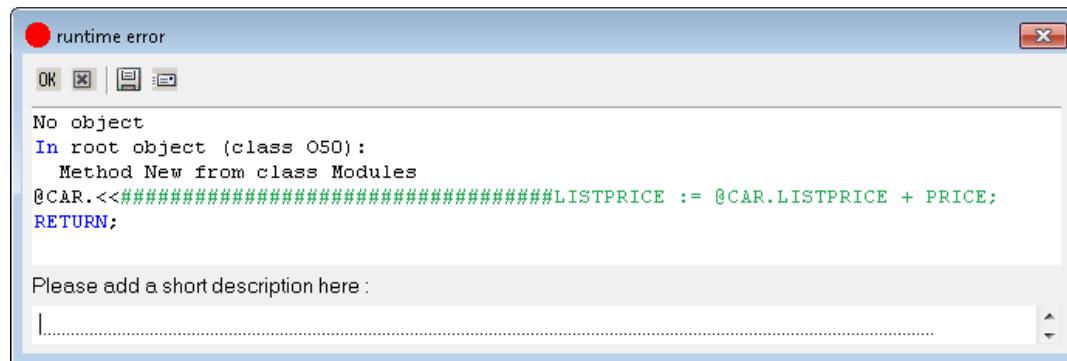
11.5. Error Sources

If you forgot to define a currency in the frame of the knowledge base, the syntax check provides the error "There is no currency defined". During runtime the interpreter error "No currency defined" appears with the first access to a currency feature.



Because of the existence of constructor and destructor in the module classes, from now on you will get an interpreter error if you start the application from a different class than *start*. If e.g. the class *SteelRims* is selected in the class tree and you start the debugger, the interpreter error message "No object" is displayed.

This message means that you try to access a property of an object that does not yet exist. So the predecessor component @Car only gets a value if the application was started from the class *start* (object *start* exists) and a model was selected (object *Car* exists).



[Basics/Errorstack](#)

11.6. Repetition

- The constructor (method *New*) is always running if an object is created.
- The destructor (method *Delete*) is always running if an object is deleted.
- The wasele of a superior object can be accessed via a predecessor component.
- Predecessor components begin with the prefix @.

12. Integrate further modules

12.1. Intention

The target of the following chapter is the extension of the previous configuration by further modules. In addition to the already existing modules also the interior fittings, the paintwork and any combination of accessories should be configurable.

To do so, you will complete the existing class structure and repeat the maintenance of prices and namings. You will get to know list components and again extend the form by the created components.

In the previous practice chapters you already carried out many of the required actions in a similar way and therefore the work instructions in this chapter will not contain all of the individual steps.

If you should not remember the single work steps, you can look up the creation of the class structure in chapter 6.2.1 and the initially valuing and overloading of the feature *Price* in the chapter 10.3.3 and the following chapters.

12.2. Practice: Extending the class structure

In the first step the modules for the interior fittings, the paintwork, the accessories and the soft top should be integrated in the application.

For the interior fittings you have the variants fabric and leather, for the paintwork the variants normal and metallic and as optional accessories radio, air conditioning, sunroof, CD changer, baggage roof carrier and navigation system can be selected.

The soft top is a special case. There is no typifying through different soft top variants. Also, the soft top may only be part a convertible. Therefore the soft top is realized as object class; compare it with the class *Wheels* in chapter 6.2.1.

Create the class structure according to the above considerations in creating the required base- and object classes as child classes of *Modules*.

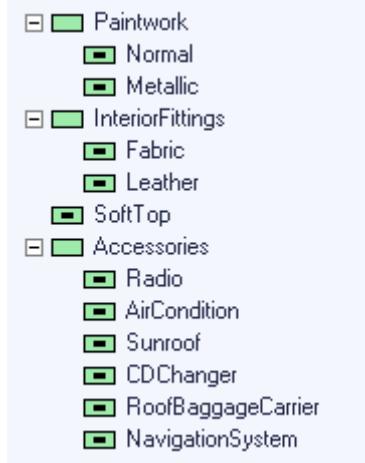


Since the new classes were created as derivation of *Modules*, the feature *Price* was inherited to all classes.

Overload the init value of the feature *Price* with any value in each of the object classes. If necessary, maintain the object namings for classes that do not have a meaningful name.



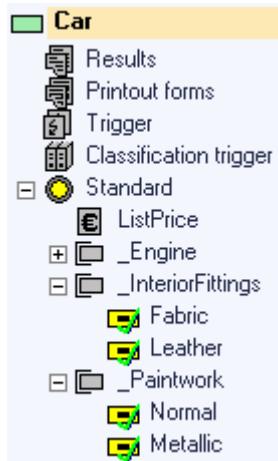
After your extension the class tree under *Modules* should look like this:



12.3. Practice: Create components



Create components of the classes *InteriorFittings* and *Paintwork* in the class *Car* and apply all values to the structure tree.



Since the soft top is only possible for cars of the type *Convertible*, the component *SoftTop* is not integrated in the base class *Car*, but it is created as a component in the class *Convertible*.



Create the component *_SoftTop* in the class *Convertible*.

Since each convertible model must have a soft top, the component *_SoftTop* (see init value of the component *_Wheels* in chapter 6.2.2) is initialized with the value *SoftTop*.

To do so, you drag the value *SoftTop* with the pressed mouse button into the field *Init value*.

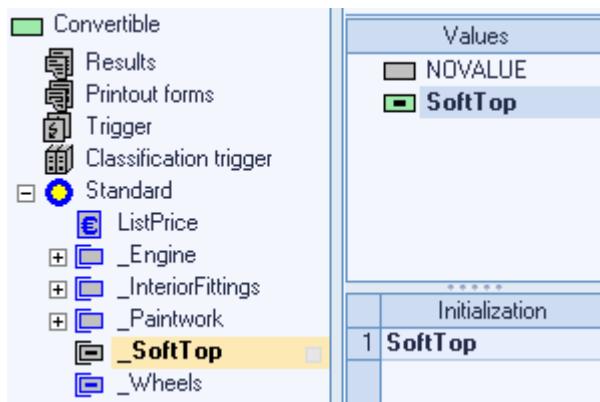
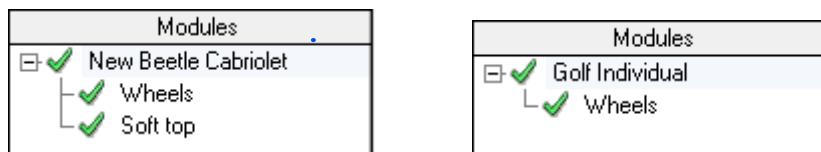


Figure 30: Initialisation of a component

Start your application and observe the change in the component tree with the switching of the car models.



The initialized structure of the model is visible in the component tree. If a car of the type *Convertible* is selected, the component *SoftTop* is automatically added; with closed models this is not the case.



12.4. List components

For the accessories it should be possible to select not only one but two or more parts. It should also be possible to select no accessory.

To do so, a **list component** of the class *Accessories* is created.

A list component gets the extension `[]` in the name. List components can contain any amount of values of the same type.

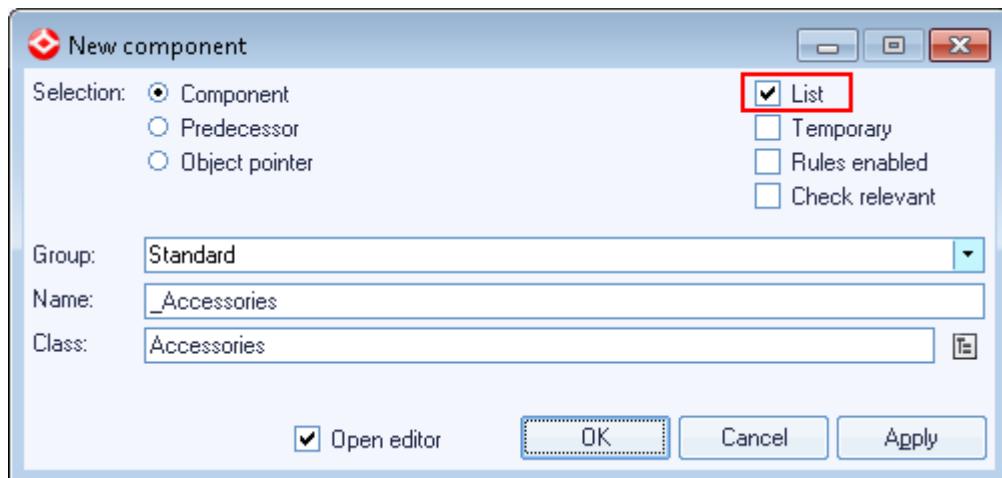
The individual list entries are addressed via the specification of the list location (index) in square brackets, e.g. *Accessories[4]*. In camos Develop lists begin with the index 1.

List elements can be initialized with any amount of values. The same is valid for list features.

12.4.1. Practice: Create a list component



Open the class *Car*. Drag the class *Accessories* onto the group *Standard* of the structure tree. In the dialog *New component* you activate the option *List*.

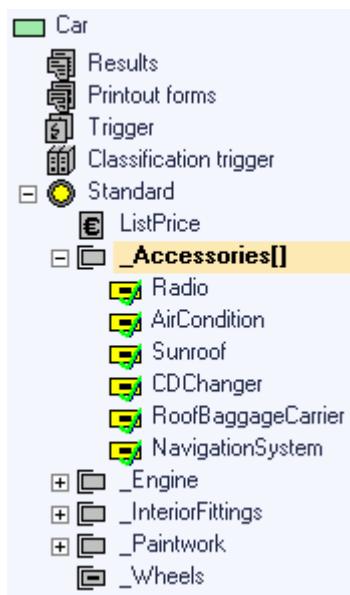


With the creation of the list component square brackets are added to the name of the component as far as they were not manually specified with the creation.



Apply all values of the list component *_Accessories[]* to the structure tree.

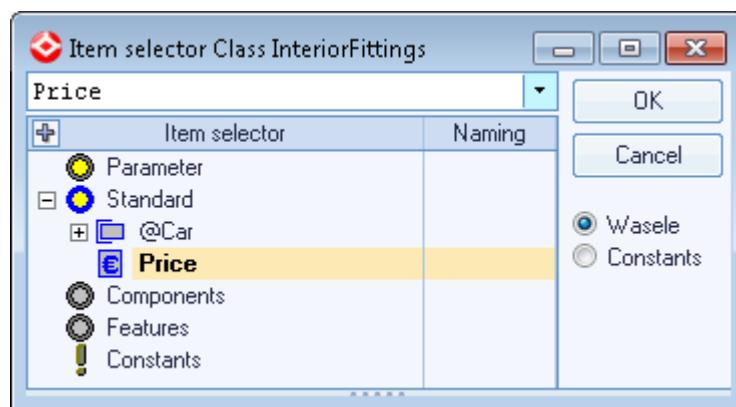
Then the structure tree of the class *Car*, with all components, looks like this:



12.5. Practice: Add the modules to the MainForm

As in the previous practices, the components `_InteriorFittings` and `_Paintwork` should be displayed on the form via configuration boxes for components.

Create two new configuration boxes components and specify the respective cause variable (`_Car._InteriorFittings` or `_Car._Paintwork`). Add for each a configbox a column for the feature `Price`.



Open the form preview and place the configboxes at an empty location.

Start your application and check the correct display of the new modules incl. values and prices.

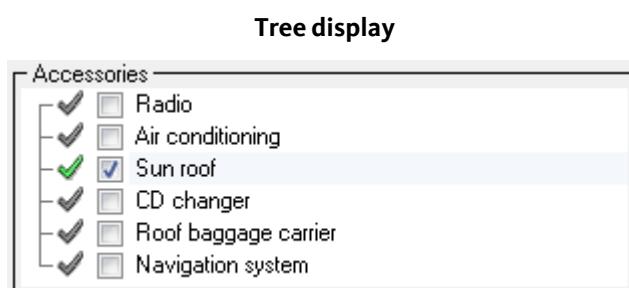


12.5.1. Representation of list components

Depending on the type of the cause variable, configboxes can be displayed in different representations. For configboxes with a scalar cause variable, the representations `Compact` , `List`  or `Tree`  can be selected.

Configboxes with list components as cause variable offer the representations `List` , `Extended list`  and `Tree` .

Create a further configurationbox component on the form, assign the cause variable `_Car._Accessories[]` and arrange the configbox on the form preview. Test the different representations in the interpreter.



List display

Accessories	
<input checked="" type="checkbox"/>	<input type="checkbox"/> Radio
<input checked="" type="checkbox"/>	<input type="checkbox"/> Air conditioning
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> Sun roof
<input checked="" type="checkbox"/>	<input type="checkbox"/> CD changer
<input checked="" type="checkbox"/>	<input type="checkbox"/> Roof baggage carrier
<input checked="" type="checkbox"/>	<input type="checkbox"/> Navigation system

Extended list display

A click on that icon opens a dialog to select new values

Dialog in which new values are assigned via ← and previous values can be deleted via X.

After you have decided which type of representation you want to use, you create the configurationbox column for the feature *Price* of the accessories.



Test your application.

The following figure shows the current state of the development:

Car configurator

Language: English

Model Selection: Three car models are shown: New Beetle Cabriolet, Golf, and Passat.

Possible engines

	Price
Otto	\$350.00
Otto engine 50	\$350.00
Otto engine 120	\$800.00
Diesel	\$330.00
Diesel engine 70	\$330.00
Diesel engine 110	\$750.00

Rim selection

	Price
please select...	
Aluminium rims	\$1,200.00
Steel rims	\$800.00
175 tyres	\$160.00

InteriorFittings

	Price
please select...	
Fabric interior	\$500.00
Leather interior	\$800.00

Accessories

	Price
Radio	\$605.00
Air conditioning	\$1,020.00
Sun roof	\$800.00
CD changer	\$600.00
Roof baggage carrier	\$450.00
Navigation system	\$1,250.00

Price info: List price: \$21,244.00

Modules

- New Beetle Cabriolet
 - Wheels
 - Aluminium rims
 - 175 tyres
 - Otto engine 50
 - Radio
 - Roof baggage carrier
 - Navigation system
 - Fabric interior
 - Regular paintwork
 - Soft top

12.6. Repetition

- In order to be able to select several values from a component, the component has to be defined as a list.
- List components are indicated by an attached [].
- In lists the index begins with the value 1.
- On configbox components the display types refer to the cause variable (scalar or list).

13. Options of the component tree

13.1. Intention

As already mentioned in chapter 6.3.2, an application in which products with a lot of variants are configured can barely do without the form element component tree. This tree illustrates the current product structure, i.e. it displays which modules are selected and how the individual modules are connected to each other or if and how they build up on each other (multilevel object structure).

The component tree is not only a browser for the product structure, but it also offers a lot of view- and interaction possibilities that simplify the working and dealing with the component tree.

The most frequently used options will be introduced in this chapter. The required adjustments on the knowledge base do not contain any new elements. In the following practices you repeat the creation of classes, features and the form element columns. Only the purpose and the use of the new classes and wasele differ from the previous cases because the product car is not affected by these adjustments.

13.2. Component tree columns

Let us have a look at the object structure of a Golf if all its modules are selected.

Modules	
<input checked="" type="checkbox"/>	Golf Individual
<input checked="" type="checkbox"/>	Wheels
	Aluminium rims
	155 tyres
<input checked="" type="checkbox"/>	Otto engine 120
<input checked="" type="checkbox"/>	Radio
<input checked="" type="checkbox"/>	Air conditioning
<input checked="" type="checkbox"/>	Navigation system
<input checked="" type="checkbox"/>	Fabric interior
<input checked="" type="checkbox"/>	Regular paintwork

The *root* of the component tree shows that the current model is a Golf.

Under the car object all selected modules can be seen, i.e. the modules are *child objects* (= components) of the car.

Under the module *Wheels* we see the *Rims* and *Tyres* since these are in turn child objects of the object *Wheels*.

Due to the option *Object naming* that is activated in the form editor of the component tree we can see the (multilingual) object namings instead of the class names.

However, the user cannot see the price of the selected components in the component tree. With a relatively small product structure and an accordingly clear surface the user can find the price of a certain component very quickly. But often the products consist of hundreds of individual parts. Therefore it is useful to display the prices directly in the component tree.

As you already know from the price display in the configboxes, columns are used to display the value of a feature of an object. Contrary to the configboxes the component tree has no column type that can display all data types. Therefore you have to consider the type of the feature that has to be displayed with the creation of a new component tree column.

In general the type of the column corresponds to the data type of the feature that has to be displayed. In case of the price feature therefore a currency column has to be selected.



Open the *MainForm* in the class *start* and select the component tree in the form tree. Add a new currency column to the component tree via Context menu -> New -> Currency.

In a component tree the most different objects can be displayed, e.g. the car objects and the module objects are not related to each other according to the class tree. I.e. it is not guaranteed that all objects have the same properties, e.g. the *Car* does not have a feature *Price* and the *Modules* do not have a feature *ListPrice*.

In order to allocate a cause variable to a component tree column, you first have to determine which classes contain this cause variable. This specification is made using the field *Column class*. In our case all classes that are derived from *Modules* contain the feature *Price*.



Drag the class *Modules* in the field *Column class* and then you select the feature *Price* in the item selector of the field *Cause variable*.

Specials	
Cause variable	<input type="button" value="Price"/>
Column class	<input type="button" value="Modules"/>
Currency	<input type="button" value="Automatic"/>



Start the application and check appearance and function of the component tree. Define a column heading and adjust, if necessary, the width and alignment of the price column.



If you click in the price column, you will find out that it can be edited. Contrary to the configbox (most) columns of a component tree can be edited. I.e. the user can change the value of a feature in an object as he likes. Here this is not wanted and therefore the column has to be write-protected.

Activate the option *Readonly* in the form editor of the price column.

Handling	
Readonly	<input checked="" type="checkbox"/>
Not scrollable	<input type="checkbox"/>
Menu object reference	<input type="checkbox"/> Object of the selected component
Popupmenu	<input type="checkbox"/>
ViewShot relevant	<input checked="" type="checkbox"/>
Texts	
Header	<input type="text"/> "Price"
Header connect with the next column	<input type="checkbox"/>
Tooltip	<input type="text"/>
Specials	
Cause variable	<input type="button" value="Price"/>
Column class	<input type="button" value="Modules"/>
Currency	<input type="button" value="Automatic"/>

Now the component tree looks as follows:

Modules	Price
☐ ✓ Golf Individual	
☐ ✓ Wheels	\$0.00
└ ✓ Aluminium rims	\$1,200.00
└ ✓ 185 tyres	\$210.00
└ ✓ Diesel engine 70	\$330.00
└ ✓ Radio	\$605.00
└ ✓ Air conditioning	\$1,020.00
└ ✓ CD changer	\$600.00
└ ✓ Roof baggage carrier	\$450.00
└ ✓ Fabric interior	\$500.00
└ ✓ Metallic paintwork	\$980.00

13.3. How to open the component tree automatically

By default the nodes of the component tree are always closed. Contrary to the configbox in the tree representation where the opening default is available as an option in the form editor, this has to be implemented manually for the component tree.

To do so, a **system feature** has to be defined for each object that is displayed in the component tree. System features are features with a predefined name and type that are – if existing – automatically evaluated by the form element.

In order to control whether a node (= an object that has child objects) has to be displayed opened or closed, the numerical system feature *SubTreeOpen* is defined. If this feature has the value 0, the node is displayed closed, otherwise it is opened.

The feature has to be defined for all objects that are displayed in the component tree and that have to be opened by default. In this case these are all module objects as well as the car object itself.

Therefore you should create the same object with the same task at two different locations. According to the basics of the OOP classes with the same properties should be combined under a common base class.

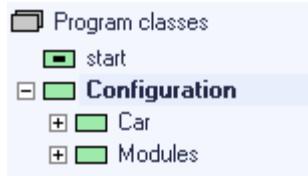
To do so, the class structure is extended by a class *Configuration* and the classes *Car* and *Modules* are shifted (= derived) with their child classes under this class. Like the class *Modules* the class *Configuration* has no function. It is only used to inherit properties to the car and the modules.

Create a new class with the name “*Configuration*” on first level in the class tree and shift the base classes *Car* and *Modules* under *Configuration*.



The D&D in the class tree is only possible if the debugger is not started at that time.

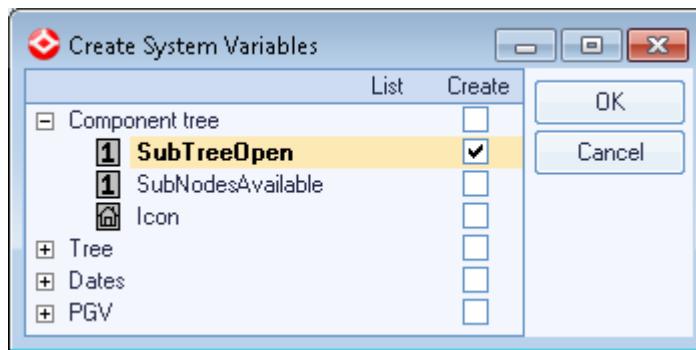




In the class *Configuration* now properties can be created which are automatically inherited to all *Car*- and *Module* classes.



In the class *Configuration* you open the dialog to create a new feature. Select the button *System variables*. In the dialog *Create System Variables* you open the node *Component Tree* and activate the switchbox in the column *Create* of the feature *SubTreeOpen*.



The feature is created after the confirmation with *OK*. Open the feature editor of *SubTreeOpen* and define the init value 1 in order to display all nodes of the component tree open.

Since *SubTreeOpen* is a system feature, further adjustments are not necessary.



Start the application and check how the first node in the tree was automatically opened with the selection of the model.

With the selection of a rim also the node *Wheels* is automatically opened.

13.4. Change the sorting of the components

A further requirement to the component tree is to keep a certain sorting of the objects. By default the objects are sorted in the object tree in an accidental order. Neither the position of the components in the structure tree nor the order in which the objects are created are decisive and therefore these factors cannot be affected.

In order to be still able to manually sort the components, the **component sorting** is activated.

With the instantiation all objects internally get an *order number* that determines the position of the object in the object tree. In order to affect this order number, the *Local sorting for components* is activated in the class editor of the classes whose components have to be sorted.

Via the icon  in the toolbar of the structure tree a separate dialog is opened in which the components can be sorted via D&D as you like. The manual sorting is not visible in the structure tree, there the components are still displayed in alphabetical order.

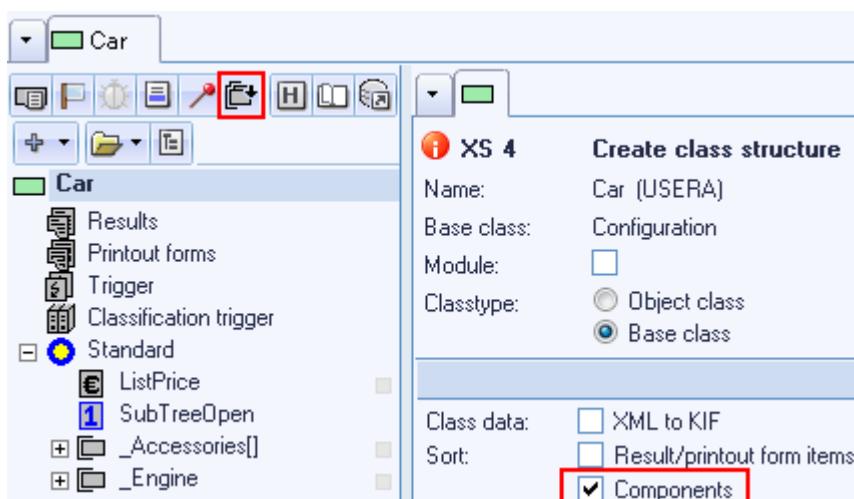
In order to switch the component sorting during runtime, the option *Sort* has to be additionally activated for all affected components.

The module objects in the car configurator should be sorted as follows:



To do so, first the *component sorting* has to be activated.

Open the class *Car* and activate in the class editor the option *Components* under *Sort*. In this moment the icon  in the toolbar of the class editor and the structure tree is activated.



Click on the icon  to open the dialog *Component sorting*.

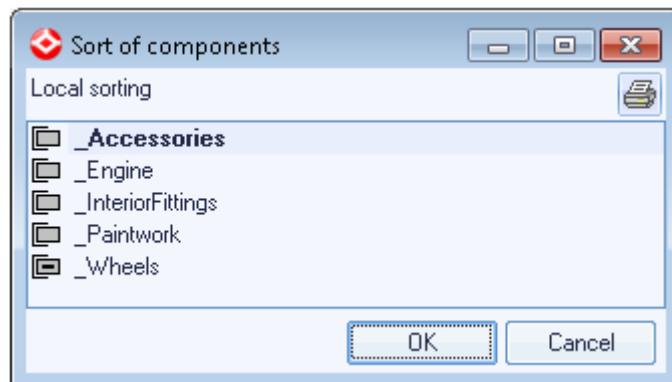


In this dialog you can see the order of the components how it currently exists in the structure tree. Via D&D you can bring the components in the desired order.



Bring the components in the above mentioned order.

Before



After

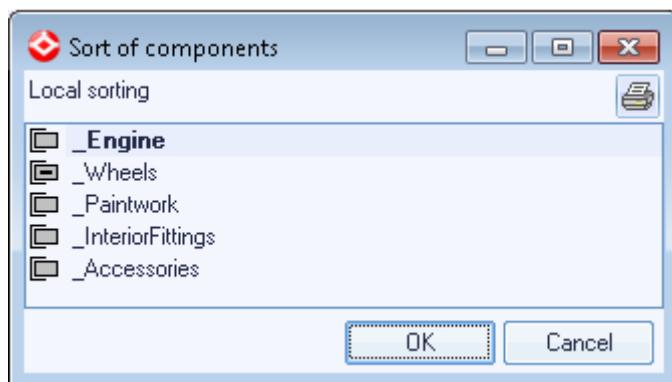


Figure 31: Sorting components



In the structure tree still the alphabetical order is displayed! If the component sorting is activated can only be recognized by the active/inactive icon

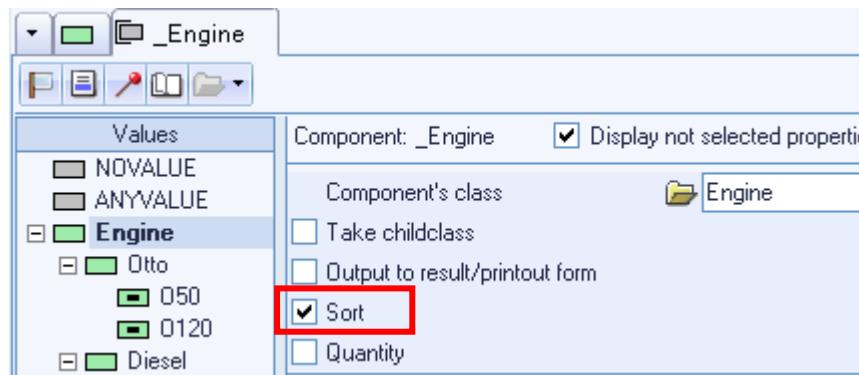
Also the sorting of the rims and tyres has to be changed.



Open the class *Wheels*, activate the *Component sorting* and bring the components *_Tyres* and *_Rims* in the desired order.



In order to use the sorting during runtime, the option *Sort* has to be set on all components that have to be sorted.



13.5. Tips & Tricks

13.5.1. Find properties...

You can set the option *Sort* manually for each component or you can use the dialog *Find properties*.

Mark the class *Configuration* in the class tree and then select the main menu item *Edit -> Find...* or *Find...* in the context menu of the class.

In the dialog *Search* you select the icon *Property* . Select the property *Sort* in the Combobox. Select *No* in the field *Search* and start the searching with the button *Search*.



Per default the search is carried out in the class, which is selected when calling the dialog *Find*. The derived classes can be included or excluded from the search. In spite of this presetting, the search can be carried out for all *program classes* via setting the correspondent option in the search dialog.

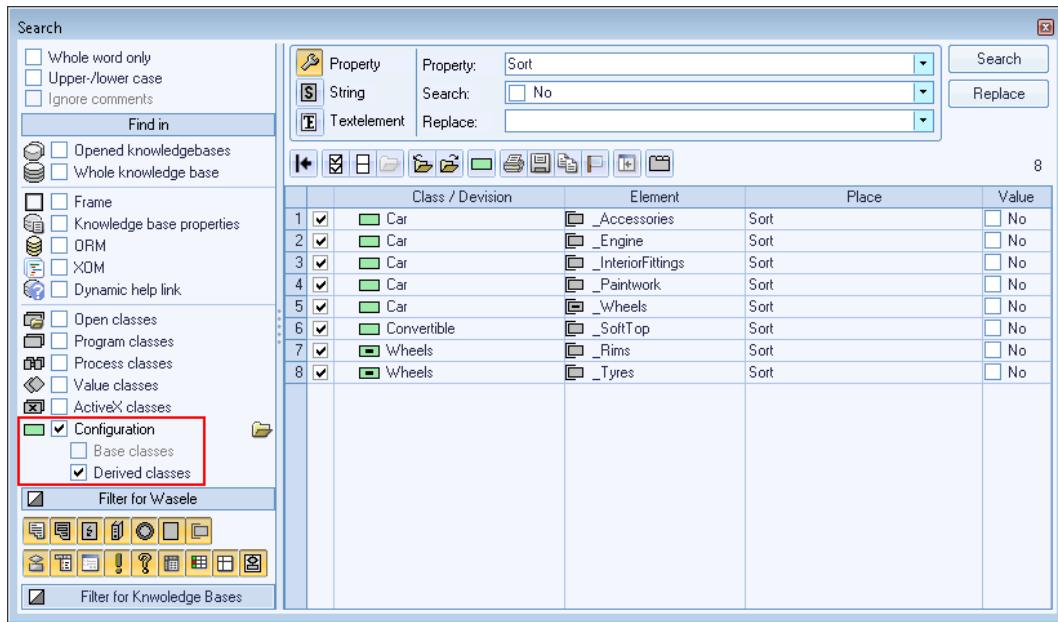


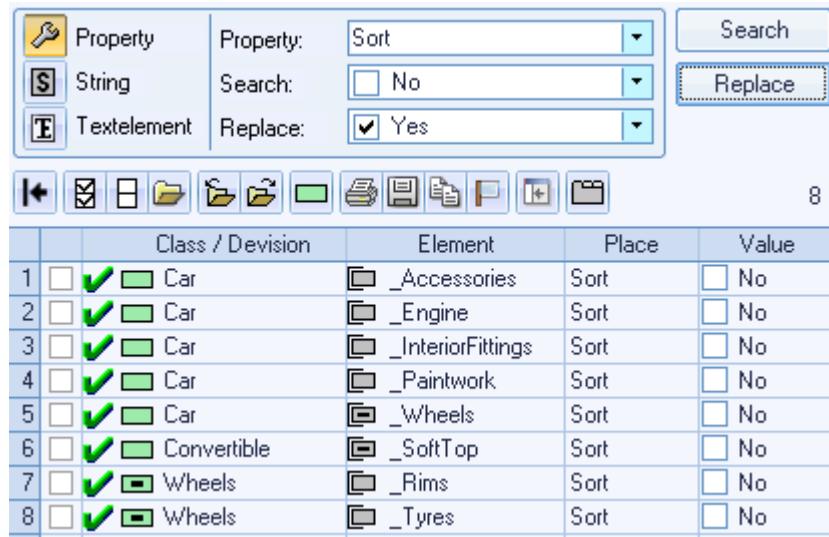
Figure 32: Search for not set property “Sort”

In the lower table in the dialog all classes are listed that contain components without *Sort*. Child classes that have only inherited components are not listed.

 In order to set the value of the option *Sort* for all found components from 0 to 1, you activate the switchbox in the first column (Icon) of the table.

Then you select Yes in the combobox *Replace* and press the button *Replace*.

If the replacement was successful, a green flag is displayed in the second column of the table. In this case the option *Sort* was activated for all components in the classes *Car*, *Wheels* and *Convertible*.



The screenshot shows a software interface for managing a component tree. At the top, there are search and replace fields for 'Property' (set to 'Sort'), 'String' (searched for 'No'), and 'Textelement' (replaced by 'Yes'). Below these are standard file navigation icons. A table lists components across four columns: Class / Devision, Element, Place, and Value. The 'Place' column contains 'Sort' for all entries. The 'Value' column contains 'No' for most entries, except for 'SoftTop' which has 'Sort'. The table rows are numbered 1 to 8.

	Class / Devision	Element	Place	Value
1	Car	_Accessories	Sort	No
2	Car	_Engine	Sort	No
3	Car	_InteriorFittings	Sort	No
4	Car	_Paintwork	Sort	No
5	Car	_Wheels	Sort	No
6	Convertible	_SoftTop	Sort	No
7	Wheels	_Rims	Sort	No
8	Wheels	_Tyres	Sort	No

Now you start the application and check if the sorting is correctly displayed in the component tree.



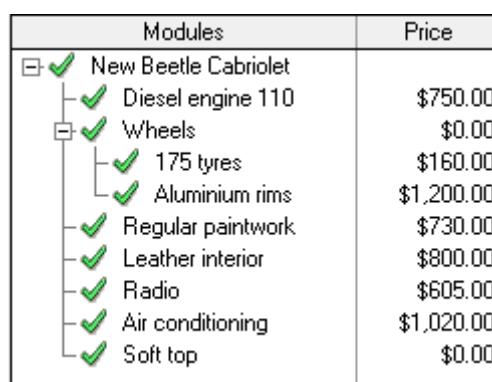
13.5.2. Overloaded component sorting

The in class *Car* defined component sorting is inherited to all child classes. This is visible through the blue colored icon e.g. in the class *Golf*.

In order to carry out another sorting in a derived class or to put components in the inherited sorting that exist only in this class, you can overload the inherited sorting. The procedure is analog to the original definition: in the derived class the *Local sorting for components* is activated and the sorting is carried out via the (now again black) icon .

If the overload is carried out in a class that has in turn child classes, e.g. *Closed*, the overloaded sorting is inherited.

The following figure shows the object structure of the model *Beetle*. The object *SoftTop* is displayed at the last position, because the component *_SoftTop* was not yet sorted or because it is sorted in at the last position due to its name.



The screenshot shows a component tree for the model *Beetle*. The tree structure is as follows:

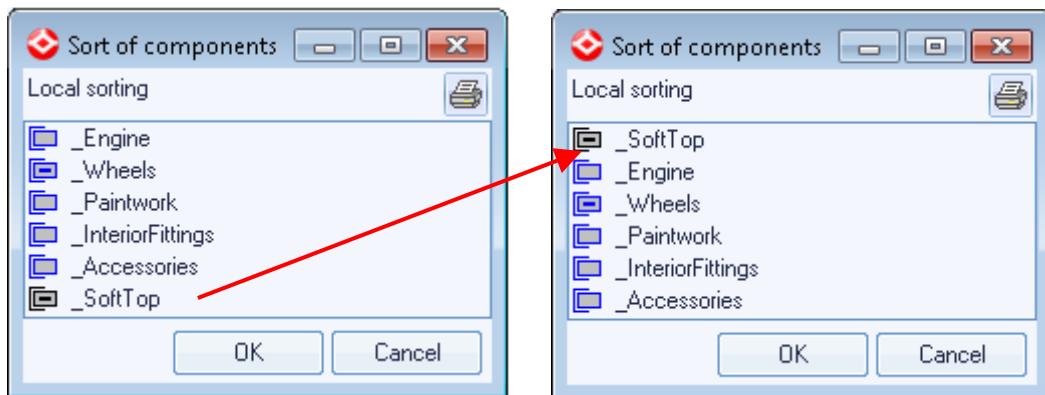
- New Beetle Cabriolet
 - Diesel engine 110
 - Wheels
 - 175 tyres
 - Aluminium rims
 - Regular paintwork
 - Leather interior
 - Radio
 - Air conditioning
 - Soft top

The table lists the components and their prices:

Modules	Price
New Beetle Cabriolet	\$750.00
Diesel engine 110	\$0.00
Wheels	\$0.00
175 tyres	\$160.00
Aluminium rims	\$1,200.00
Regular paintwork	\$730.00
Leather interior	\$800.00
Radio	\$605.00
Air conditioning	\$1,020.00
Soft top	\$0.00



In order to change this, you activate the *Local sort* for *components* in the class *Convertible* and position the *_SoftTop* at the desired position, e.g. at the first one.



Then the option *Sort* has to be activated on the component *_SoftTop*.

If the setting of the option *Sort* was carried out via the property search (see chapter 13.5.1), the option should be already activated.

13.6. Repetition

- Properties of objects can be displayed and changed in component tree columns. Before the cause variable of a component tree column can be defined, you have to specify in the field *Column class* from which class the objects originate that have this cause variable.
- Via the system feature *SubTreeOpen* you can control the opening or closing of the nodes in the component tree.
- If you want to change the order of the components in the object tree, the *Local sorting for components* in the class editor is activated. The manual sorting is not displayed in the structure tree. Additionally the option *Sort* has to be activated for the affected components.

14. Color of the paintwork

14.1. Intention

The application already provides quite a number of selection possibilities to configure a car model according to your desires. Which other specifications are important for an order?

It is possible to select the type of paintwork, but the color of the paintwork is not yet considered. Therefore the user should have the possibility to select between the paintwork colors blue, red, green and black.

To do so, you have to create another feature and create multilingual string constants for the colors. You will also get to know the form element configurationbox feature.

14.2. Decision: feature or class?

Before you can integrate the new property “Color of the paintwork” in the application, the question arises if the color is an own module of the car and therefore must be realized via classes and components. Or is the color rather a property that describes the module *Paintwork* more precisely?

In this case the paintwork color has to be definitely considered as a property of the paintwork and not as a separate object, because a color is not an object that has e.g. a status or that can carry out operations.

You cannot always make a decision that a property is imaged as a feature or a class due to the statement “is an/not an object”. Especially with sections of the application that are not concretely dealing with the product that has to be configured, e.g. database interaction or text maintenance environment, the required properties are often so abstract that they can be hardly identified as property or object.

In these cases it is often useful to know if the property contains any further information. If the starting of the car configurator has to be e.g. carried out via a login, an user maintenance has to be created. The question how the users are imaged can have two answers:

1. Via a feature list which contains all the names of the users that are allowed to login.
2. Via a list of objects of a class “User”. This way each object can contain further information such as name, first name, department, access authorizations and of course the password of each user.

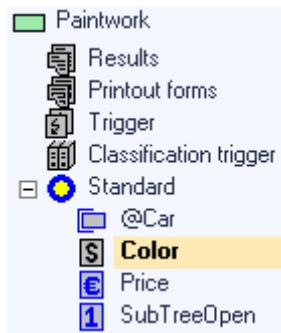
Therefore there are no definite rules that have to be kept with the decision “Feature or Class?”. With regard to the paintwork color the selection came to the feature, because except of the color name no further data (price, article number, etc.) is needed.

14.3. Practice: Create the feature Color

In the *MainForm* the possible paintwork colors blue, red, green and black should be offered for selection. Since the color can only be configured for the paintwork, the feature for the color is created in the base class *Paintwork*.



Create a new string feature “Color” in the class *Paintwork*.



The user should not enter the color of the paintwork as he likes, but he should select it from a predefined range of colors. With the module components, this “predefined range” was predetermined through the derived object classes of the component class; they only had to be applied in the structure tree.

With features the procedure is similar, but there are no predefined values. The paintwork colors (blue, red, green and black) have to be defined manually as values of the feature *Color*.

Since the application is bilingual, you must create the values (colorname) in German and in English.

Create a new group “Colors” in the textmanager. Create therein four new Textelements and enter the English and German colorname.

Textelement	Value	English	German
black	black	black	schwarz
blue	blue		
green	green		
red	red		

These four colors are the possible values for the feature *color*. But how can they be assigned? By using the *context menu of the value list -> New* you can create new values, but there is no way to access Textelements.

The use of the Textelement as a value is generally not possible. Therefore a constant has to be used which contains the Textelement (see chapter 8.6.1). The constant, can be assigned as value to a feature.

Create a new constant of the type string with the name „ColorGreen”. At constant editor, choose the toolbar icon  to bring the constant in the Textelement mode. Now, the Textelement °green° can be assigned to the constant as content. For this you type either “green” in the naming field, after that the Textelement °green° is detected and assigned. Or you open the Textmanager with the icon  left of the naming field and select the desired Textelement here.

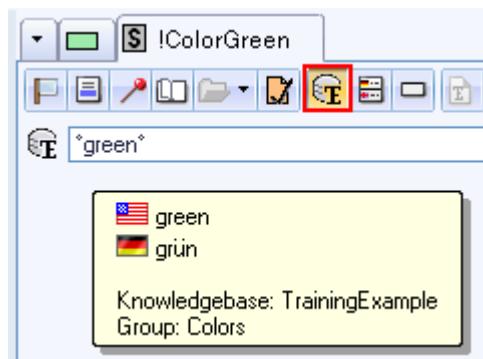


Figure 33: Integrate Textelement in String constant

Create string constants for the colors “blue”, “red” and “black” in the same way



14.4. Practice: Create and assign feature values

In the next step the string constants of the feature *Color* are assigned as values.

Open the feature *Color* of the class *Paintwork*. In the value list (upper left section of the editor) you select the context menu item *New...*.



In the dialog *Edit potential value* you can define the possible values of a feature. As with the init value of a feature (see chapter 10.3.2.2), you can select between *Static value*, *Expression* and *Constant*.

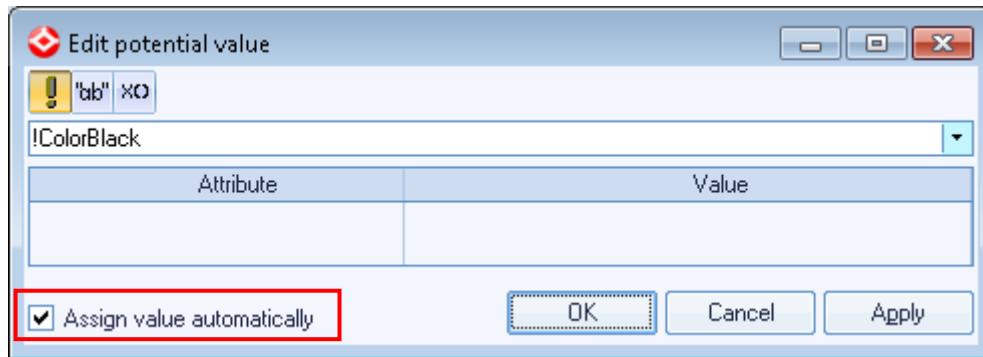


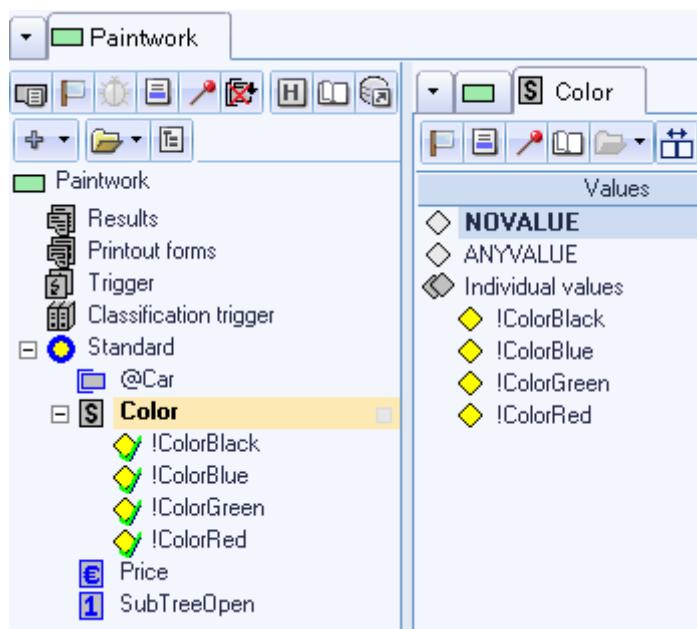
Figure 34: Feature editor - Editing values

Activate the button *Choose constant...* to define a constant as value. Select the constant *!ColorBlack* from the drop down list and confirm the selection with *OK*.

In the dialog *Edit potential value* you can now see the selected constant. If you activate the option *Assign value automatically*, the value is automatically applied to the structure tree with the confirmation. Otherwise you have to do this manually via a double click.

Add the three other constants to the value list and apply all values in the structure tree.

Tip: If you confirm the creation of the individual values via *Apply*, you do not have to reopen the dialog for each new value.



You can also drag the string constants via D&D from the wasele list into the value list. This way, the values are not automatically applied as values in the structure tree.

14.5. Configurationbox feature

In order to display the paintwork colors on the form, the form element **configurationbox feature** is used. This works similar to the configbox components, i.e. the configbox displays all assigned values of the feature that is entered as cause variable.

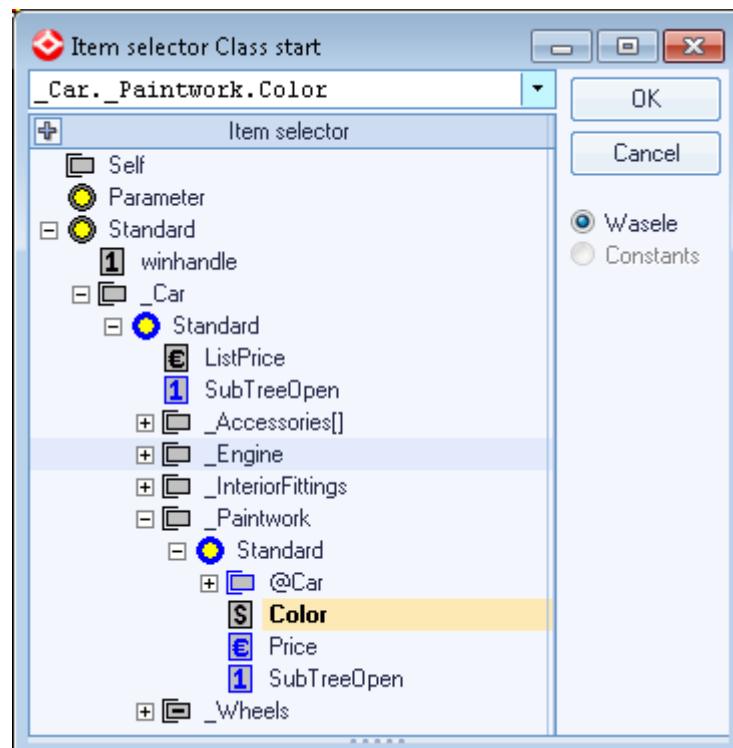
The configbox feature does not have a naming- or a name column; the displayed text corresponds to the value of the assigned values.

14.5.1. Practice: Extend the form by a configurationbox feature

Open the *MainForm* in the class start. Create a *Configurationbox Feature* via the context menu of the form root.



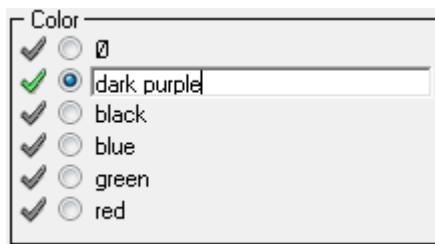
In the field *Cause variable* you click on the icon , select the feature *Color* under the component *_Car._Paintwork* and confirm with *OK*.



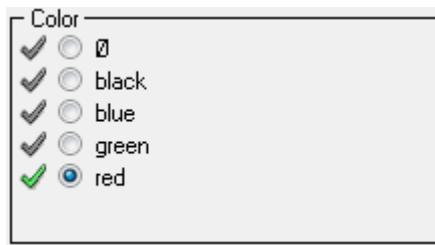
Position the configbox feature at an empty position and adjust size and design. Start your application and test the behavior of the configbox feature.



As in the configbox component the user is able to choose between the four colors. In addition a naming field appears. This allows the user to input a free definable text.

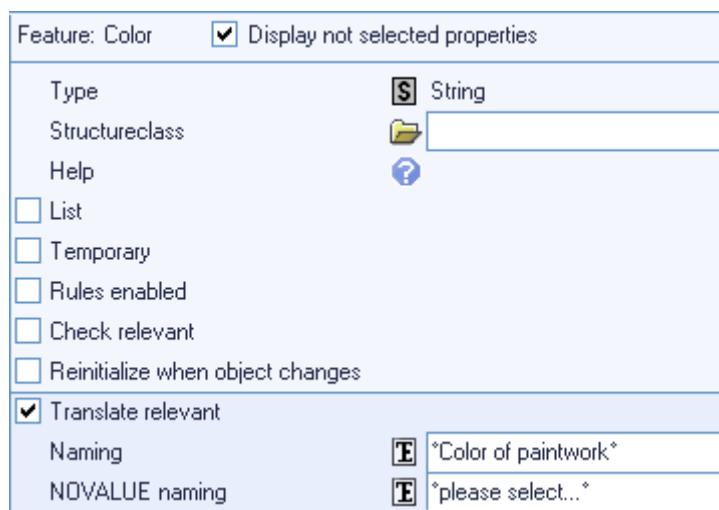


Set the option *Selection only* in the properties of the configbox feature. Therefore the user can only choose one of the assigned values.



Namings can also be defined for features.

Activate the option *translate relevant* in the editor of the feature *color* and define a feature- and a NOVALUE naming.





14.5.2. Notes for the form element configbox feature

If you check the configbox for the paintwork during runtime, the following questions might possibly occur:

Why is the configbox for the color blank sometimes?

As soon as a car model is selected, all configboxes are filled with the possible values. Only the configbox for the paintwork color remains blank. Why?

Paintwork		Price
<input checked="" type="checkbox"/>	<input checked="" type="radio"/>	please select...
<input checked="" type="checkbox"/>	<input type="radio"/>	Regular paintwork \$730.00
<input checked="" type="checkbox"/>	<input type="radio"/>	Metallic paintwork \$980.00

?	

The question has the same answer with regard to logical as well as programming reasons: because the paintwork does not yet exist. The color is a property of the paintwork. As long as no paintwork was selected (NOVALUE is selected), a paintwork object does not exist and therefore paintwork properties do not exist.

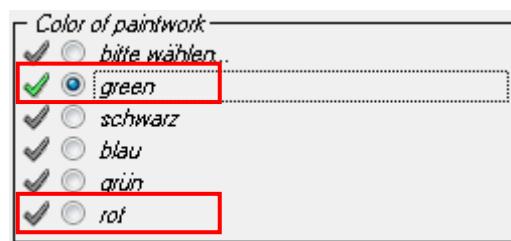
As soon as the component *_Paintwork* is allocated with a value, the configbox for the color is filled:

Paintwork		Price
<input checked="" type="checkbox"/>	<input checked="" type="radio"/>	please select...
<input checked="" type="checkbox"/>	<input type="radio"/>	Regular paintwork \$730.00
<input checked="" type="checkbox"/>	<input checked="" type="radio"/>	Metallic paintwork \$980.00

Color of paintwork		
<input checked="" type="checkbox"/>	<input checked="" type="radio"/>	please select...
<input checked="" type="checkbox"/>	<input type="radio"/>	black
<input checked="" type="checkbox"/>	<input type="radio"/>	blue
<input checked="" type="checkbox"/>	<input type="radio"/>	green
<input checked="" type="checkbox"/>	<input type="radio"/>	red

Why is the currently selected color always displayed twice?

If you click on a value, the focus jumps in the first line and then the value appears twice. Why?

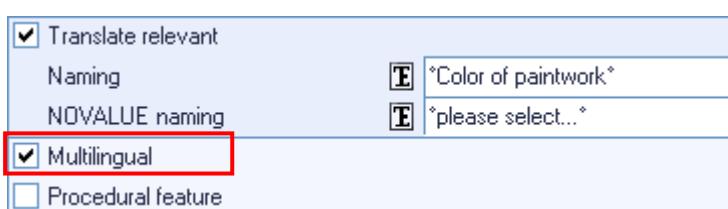


This phenomenon can be explained with the multilingualism of the values. All color names were entered in multilingual string constants in order to be able to display the correct descriptions also for the dialog language German.

The feature itself was actually not prepared to take up multilingual values. With the selection of the color (= multilingual string constant) therefore only the value of the current dialog language can be assigned to the feature: "green". Since this (now monolingual) string is unequal to the multilingual string constant {"green", "grün"}, the current feature value as well as the multilingual string constant is displayed.

The problem is recovered with declaring the feature *Color* as multilingual.

Open the feature *Color* of the class *Paintwork* and activate the option *Multilingual*.



Start the application and test the behaviour again.

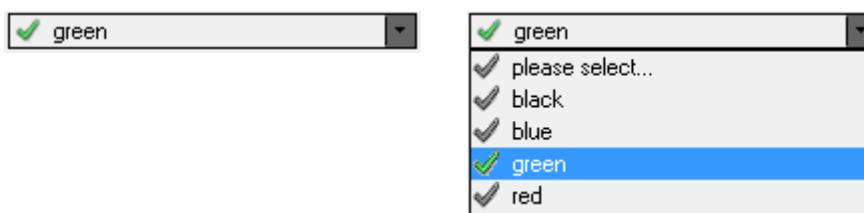
Workbench/Structure Tree/Features/Properties/Multilingual

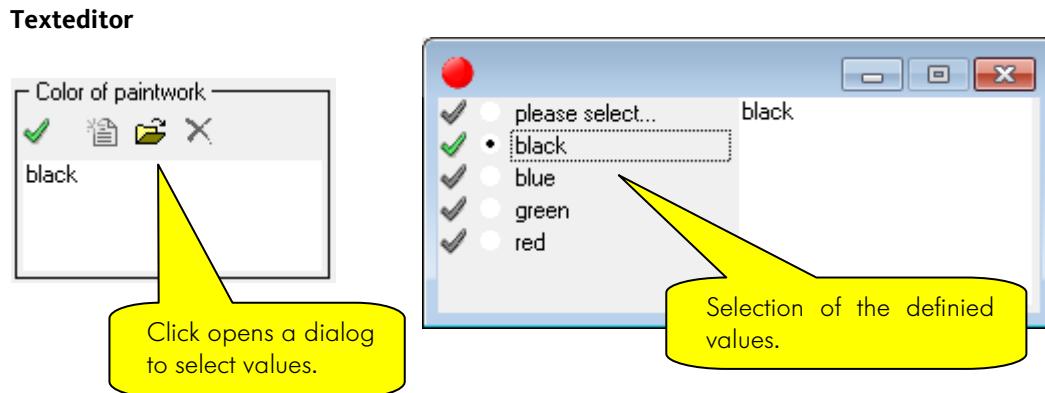
14.6. Tips & Tricks

14.6.1. Representation options for the configurationbox feature

In addition to the by default used representation type *List* there are two further display types:

Compact





14.7. Repetition

- The form element configurationbox feature is used to switch between any feature values.
- The possible values of a feature are created manually in the value list in the feature editor. With this you can select between fixed value, expression and constant.
- A configbox feature does not have a naming column; always the data of the assigned value is displayed as naming. In order to realize multilingual values, e.g. color names in English and German, multilingual string constants are used.
- If multilingual constants are used as values of a feature, the option *Multilingual* has to be set in the feature editor.

15. Power of the engine

15.1. Intention

The approach from chapter 10 to differentiate the components not only by the name but additionally by characteristic properties is continued in this chapter. Which property is important for the user and which is not?

Example engine: The user is usually not very interested in the amount of the individual parts, the piston diameter or the total weight of the engine. The power of the engine is much more interesting. Therefore this detail should be displayed on the form as additional information to the engine.

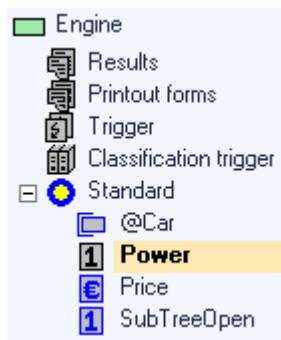
In the following chapter you will repeat how to create, initialize, and overload a feature and how to display it in a configurationbox column. Since identification numbers such as the engine power are generally specified in a certain unit (in this case HP), you will also learn how to define and use units of quantity.

15.2. Practice: Create the feature Power

The engine is the only component for which the property Power is useful. Therefore this feature has to exist only in the class *Engine* and the derived object classes.

Create the numerical feature “Power” in the class *Engine* and initialize it with the value 0.

Activate the option *Reinitialize when object changes*, see chapter 10.4.



15.3. Practice: Define units of quantity

Depending on the selected dialog language, the power of the engine should be displayed in the country-typical unit. If the application is operated in English, the engine power is specified with kW. In the German dialog language the display shows HP (PS).

In order to convert this requirement, you have to work with **units of quantity**.

In camos Develop, **units of quantity** (short: UOQ) are administrated in the frame of the knowledge base. They are used to allocate numerical values with certain (measuring) units.

Any amount of units of quantities is possible. The units of quantity are combined in groups and they are allocated with conversion factors that refer to a certain reference unit of quantity, the so-called Master UOQ. A group "Lengths" e.g. exists with the reference unit "Meter" and the units of quantity "Kilometer" (conversion factor referred to "Meter" = 1000) and "Centimeter" (conversion factor = 0.1).

In some cases additionally an *Offset* has to be specified, e.g. if a conversion from the temperature unit "Fahrenheit" to "Celsius" has to be carried out. In this case the *Factor* is 1.8 and the *Offset* is 32.

The allocation of the unit of quantity is carried out in the wasele- or class editor. Depending on the dialog language another unit of quantity can be assigned. In this case an automatic conversion between the set units is carried out with switching the language during runtime.

First the units of quantity "kW" and "HP" (PS) are defined and assigned to the feature *Power*.

 Go to the *Frame maintenance* and open the *TrainingFrame*. Switch to the navigation area *Units of Quantity*. Call the context menu and select *Reserve*.

Select the menu item *New* in the context menu in the left upper table to create a new UOQ group. The dialog *UOQ Group* is opened. Allocate the description "Power" and the Master UOQ "Kilowatt". Confirm with *OK*.

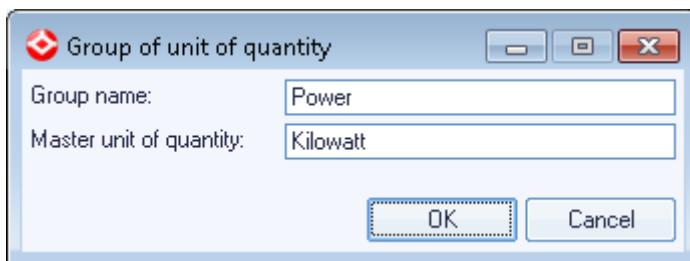


Figure 35: Creating a new UOQ group in the frame

 In the right area of the group *Power* you can now maintain the units of quantity.

In the table *Unit of quantity* enter the unit „kW" that is to be displayed at runtime.

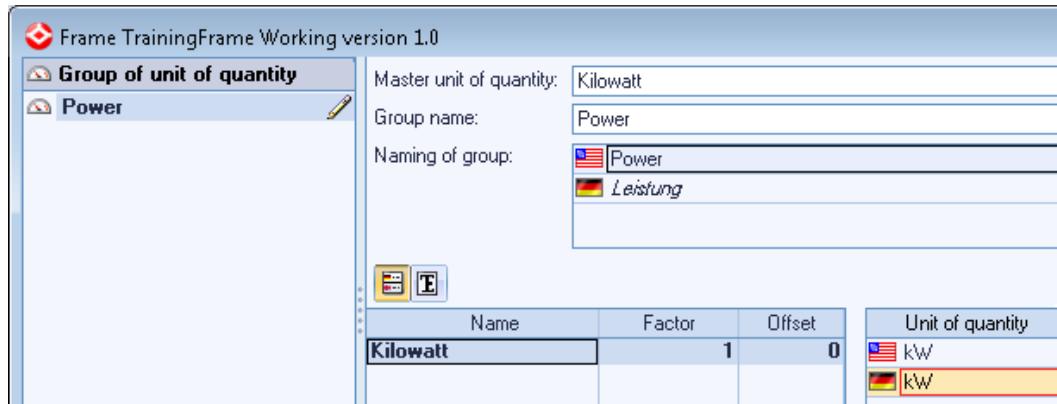


Figure 36: Maintenance of units of quantity in the frame

The *Naming of group* is used primarily for the maintenance and thereby only the translation of the units of quantity into other languages. The entries, that you deposit here, don't influence the representation on the form.

Create a second unit of quantity within the group *Power* by selecting the icon *New* in the lower table. Allocate the name "Horsepower", the factor "1.358", the offset "0" and the namings "HP" for English and "PS" for the German *Unit of quantity*.



Name	Factor	Offset	Unit of quantity
Kilowatt	1	0	HP
Horsepower	1.358	0	PS

Close the frame via the button *Release all and close*.



Because the power can be specified in kW and PS, but they don't have the same factor, the factor 1,358 is specified for the unit of quantity Horsepower. 1 kW is equal to 1,358 kW.

15.3.1.1. Enter the unit of quantity on the feature

The feature does already exist and the unit of quantity also exists in the frame. But currently both are independent of each other. Therefore the desired unit of quantity has to be allocated to the feature *Power*.

If the dialog language is switched to German, the unit of quantity has to be changed from kW to PS. Therefore a unit of quantity is entered for both country codes.

Open the class *Engine* and therein the feature *Power*.

In the column *Unit of quantity* of the feature editor you select the icon . The click opens the assigned units of quantity. Via the context menu *Select units of quantity...* the dialog *Choose unit*



of quantity is opened. Allocate the unit *Kilowatt* to the icon of the United States of America and *Horse power* to the German icon.

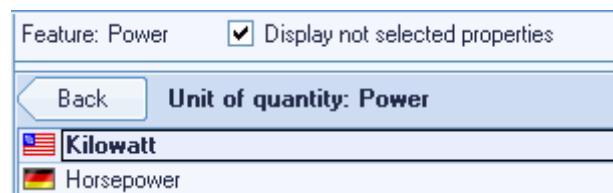


Figure 37: Allocate a unit of quantity to a feature

15.3.2. Practice: Overload the init value of the feature Power

Since the power of the engine was inherited to all child classes, it is initialized in all object classes with 0. In order to allocate a corresponding value to each engine, in the object classes (O50, O120, D70, D110) the init value is overloaded in the feature *Power* and allocated with another value.

The values correspond to the description of the engine, i.e. 50 for the engine O50 etc. The init value is entered for the unit kW, because the set main language is English.

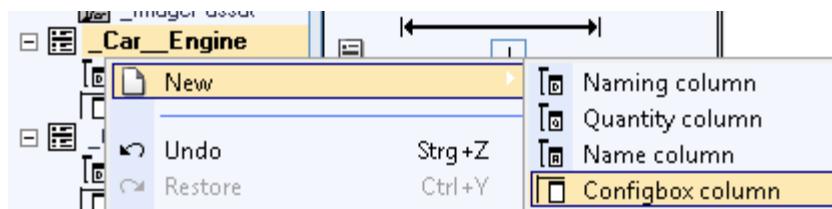
Initialization
1 50

 On all engines you overload the init values with the corresponding power.

15.3.3. Practice: Extend the form by the Power

Finally you only have to extend the form in order to display the power.

 Create a new configbox column on the configurationbox component of the engine.



Specify the feature *Power* from the class *Engine* as cause variable. Enter the headings, change the text alignment, adjust the width of the column and specify the width of the configbox, if necessary.



Specials	
Cause variable	Power

Start your application and test the behavior with the switching of the language.



The conversion from kW to HP(=PS) is carried out automatically with the switching of the language.

Dialog language English

Possible engines		Power	Price
✓	Otto		
✓	Otto engine 50	50 kW	\$350.00
✓	Otto engine 120	120 kW	\$800.00
✓	Diesel		
✓	Diesel engine 70	70 kW	\$330.00
✓	Diesel engine 110	110 kW	\$750.00

Dialog language German

Possible engines		Power	Preis
✓	Otto		
✓	Ottomotor 50	67,9 PS	\$350.00
✓	Otto engine 120	162,96 PS	\$800.00
✓	Diesel		
✓	Diesel engine 70	95,06 PS	\$330.00
✓	Diesel engine 110	149,38 PS	\$750.00

15.4. Tips & Tricks

15.4.1. How to assign units of quantity dynamically

The unit of quantity with which a feature has to be displayed does not have to be necessarily entered in the feature editor. With the use of unit of quantity functions the unit of quantity that has to be used can be dynamically changed during runtime.

 *Function reference/Units of quantities - Functions*

15.5. Repetition

- Units of quantity are administrated in the frame.
- Several units of quantity can be entered to the same group.
- Units of quantity can be bound to a language.
- A conversion to another unit of quantity of the group is possible via entering a factor and/or an offset.

16. Settings and features in the development system

16.1. Intention

Now we take a time out of the development work in order to get to know a few other setting possibilities and features of camos Develop.

In this chapter you will learn how to sort the wasele in groups in the structure tree and in the wasele list. Furthermore the user- and knowledge base options are introduced.

16.2. Group-oriented structure

In the structure tree as well as in the wasele lists groups can be defined to structure the individual wasele e.g. according to type or field of use. By default all classes contain the group **Standard**.

Groups are only visible in tree view. In the list view all wasele are alphabetically sorted without consideration of the group to which they belong. The view of the structure tree and the wasele list is controlled via the icon or in the toolbar.

Definition

In order to guarantee the clarity also in the progressive state of the development, the wasele in the structure tree should now be subdivided into groups.

Open the class *start* and mark the root of the structure tree, the so called structure tree root. Here you select the context menu item *New Group...* and allocate the name “Components”.

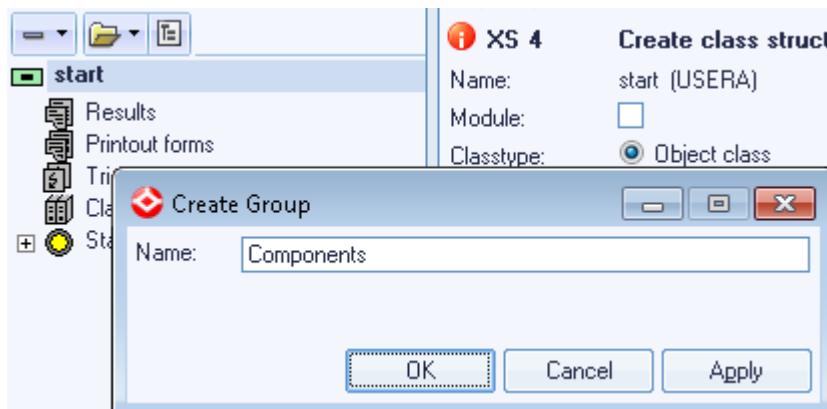
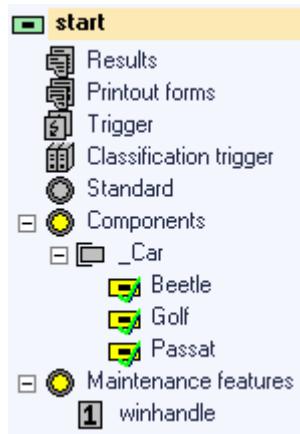


Figure 38: Groups in the structure tree

Shift the component *_Car* via D&D under the group *Components*.

Then you create a new group “Maintenance features” and shift the feature *winhandle* under this group.



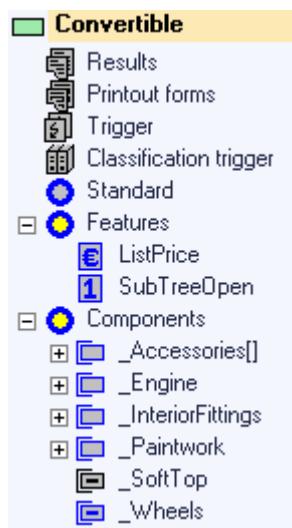


Groups should be also created for all other classes in order to structure the features and components.

💡 Open the class *Configuration* and create two new groups “Features” and “Components”. Shift the feature *SubTreeOpen* under the group *Features*.

The defined groups are inherited to all derived classes and also the allocation of the wasele. Open all *Car*- and *Module* classes and shift the features and components under the respective groups.

Finally the group *Standard* should contain no longer a wasele in any class (no + in front of the node), e.g. class *Convertible*:



The shifting between the groups is principally only possible with locally defined wasele. If you try to e.g. shift the inherited feature *Color* of the class *metallic*, it is not shifted but created as a copy named “*Color_1*” under the other group.

💡 If you delete a group, also all wasele in the group are automatically deleted!

16.3. User options

The surface of the development environment can be affected user-defined at diverse places. E.g. the favored view of the structure tree and the wasele list can be saved as default in the **user options** via the icon  in the toolbar of the structure tree.

The user options are setting possibilities for the view and the behavior of the development environment that are saved with regard to the current user.

The user options can be found in the main menu *Extras*.

The dialog *User options* is subdivided into the following sections:

Classes

Here you can set options for the display of the class tree and the behavior of classes. E.g. if Drag&Drop is allowed in the class tree.

Wasele

Here the options for the handling of Wasele (e.g. Creating / Deleting / etc.) can be defined. The option Ask for deleting a wasele should always be set to prevent the accidental deleting of e.g. a feature with the hotkey .

Structure tree

In this area various view options of the structure tree can be defined. E.g. if namings, initializations and / or debugger values should be displayed.

Wasele list

Here the view options of the wasele list can be defined, e.g. if the size and value of constants should be displayed. .

Miscellaneous

These options affect among others the behavior of the camos Develop, e.g. if the knowledge base and the last opened classes should be reopened when starting the development environment. Or e.g. if the debugger should always be opened in a separate window.

Directories

Here directory paths for certain Open- and Save dialogs can be initialized. The default directory for graphics is e.g. used with the loading of a graphic in the graphic constant editor (icon  in toolbar).

Settings

Among these options *Cycle of saving* has to be emphasized. The number specifies in which intervals camos Develop writes all changes on the knowledge base automatically in the database.

With the default value of 20 seconds a maximum of 20 seconds of programming time would be lost in case of a system breakdown.

Specials

On this tab page e.g. the position of the debug window can be reset, if this window is not reachable anymore e.g. because of changing the display resolution. It is possible to change the password for the login aswell.

 *Main menu/Extras/User options*

16.4. Knowledge base options

In addition to the user-related options also settings on the knowledge base can be carried out which are therefore valid for all users.

The knowledge base options can be opened as follows:

- Main menu Knowledge base -> *Properties*
- Context menu of the Label of the knowledge base name -> *Properties*
- Via the icon  (Properties) in the main toolbar
- Double click on class tree root

There you will find the following sections in the area *Options* of the sector *Basic data*:

Preface

Here you can e.g. define the *Start class*. If you enter the name “start”, you can start the debugger always via the hotkey F5 independent of the currently marked class in the class tree.

Furthermore you can find the link of the knowledge base to the frame here.

Defaults

Here the name of the *Default group* can be defined, which is the group which should always be created.

Documentation

In this section you can make presettings for the development of the developer documentation  and the user help .

Textinputmode

Here can be defined in which form the texts in the knowledge base should be defined. If more than one option (Textelement, Constant and simple expression or Multilingual text) is activated, the developer can define the input mode himself.

Restrictions

The setting *Size of graphics* was introduced to avoid or detect unnecessary big graphics in the knowledge base (the syntax check provides a corresponding error), because these graphics needs a lot of storage place. With a value of e.g. 50 KB (default value) it is therefore not possible to insert graphics with a size of 50 KB via Copy&Paste in the graphic constant editor. In this case you first have to increase the value of *Size of graphics*.

 *Knowledge base/Properties/Options*

16.5. Repetition

- The wasele in the structure tree and in the wasele lists can be structured in groups.
- Groups are only displayed if the structure tree or the wasele list is in the tree view.
- The *user options* can be found in the main menu *Extras*. The here defined presettings are separately valid for each user.
- In the *Properties of the knowledge base* you can set options that affect the complete knowledge base.

17. Introduction to the rule work

17.1. Intention

In the car configurator different connections between the individual components and properties of a car have to be formulated as rules. It is e.g. unreasonable to offer a convertible with a sunroof.

Via rules individual values are assigned, forbidden or allowed in dependency of other elements. The user should be able to recognize why certain values may not be selected at a particular time.

In this chapter you will learn which types of rules camos Develop provides and how you can use them. You will also be introduced to some guidelines that you should consider with the creation of rules.

17.2. Before creating a rule

17.2.1. Formulating rules

The first consideration with the creation of a rule is always how it is formulated. In camos Develop rules follow a simple logic:

<Feature/Component> <Value> **is** <Rule type> **if** <Condition>

Where are rules defined? Rules are created, where they should have an effect on – this means at the value in the structure tree that has to be ruled. Therefore the value has to be applied as an assigned value (via a double click, also see chapter 5.4.2.1) to the structure tree.

Preconsiderations for the example: In the car configurator it is not allowed to order a sunroof when a convertible model is selected. The component is Accessory, the value is Sunroof, the rule type is forbidden and the condition is Car is a convertible. Therefore the rule is:

Accessory Sunroof is forbidden if Car is a convertible.



Since the accessories are defined in the class Car, it is useful to define this rule also in the class Car in the component _Accessories[] on the value Sunroof.

17.2.2. License for the rule work

The most important prerequisite for the rule processing is a valid license. In this example a license of the type camos.Configurator is demanded with the following function call:

```
LicenseDemand ("camos.Configurator");
```

The function returns 1 if the license demand was successful, otherwise 0 is returned.

Without a successful license demand the application is carried out in the development environment, but all values remain allowed.

If the application is operated outside the development system, an error message occurs if no license was demanded 60 seconds after the program start.

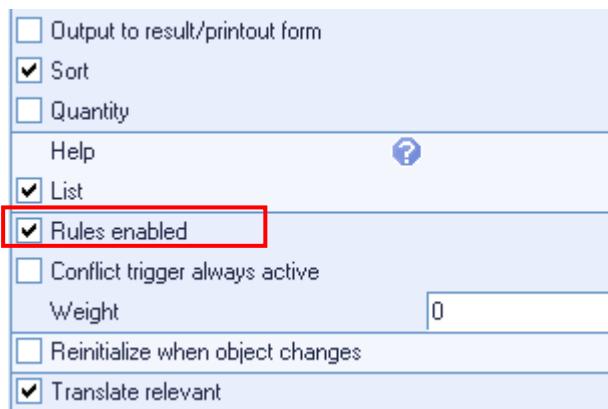


Open the method `New` in the class `start` and add the following call in the first line:

```
LicenseDemand( "camos.Configurator" );
```

17.2.3. The property “Rules enabled”

In order to affect a value of a feature or a component by a rule, the property *Rules enabled* has to be set in the workbench of the respective wasele.



With the setting of the option *Rules enabled* the symbol of the wasele is colored turquoise (), the wasele is automatically check relevant (see chapter 17.2.4). The properties *Rules enabled* and *Temporary* exclude each other.

Only if the property *Rules enabled* is set, rules can be defined for a value.



Open the class `Car` and the component `_Accessories[]`. Set the option *Rules enabled* in the component editor.

17.2.4. The property “Check relevant”

The rule eradication is started, as soon as a check relevant wasele is changed. In order to not carry out the rule eradication with every change of a wasele, wasele are not check relevant by default.

The option *check relevant* should only be activated for wasele, which are used in the condition of a rule. For this reason the performance of the application can be enhanced.



When creating wasele the flag *check relevant* is not set per default. To change this, the property can be preallocated in the *frame* under *Defaults*.

Wasele, which are used in rule conditions, have to have the property *check relevant*.

Wasele with the property *Rules enabled* have automatically the property *check relevant*, therefore this property is faded out in the editor.

17.3. Rule types

In camos Develop there are different rule types that directly affect values of features and components. Principally there are possibilities to forbid, allow or directly assign a value in dependency of certain conditions.

The following rule types are differentiated:

Icon	Name	Description
✓	May	The ruled value is allowed.
✗	May not	The ruled value is forbidden. If the value is displayed, it can still be selected.
!	Must	The ruled value must be selected. Other possible values of the cause variable still can be selected, but they are not allowed.
:=	Assignment	The ruled value is automatically assigned if the condition is valid. As long as the condition is valid, the value cannot be deselected. Assigned values are principally allowed.
─	Assign/Delete	Principally the same as the assignment rule, but the assignment is made undone as soon as the condition is no longer valid.
☒	Invisible/MayNot	This rule forbids the value and fades it out in form elements as far as the value is not selected (contrary to the <i>MayNot</i> rule).
ghost	Invisible	This rule fades out the value if it is not currently selected. With this the value is not forbidden. (E.g. a component is initialized with this value, but it should not be offered any longer as selection possibility).

17.4. Creating a rule

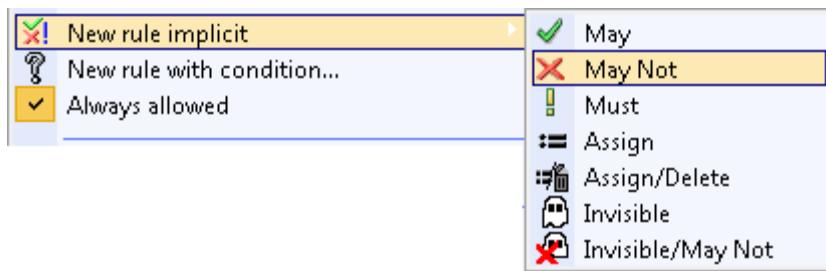
17.4.1. The rule editor

In order to create a new rule, you select the item *New rule implicit* from the context menu of the relevant value. So a submenu is faded in from which you can select the desired rule type.

The rule type determines what should be done with the selected value (see chapter 17.3).

Rightclick on the value *Sunroof* of the component *_Accessories[]* and select *New rule implicit -> May not*.





After the selection of the rule type the graphical rule editor is displayed in the workbench.

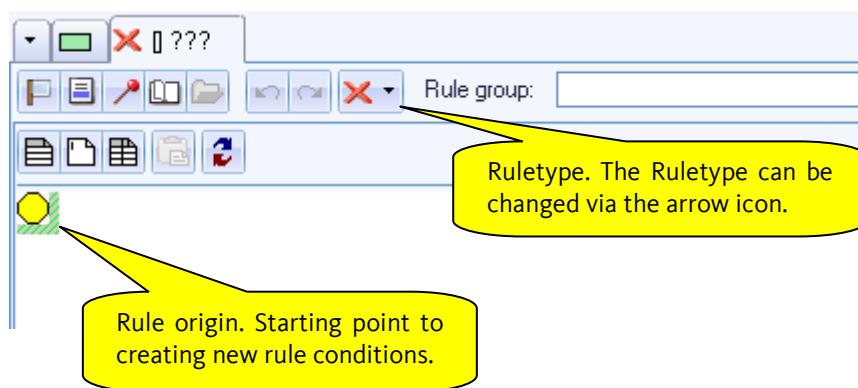


Figure 39: The rule editor

Here you can graphically edit the condition for your rule. First a yellow circle symbol is displayed that represents the rule origin. Out of this origin rule elements can be created. Each element represents a logical statement. Each condition elements provides true (1) or false (0) as a result.

Basics/Rule editor

The rule elements are graphically connected via lines. Linear connected elements are linked with a logical AND while parallel connected elements are linked with a logical OR. With this the system follows the **Boolean algebra**.

The **Boolean algebra** (named after George Boole) defines the properties of the logical operators AND, OR and NOT.

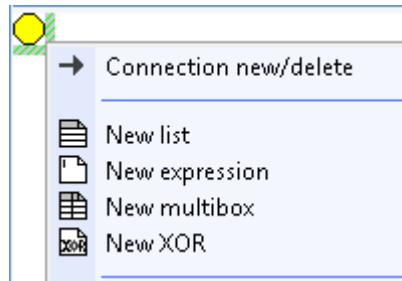
Two logical values that are linked with AND have only 1 as a result if both logical values are 1, otherwise the result is 0.

Two logical values that are linked with OR have only 0 as a result if both logical values are 0, otherwise the result is 1.

NOT reverses the result of a logical statement: 1 becomes 0 and vice versa.

These operators can also be chained and nested with brackets.

Via a rightclick on the rule origin you get a context menu with which you can create new condition elements.



For now we just want to have a look at the first two elements.

- Via *New list* features and components for the condition can be selected and allocated with values.
- *New expression* allows you to write a logical expression that can contain also simple method- and function calls. "Simple" means here that the function/method may not carry out assignments. With self-defined methods the property *allow side effects* has to be disabled, see chapter 20.4.2.2 and Figure 51.

The graphical connection between the rule origin and the new created rule element is automatically created.

If further rule elements are required (if the rules consist of several partial conditions), a further element can be defined on the rule origin or already existing rule elements via the context menu item *New list* or *New expression*. This is connected with the element on which the context menu was called. In this way elements that are linked with AND or OR can be created.

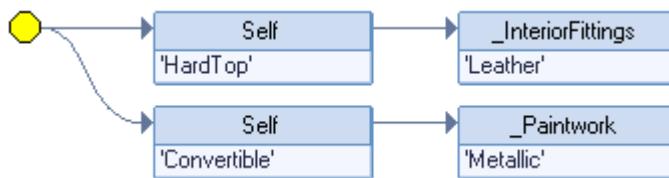


Figure 40: Creating linked rule conditions

After the creation, the rule is added in the structure tree as a subordinated element of the assigned value: the rule type as well as the rule expression that results from the graphical rule maintenance is displayed.

For our example you open the context menu on the rule origin and select *New list*, then the item selector opens.



Select the wasele that has to be in the condition of the rule. The condition is "if Car is a Convertible". Therefore you need the component _Car. Since the rule is defined in the class Car itself, you select *Self* and click on *OK*.

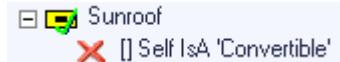


With this a first element of our condition is created. It accesses the wasele *Self* in the context of the class *Car* (therefore to the car itself). In the next step is specified which values complete the condition.

This example shows that it was useful to introduce the class *Convertible*, because this rule would be also necessary for other convertible models.



Double click on the value *Convertible*, then "Convertible" is entered in the rule element. Therefore the value in the structure tree gets the following rule:

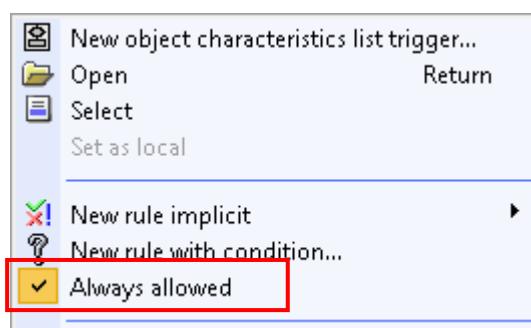


Now you test the behavior of the application. Select the Beetle and the value *SunRoof* from the accessory box. The sunroof gets the symbol **X** to indicate that it is not allowed to be selected under these prerequisites.

17.4.2. Always allowed?

For each value you have to determine whether it is initially allowed or forbidden.

If you carry out a right click on an assigned value in the structure tree, you can set the default condition of the value in the context menu with the menu item *Always allowed*.



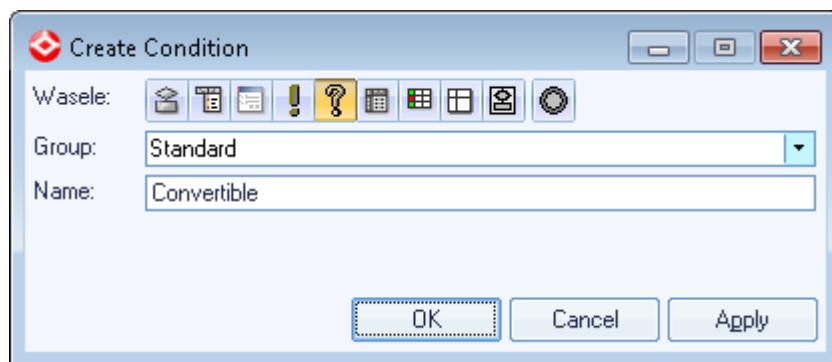
This condition is visible on the symbol of the value in the structure tree. If this symbol is overlapped with a green flag (), the value is *Always allowed*. If the green flag is missing () or the value is not *Always allowed*.

Each new applied value is at first *Always allowed*.

If a value has to be allowed for only one condition, the *Always allowed* flag has to be removed (then the value is by default forbidden) and a *May* rule has to be formulated.

17.4.3. Conditions

With the creation of a configuration logic it can happen that there are many rules that have the same condition part. In this case you can define a wasele **conditions** in order to be able to formulate the condition part independently from a specific rule. Later, you can assign the condition to a value in rules via the context menu item *New Rule with condition* of the assigned value.



The wasele conditions are administrated in the wasele list. Names of conditions principally begin with a question mark, e.g. `?Convertible`.

A condition is defined with the same graphical rule editor that is used with the implicit rules.

In order to use the condition with a rule on a value, you use the context menu item *New rule with condition* of the value. In the following dialog all conditions that are defined in the current class are listed. Also you can choose the desired rule type here.



Figure 41: Creating a rule with condition

The behavior of a rule with condition in the interpreter is identical to implicit rules. But using conditions the maintenance of the rules can be simplified, because for several rules the condition has to be maintained at only one position.

17.4.4. Practice: Creating rules

In the framework of this practice certain colloquially formulated rules for the car configurator should be changed to a form that the system can understand. There are several possibilities to achieve this.

The rules to realize are as follows:

1. The diesel engines are only allowed for vehicles with a hard top.
2. For the two weakest engines wheels with a tyre width of 205 are forbidden.
3. For the convertible models the Sunroof and roof baggage carrier are not possible.
4. The CD changer is only available in connection with a radio.



Now set the conditions according to the information in the car configurator.

In the following an exemplary possibility is described how to use the rules in camos Develop.

This can deviate from your solutions, but it does not mean that yours are wrong. Depending on the main emphasis the one or the other solution can have advantages or disadvantages.

1. The diesel engines are only allowed for vehicles with a hard top.

This would mean that the component that has to be affected is "Engine", the value is "Diesel", the rule type is "allowed" and the condition is "Car is a hard top".

"*Engine Diesel engine is allowed if car is a hard top.*"

Where is this rule defined? In the component *_Engine*, this component is in the class *Car*. Here all values were already applied as *always allowed*.

Open the component *_Engine* in the class *Car*, check if the property *Rules enabled* is set and set the property, if necessary.

Apply the value *Diesel* via a double click and remove the property *Always allowed* from the applied value. Now you create an implicit *May* rule (right-click on *Diesel* -> *New rule implicit*) to the newly applied value.

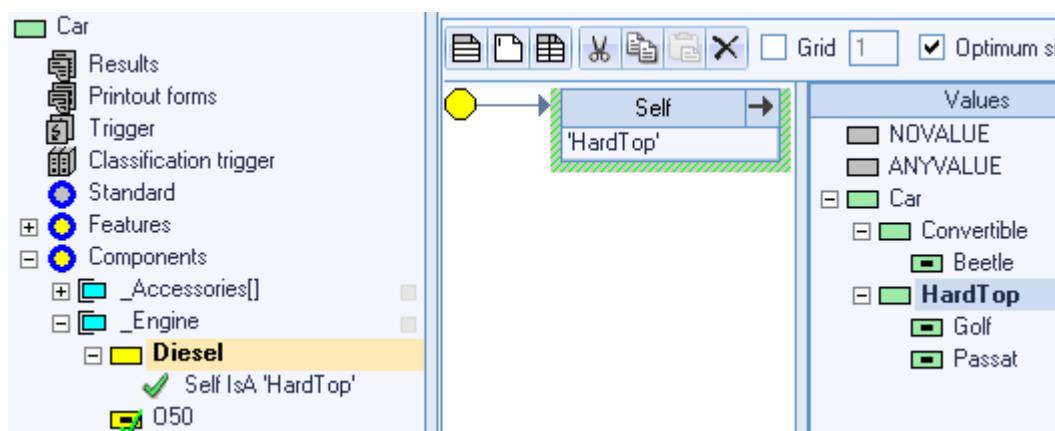
Via a right-click on the rule origin (the yellow circle) you create a new list element which gets *Self* as cause variable. Self, because *Car* is a part of the condition and you are already in the class *Car*. Via a double click you add the value *HardTop* to this list.

When you now start and test the configuration, you will find out that diesel engines can still be selected for convertible models. Why?

This is due to the weighting of the different rule statements. Principally the following determinations can be made:

- Rule statements on concrete object classes have the priority to rule statements on more abstract base classes.
- May not rules have the priority to May rules.
- Rules have the priority to “(not) always allowed”.

If you delete now the two concrete values D70 and D110 from the list of the values in the structure tree, the new rule is active with regard to diesel engines.



2. For the two weakest engines wheels with a tyre width of 205 are forbidden.

What does that mean in our rule language?

“Tyre 205 is forbidden if engine is O50 or D70.”

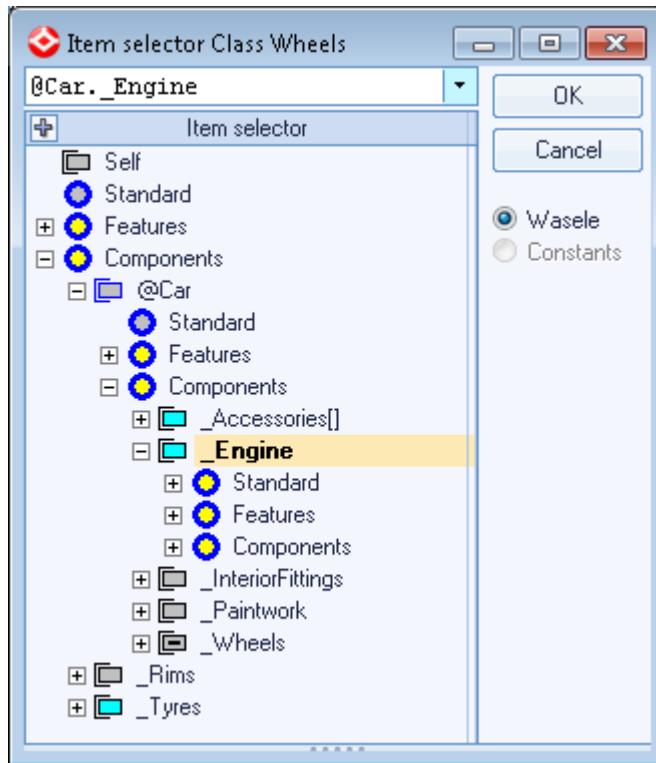
The component `_Tyres` is however not defined in `Car`, but in the class `Wheels` which is in turn used in the component `_Wheels` in the class `Car`. This rule can also be formulated more expressly:

“Wheels with tyres 205 are forbidden if the car has the engine O50 or D70.”

Therefore we want to forbid the value 205 of the component `_Tyres` of the class `Wheels`.

Open the class `Wheels` and the component `_Tyres`. Check if the property `Rules enabled` is set and set this property, if necessary. Add an implicit `May not` rule to the value 205 in the structure tree. Create a new rule element `List` on the rule origin. In the following Item selector dialog you open the predecessor component `@Car` and select the component `_Engine`.





Confirm this with **OK** and assign the two engines O50 and D70 in the values via a double click.

3. For the convertible models the Sunroof and roof baggage carrier are not possible.

This means as rule:

"Accessories Sunroof and Roof baggage carrier are forbidden if Car is Convertible."

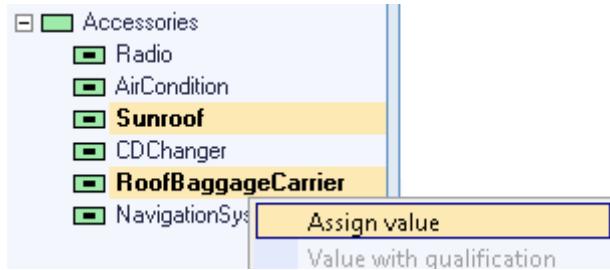
Open the component `_Accessories[]` in the class `Car`. Now you have three possibilities:

- define the same implicit rule on both values `SunRoof` and `RoofBaggageCarrier`
- define a condition with the statement "`Car is convertible`" and creating the same rule with condition on both values
- create a new combined value that is put up with the rule

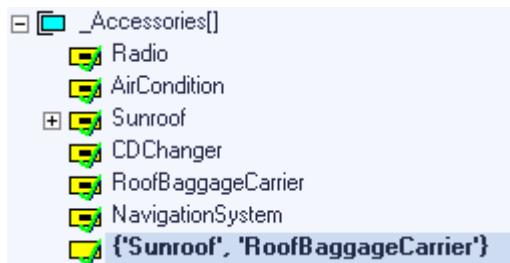
The first two possibilities are okay, but we concentrate on the realization of the third option.



To do so, you select the two values `SunRoof` and `RoofBaggageCarrier` (multiple selection via pressed Ctrl-button) in the list of values in the component editor. Select `Assign value` from the context menu of the values.



The list in the structure tree should now look like this:



Now you create an implicit rule *Invisible/MayNot* for the newly created combined entry.

Now you follow the known pattern: create a new element on the rule origin, select *Self (= Car)* as cause variable and apply *Convertible* via double click from the value list to the rule element.



Test the behavior of the *Invisible/MayNot rule* in your application.

Both entries are faded out from the list if a *Convertible* was selected and they appear again as soon as a hard top vehicle is selected.

4. The CD changer is only available in connection with a radio.

This means that the radio has to be automatically selected if the CD changer is selected:

„Accessory Radio is assigned if Accessory is CDChanger”

When using assignment rules you should always consider how the value should behave if the condition is no longer valid. Should the radio remain even if the CD changer is deleted again? Or should the radio then be automatically deselected?

In this case, we use an *assign/delete* rule in order to have the assignment automatically deleted as soon as the condition becomes false.

Open the component *_Accessories[]* in the class *Car*. Create a new implicit rule “*Assign/Delete*” to the value *Radio* in the structure tree.



On the rule origin you create a new list with the cause variable *_Accessories[]* and select the value *CDChanger* via double click.

With a list this means that the condition is valid if the element CD changer is contained in the list.



Test the behavior of the accessories if the radio is automatically assigned via the `CDChanger` and if you already selected the `radio` by yourself.

With the accessories you can see that the symbols in front of the values change to exclamation marks if no accessories are selected. Since accessories should be optional, NOVALUE has to be also applied in the list of the valid values.



Apply the value `NOVALUE` via double click to the list of the valid values of the component `_Accessories[]` and test the application again.

17.4.5. Condition element Multibox

Define a rule that represents the following statement:

"If for a passat the engine 0120 or O50, alloy rims, tyres 185 and metallic paintwork is selected, the paintwork hast to be black."



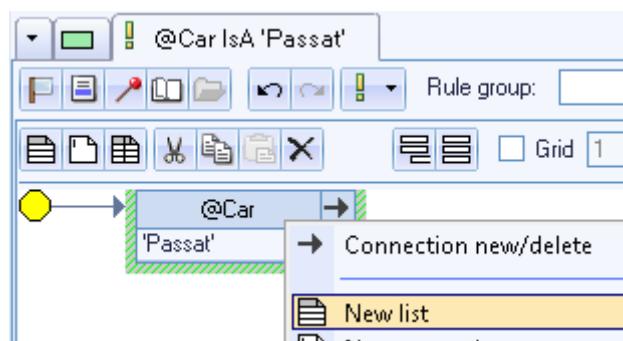
Open the class `Paintwork`, there the feature `Color` and activate the option `Rules enabled`. At the assigned value `!ColorBlack` create a new Must-Rule.

The rule condition consists of five parts, that have to be linked with AND or OR.

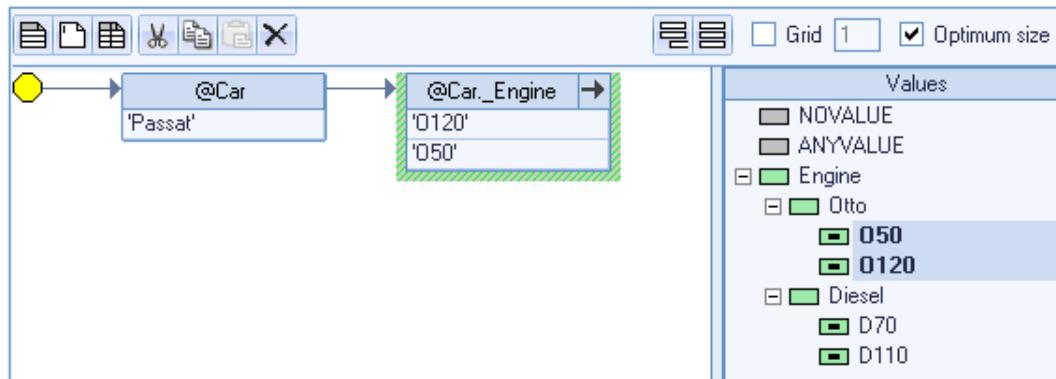


Create the first part of the condition with a condition element list: `@Car IsA „Passat“`.

The second part of the condition is related to the engine. This part has to be linked with the first part of the condition by a AND-connection.



Select the first condition element and call the context menu `New list`. Insert `@Car_Engine` as the cause variable and doubleclick on the values `O50` and `O120`.



By adding a condition element to an existing condition element an AND-connection is created. This means that both parts of the condition have to be true, that the rule acts.

If the second condition is created from the rule origin this leads to an OR-connection. In this case only one condition has to be true, that the rule acts.

Create similarly three further parts of the condition. Notice that all parts of the condition are linked with AND.



Carry out the syntax check.

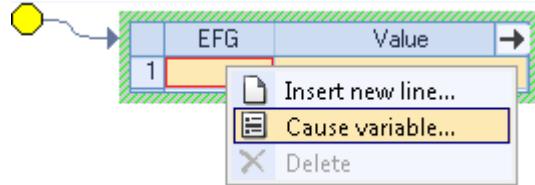
The syntax check reports that the components _Rims and _Wheels are not check relevant.

Activate the property *check relevant* for the components _Rims, _Wheels and _Paintwork. Test the consequences during runtime.

During the work with rules, that consist of lots of conditions, the rule editor quickly becomes confusing. Relief is possible with the condition element *Multibox*, in which as many conditions as you want can be combined. The *Multibox* consists of a table with two columns: cause variable and value. Each row represents a part of a condition, that are all linked with AND.

Delete the existing parts of the condition. Tip: Drag a frame with pressed mouse-button around the five condition elements and click on the Delete-Icon in the toolbar.

Now create a new *Multibox* from the rule origin. Via the context menu of the column cause variable you can assign the cause variable to the row (= part of the condition).



Select the cause variable @Car and doubleclick on the value Passat to assign the value to the column. Notice: are there several values part of the condition, they have to be selected via Ctrl+Click and then be assigned via the context menu item Assign value.

	EFG	Value
1	@Car	'Passat'

Deal with the other parts of the condition in the same way. Tip: Is the required cause variable located in the actual class, it can be dragged per D&D into the multibox.

	EFG	Value
1	@Car	'Passat'
2	@Car._Engine	in ('O50', 'O120')
3	@Car._Wheels._Rims	'AluminiumRims'
4	@Car._Wheels._Tyres	'185'
5	Self	'Metallic'

On the tab page *Expression* you can see the components of the rule expression:

(@Car IsA 'Passat' and @Car._Engine IsA {'O50', 'O120'} and @Car._Wheels._Rims IsA 'AluminiumRims' and @Car._Wheels._Tyres IsA '185' and Self IsA 'Metallic')

17.5. Interpretation of rules

17.5.1. Validity symbols

The symbols that display the validity of a value are described as validity symbols.

There are three different basic symbols:

- ✓ Value is valid
- ✗ Value is invalid
- Value has to be selected

If the symbols in a configurationbox are only displayed in gray shades, the value that goes with them is currently not selected.

[Basics/Validitysymbols](#)

In the properties of the form elements *configuration box*, *tab page* and *First component tree column* you find the property *Validity symbols* below the group *Representation*. With this option you can decide whether or not the validity symbols should be displayed on the form element.

Representation	
Visible	<input checked="" type="checkbox"/>
Font	Standard
Foreground color	COLOR_SYSTEM
BG color	COLOR_SYSTEM
Header alignment	Centered
Graphic	
Line color vertical	COLOR_SYSTEM
Fixed width	<input type="checkbox"/>
AdjustColumnWidth	<input checked="" type="checkbox"/>
Icon	<input type="checkbox"/>
Validity symbol	<input checked="" type="checkbox"/>

In configuration boxes display if the respective values are valid. The **configurationbox icon** has to be considered as a special case, because it consists of only the validity symbol to the assigned cause variable but no value is displayed.

On the form element *tab page* – with enabled option *Validity symbols* – the validity symbol is displayed next to the tab page header. With this the validity symbols of all form elements are together evaluated on the tab page. As long as at least one form element contains a forbidden value the red “forbidden cross” is shown on the tab page.



For the form element *component tree* there exist further levels of the validity symbols. In addition to the validity of the component also the validity of the subtree is displayed below the component.

Therefore the following symbols can be additionally displayed in the component tree:

- Displays that below this node in the component tree a ruled, but not yet allocated wasele exists for which NOVALUE is not a valid value.
- Displays that below this node in the component tree a forbidden (= MayNot) value exists in a wasele.
- Displays that the node in the component tree is stipulated by a Must rule whose subtree has a forbidden (= MayNot) value in a wasele.

17.5.2. Rule explanations

From the system rule explanations are automatically generated from the *May* and *May not* rules and these explanations are displayed if you click on the validity symbol.



These rule explanations are available in English and German and are generated according to certain patterns depending on the type of the used rule.

These generated explanations can appear quite cryptically. Because of this, explanations for rules and conditions can be set on the tab page *Explanation*. You can decide if the automatic explanation, a mixed explanation (partly generated, partly manually entered) or a completely manually entered explanation should be displayed.

With the option *mixed* or *manual* the explanation texts can of course be entered in any language that is defined in the frame – either as free text input, constant or Textelement. You can switch the mode via the context menu of the visible entry field. In the mode *Multilanguage text* the switching of the language is carried out via the country flag.

The explanation is valid for the case that the condition of the rule is valid. I.e. if for a value a *MayNot* rule with a rule explanation is defined, this explanation is only displayed if the value is currently forbidden. If the value is allowed, it is only displayed that it is principally allowed or why it is allowed if also a currently valid *May* rule exists.



Check the behavior of the different rule explanations in the application, especially the diesel engines with hard top vehicles or with convertibles.

In the next step the explanations with the diesel engines and the 205 tyres should be changed.



To do so, you open the rule definition of the *May* rule of the value *Diesel* in the component *Engine*, class *Car*. Click on the tab page *Explanation* and activate the *mixed* explanation. With this option you can define the why-part or (in case of a not *Always allowed* value) the “would be”-part of the explanation sentence.

You have to consider that this explanation is assigned to a *May* rule and therefore the text has to explain why or under which circumstances this value is allowed. Enter “Car is not a convertible” and leave the editor to create a Textelement.

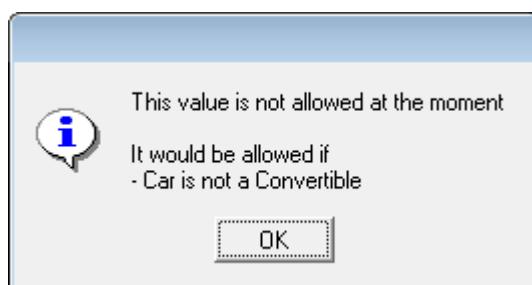


Figure 42: Definition of a mixed rule explanation

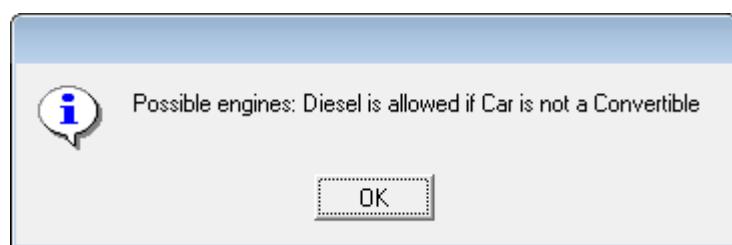
Via the context menu of the editor you can also select the input type (manually, constant or Textelement) of the text.

Check this rule explanation. You have to consider that the explanation changes for both situations (allowed and forbidden). 

For a Beetle the D70 engine is not allowed (the *Always allowed* flag was removed):



For a Passat or Golf the D70 engine is allowed due to the *May* rule:



Now open the rule definition of the value 205 of the component *_Tyres* in the class *Wheels*. There you go to the tab page *Explanation* and activate *Manually*. Enter "This engine is too weak for wide tyres. Please select a stronger engine." and leave the text field to create a Textelement. 

Check the rule explanation again. Only the text for the condition *May not* changes. Since the condition *May* does not originate from a rule, an own text cannot be defined.

17.5.3. The inference machine

With the term **inference machine** we describe the inference mechanism that checks the facts and rules according to a preset strategy and that produces conclusions from this check.

The inference machine is automatically started if not temporary components or features get a new value or after functions of camos Develop were called that carry out changes on settings of the rule processing (see chapter 17.6.8). In these cases the inference machine starts its work as soon as all program comes to a standstill, i.e. all due methods and triggers are processed.

If you need in a method the result of a processing for the inference machine, you can start this with the function *CheckAllRules()*. Then all effects of changes from the rule processing are available.

 *Inference machine*

17.6. Tips & Tricks

17.6.1. Handling in the rule editor

The following extensions, that are available since Version 7, make the rule maintenance easier:

- A cause variable from the local structure tree can be dragged via D&D in the rule editor. If you drag it on the root-mode or on an existing condition element, then a new condition element list is automatically generated, the selected cause variable is assigned and the value is displayed on the right corner.
- Several condition elements can be selected to delete, to switch or to align them. The multiple selection is carried out via Ctrl+Click on the single elements. Also a frame can be drawn around the selected elements with pressed left mouse-button. The element that has been selected at last gets a green border, all others are grey.
- Via the icon → in the head of a condition element the connection between two elements can be deleted or created. This is for example necessary, if two with AND linked parts of a condition should be linked instead with OR. To delete the connection, the icon (or the corresponding menu item *Connection new/delete* in the context menu of the element) of the leading element is selected, this can also be the root-node. The following element is clicked, so the connection-line is vanished: both elements have no logical connection anymore. To create a connection you have to act in the same way: on the leading element the connection mode is activated (the icon → is displayed animated) and then the connection to the following element is created/deleted. To cancel the connection mode, click in a free space in the editor. Attention: If you want to delete the connection from back to the front, the message *Unallowed cycle in the condition graph* is displayed. Attention: If the condition contains elements without a connection, this leads to a syntax error!

 *Basics/Rule editor/Condition editor/Description/Condition graph/Connection*

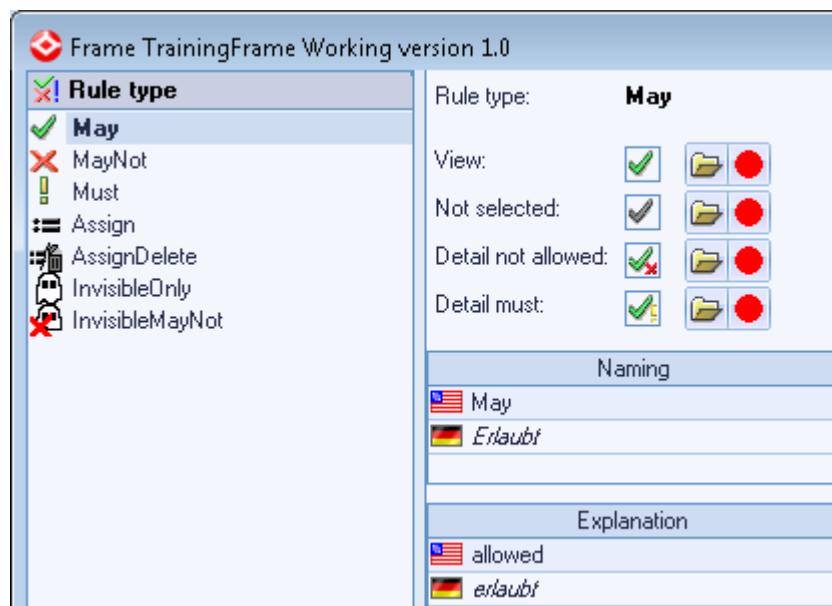
- The restoration memory in the upper part of the rule editor saves the last ten conditions of the rule. By clicking on one of the black points, an older condition is restored. Also the switching of a condition element is saved as a condition change, because the position of an element has an effect on the term of the condition!

 *Basics/Rule editor/Condition editor/Description/Condition graph/Order of the condition elements*

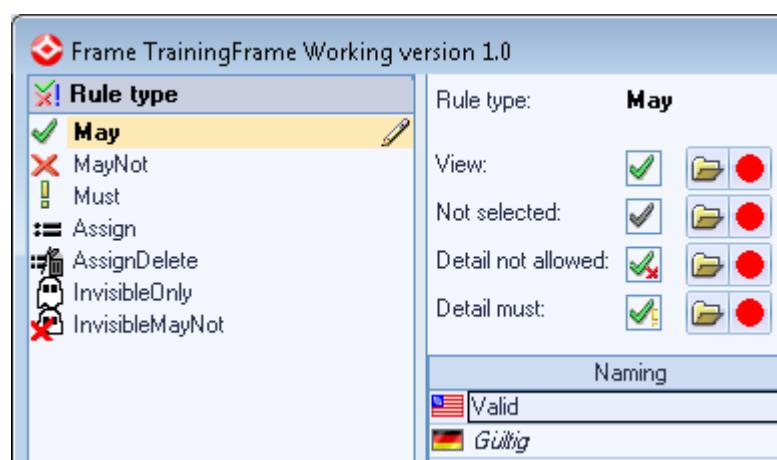
17.6.2. How to administrate rule names and explanations in the frame

The namings, explanation words and validity symbols of the system rules (May, MayNot, Must etc.) can be administrated in the frame.

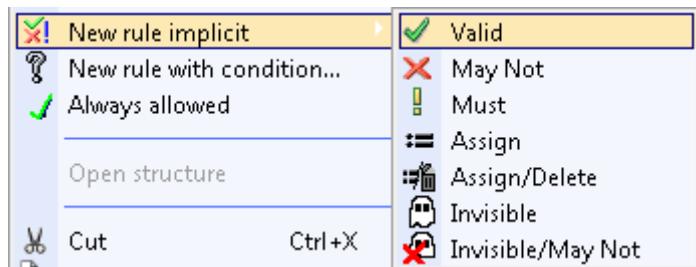
In the area *Rule Types* all seven system rules are listed with their (English) names and their standard validity symbol. If you click on the rule type *May*, the belonging properties are displayed on the right side.



Here you can define in the table *Naming* the rule name, that is displayed in the development system. If you change for example the May-Rule from "Allowed" to "Valid" ...



... then in the context menu *New rule implizit* of a value the new rule name is displayed:

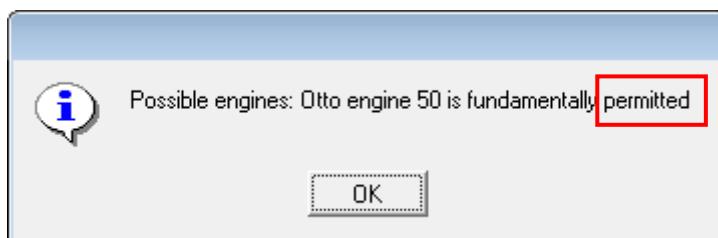


This is also possible for all other languages in the frame. You have to change the text display language in the development system (main menu *Extras*) to display the rule names in the desired language.

In the table *Explanation* the “validity word” is deposited, that has to be used in the rule explanation. If this for example is changed at the May-Rule from “allowed” to “permitted”, the explanation of rule is not called anymore:

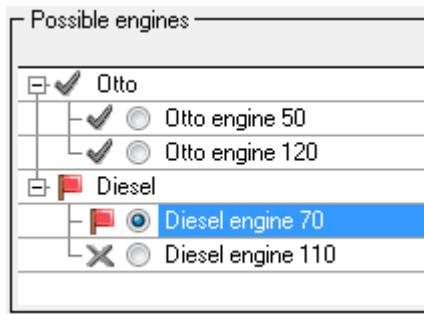


it is now called:



Furthermore the validity symbols of the rule types can be changed in the frame. To load a new symbol the Open-Icon is used. Bitmaps, JPEGs, GIFs, TIFs, PCXs and PNGs are allowed. To restore the camos Develop-standard the icon ● is used.

In the following figure the validity symbol for the *MayNot*-Rule was changed:



17.6.3. System texts for Default rule explanations

To change or translate the rule explanation of a value which is always allowed or not always allowed, a *System text* can be defined in the frame under *Options*.

Because these messages are provided from the system, up to now only an English or German text could be displayed. As from version 10.0, these texts can be influenced. To do so, a Textelement from the context public is deposited.

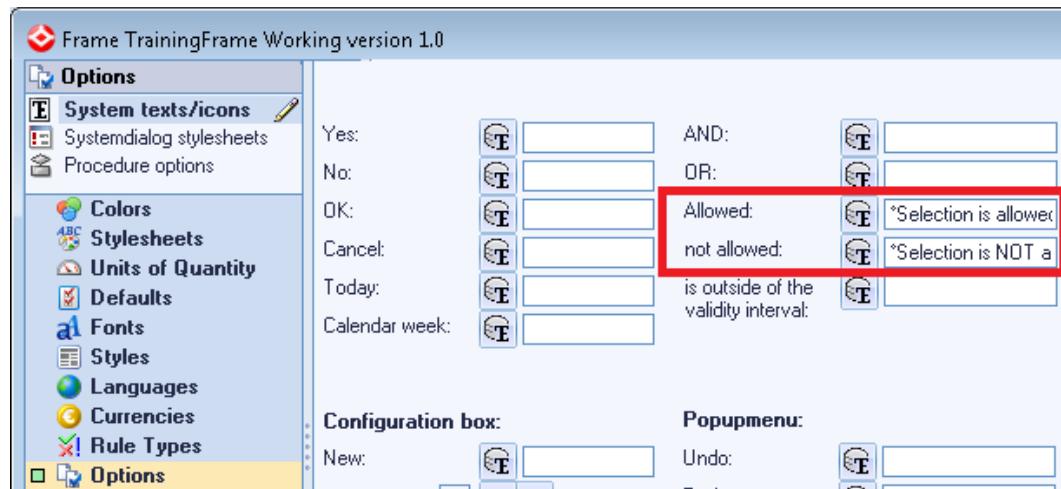
By default this message is displayed for a not always allowed value:



For an always allowed value, this message is displayed:



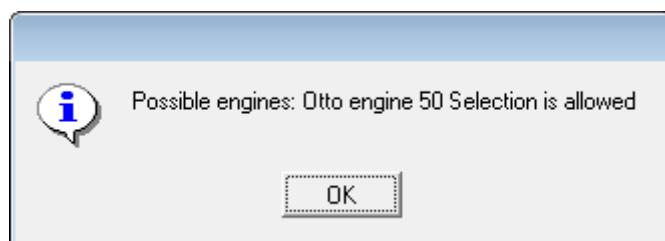
If Textelements for are deposited for *Allowed* and *Not allowed* in the frame, these will be displayed during runtime.



For a not allowed value this message:



And for an always allowed value this message:



17.6.4. User defined rule types

In addition to the system rules (May, MayNot, Must, etc.) new rules can be created and administered in the area *Rule Types* in the frame. These user-defined rules are also called Recommended-Rules, because they have free definable names and validity symbols, internally they are interpreted like all May-Rules.

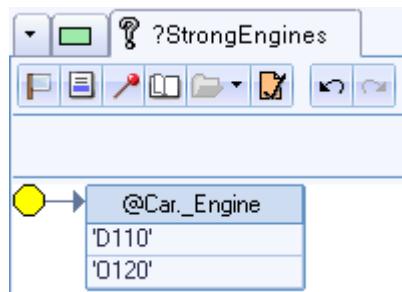
So it is possible to simulate different “degrees of May” by corresponding names and icons, for example adverse > allowed > strongly recommended. From the view of the rule, a selected “adverse value” is not forbidden but allowed.

Frame/Rule types

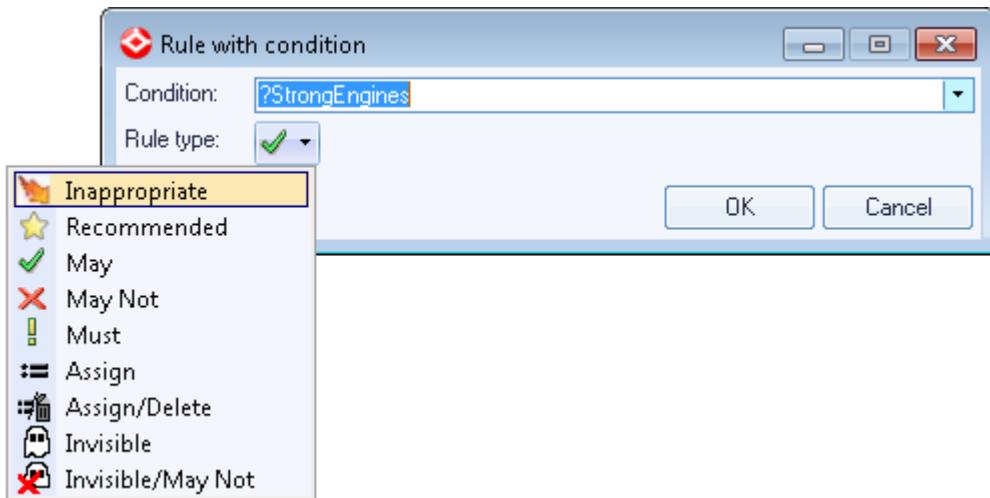
For example the tyres 185 and 205 should be displayed as “recommended”, if a strong engine is selected. The tyres 155 and 175 however should be displayed as “not recommended” in this case. To leave the free choice to the user besides this guideline, no MayNot-Rule is used but two Recommended-Rules are defined:

Rule type:	Pro	Con
View:		
Not selected:		
Detail not allowed:		
Detail must:		
Naming		
	Recommended	
	Empfehlung	
Explanation		
	recommended	
	empfohlen	
Naming		
	Inappropriate	
	Ungünstig	
Explanation		
	not recommended	
	nicht empfohlen	

In the class *Wheels* a condition is defined that contains the two strongest engines:



Create a *New rule with condition* under the values of the tyres and select the desired rule type for the value:



After the tyres 155 and 175 are equipped with the Inappropriate-Rule and the tyres 185 and 205 are equipped with the Recommended-Rule, the structure tree above the component _Tyres looks like:



During the runtime the corresponding icons are displayed at the tyre-dimensions when you select an engine:



17.6.5. How to hide forbidden values

A possibility to fade out not allowed elements you already got to know via the rule type Invisible/MayNot.

The values which are “only” forbidden can also be faded out by activating the flag “Only permitted values” in the form element configuration box.



This is carried out on the configurationbox for the component `_Tyres`. To do so, you open the form `MainForm` in the class `start`.

Select the configurationbox for `_Car._Wheels._Tyres` and activate the item *Allowed values only* in the property group *Handling*.

17.6.6. Use of rule groups

Rules can be separated in **rule groups** to e.g. allow different product complexities with the same structure in the class- or structure tree via rule types of different user profiles.

In the basic setting all rules are checked independently of the group to which they belong. The efficiency of the rule system and therefore the performance of the rule check can be increased with activating or deactivating rule groups via the functions `RuleActivate(<Group>)` or `RuleDeactivate(<Group>)`.

In camos Develop exists a predefined rule group `Init`. Rules of this group are processed only once when an object is instantiated. The rule group `Init` is always object-related activated or deactivated. All other rule groups can only be globally activated or deactivated.

Rule groups are defined in the *properties of the knowledge base* on the tab page *Rule groups*. Here you can also find the predefined rule group `Init`.

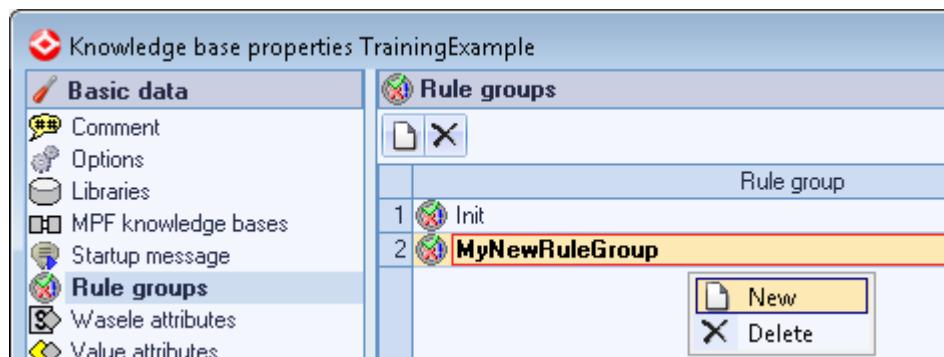


Figure 43: Creating a new rule group

In order to create a new rule group, you select the menu item `New` of the context menu in the tab page *Rule groups* of the properties of the knowledge base. Then you specify the name of the rule group and close the properties dialog. In the rule editor, the new rule group is now available and can be assigned to a certain rule.

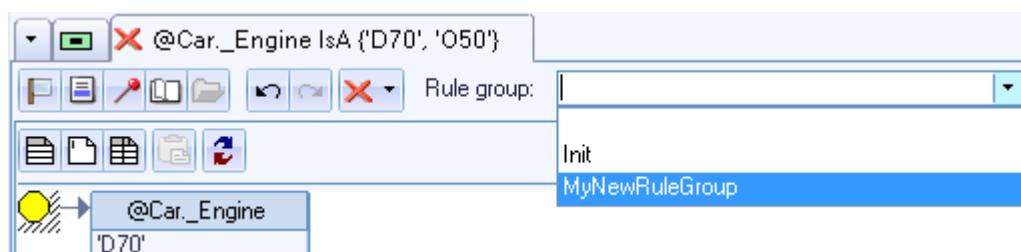


Figure 44: Assign rule group in rule editor

17.6.7. Deactivating the rule check

Via the function `SetRuleMode(<Mode>)` you can control when the inference machine should be active.

There are three modes of the rule check:

Mode	Explanation
DISABLED	The rule check is generally disabled. Therefore all possibilities of the configuration are allowed.
ENABLED	The rule check is enabled, but it is not carried out. Via the call of the function <code>CheckAllRules()</code> you have the possibility to check defined rules once.
AUTO	Default value: The rule check is always carried out before the interpreter waits for a user entry. Via the call of the function <code>CheckAllRules()</code> you have the possibility to force the rule check before the automatic rule check.

If you switch back to ‘AUTO’ via the function `SetRuleMode()`, a rule check is automatically carried out after the processing of all methods.

 [Function reference/Rule control/`SetRuleMode \(Mode\)`](#)

 [Function reference/Rule control/`CheckAllRules`](#)

17.6.8. Restricting the rule check to subtrees

By default first the rules in the start object are checked and then all subcomponents. Via the function `SetRuleRoot(<Object>)` the starting point of the rule processing can be set to a different object.

This is especially interesting if very big object trees exist and the regular complete rule check restricts the performance of the system.

If the result of a configuration is e.g. a shopping cart whose elements do not affect each other, the rule check can be reduced via `SetRuleRoot()` to the currently processed entry.

17.7. Repetition

- Prerequisite for the processing of rules is the demanding of a license that includes the rule processing such as camos.Configurator or camos.CAPP, e.g.

LicenseDemand("camos.Configurator")

- For cause variables in which rules have to be affective, the property *Rules enabled* has to be set.
- The condition *Always allowed* has to be set matching to the rules or it has to be removed.
- *MayNot* rules have priority to *May* rules.
- A click on the validity symbols displays the rule explanation.
- Rule explanations can be defined manually.

18. Constraints

18.1. Intention

In the car configurator MayNot rules should be defined that are valid in both directions. The previously introduced rules always forbid an element in dependency of another one; in the following rules should be defined that exclude or allow reciprocal values. In this chapter you will learn how to create *May* and *MayNot* rules in a table via constraints.

18.2. Use of constraints

18.2.1. Definition of constraints

The constraint is a rule mechanism that allows to reciprocally express direct dependencies between two or more components or features. This means that e.g. A in connection with B and B in connection with A is forbidden or allowed.

The mechanism is very efficient and clear in order to display reciprocal relations between components and features since these relations in a constraint are displayed and maintained in a table.

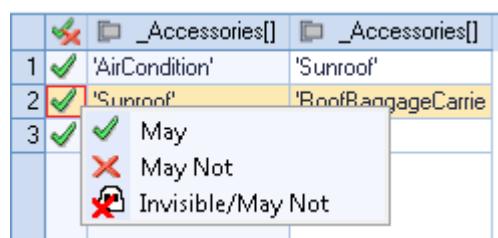
A constraint is therefore another type of display for rules. As with implicit rules a constraint only works for cause variables that have the property *Rules enabled* activated. If one of the used cause variables does not have this property, the constraint does not affect this cause variable.

Constraints are administrated in the wasele lists via the icon .

18.2.2. The constraint editor

The constraint editor is a table in which the different cause variables can be defined in the columns. The combinations of values of the cause variables that have to be processed are defined in the single rows.

Via a right click in a column you can start the cause variable selection via the menu item *Cause variable* in the context menu; this cause variable is then noted in the column head. Via the menu item *New...* further columns and lines can be added; with *Delete...* you can remove columns or lines.



	 _Accessories[]	 _Accessories[]
1	 'AirCondition'	'Sunroof'
2	 'Sunroof'	'RoofBaggageCarrie
3	 May	
	 May Not	
	 Invisible/May Not	

Figure 45: The constraint editor

In the first column the validity symbol defines if the combination of values in that row is allowed or forbidden (= May and MayNot rule). This can be switched via a right-click on the validity symbol.

If the cursor is positioned in a cell of the table, on the right side the list of the defined values is displayed (as in the rule editor). From this list you can make a selection via a double click. With features (as in the rule editor) quantity operators are available in the context menu via the menu item *Apply value...* (right click).

Under the table with which the constraint is defined you have as with the other rules the possibility to specify own rule explanations. The standard rule explanation looks like this:



As you can see, the name of the constraint and the line number are also displayed. This is very useful for the error research. If you want to change the text, you can specify a mixed or self-defined (manual) rule explanation, see chapter 17.5.2.

18.3. Practice: Constraint for accessories

First a constraint has to be defined that forbids the combination of alloy rims with normal paintwork as well as of steel rims with metallic paintwork.

 Open the class *Car* and click on the icon  in the head of the wasele list. Select in the dialog the icon for constraints  in the toolbar and enter the name of the constraint: "Rims_Paintwork". Confirm the creation with *OK*. The constraint editor is opened.

Click with the right mouse button in the first text column and assign via *Cause variable* the component *_Paintwork* to it. Proceed the same way in the second text column and select the component *_Wheels._Rims*.

	  _Paintwork	 _Wheels._Rims
1	<input checked="" type="checkbox"/>	
2	<input checked="" type="checkbox"/>	

Values

- NOVALUE
- ANYVALUE
- Paintwork**
 - Normal
 - Metallic

Explanation: Automatic
 Mixed
 Manually

Via the constraint you forbid the combination of alloy rims with normal paintwork. To do so, you set the cursor in the first row of the first column and doubleclick in the selection list on *normal*. Then you select for the second column the *AluminiumRims* in the first line of the selection.



		_Paintwork	_Wheels_Rims
1		'Normal'	'AluminiumRims'
2			

Values
 NOVALUE
 ANYVALUE
 Rims
 AluminiumRims
 SteelRims

Now you forbid the combination of steel rims with metallic paintwork. To do so, you select in the second line for the first column *Metallic* in the selection list and for the second column the *SteelRims* in the selection list. Via a right-click on the validity symbol you switch to *MayNot* rules in both rows.



Now you test the behavior of the constraint in the application. If one of the cause variables should not react as expected, check if both components activated the property *Rules enabled*. (To remind you: *_Paintwork* is in the class *Car*, *_Rims* is in the class *Wheels*.)

Set the rule explanation for the first line to *Manually* and define a rule explanation, e.g. "Aluminium rims can only be used together with a metallic paintwork."



Now the combination of a sunroof with the roof baggage carrier has to be forbidden. The rule explanation should be comprehensible.

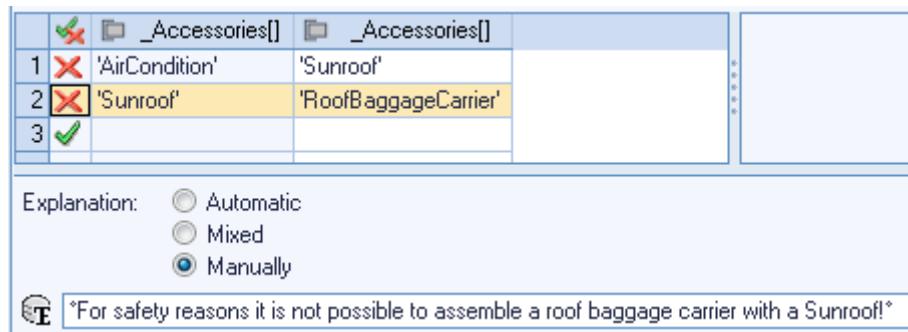
To do so, you create a new constraint "Accessories" in the class *Car*.

For both columns of the constraint table you select the component *_Accessories[]* via the context menu item *Cause variable...*.



Via a right click on the validity symbol you set the first and second line to *MayNot*. In the first line you select via a double click in the value list *SunRoof* for the first cell and *RoofBaggageCarrier* for the second cell. Then enter the combination *AirCondition* and *Sunroof* in the second line.

Then you set the rule explanation to *Manually* and enter your rule explanation.



The screenshot shows a constraint editor interface. At the top, there are two columns labeled '_Accessories[]' with icons for a red X and a green checkmark. Below this is a table with three rows:

1	<input checked="" type="checkbox"/> 'AirCondition'	'Sunroof'
2	<input checked="" type="checkbox"/> 'Sunroof'	'RoofBaggageCarrier'
3	<input checked="" type="checkbox"/>	

Below the table is an 'Explanation:' section with three radio button options: 'Automatic', 'Mixed', and 'Manually'. The 'Manually' option is selected. At the bottom, there is a note: 'For safety reasons it is not possible to assemble a roof baggage carrier with a Sunroof!' with an information icon.



Test the behavior of the newly entered constraint.

18.4. Tips & Tricks

18.4.1. Display of constraint rules in the structure tree

In order to get a better overview over the rules for a certain cause variable, it is sometimes useful to show also the constraints in the structure tree.

To do so, you select the menu item *Display constraint rules* in the context menu of the root of the structure tree, then they are displayed like the rules that are defined at the values. If you doubleclick on a constraint rule, the respective constraint editor is opened.

But this works only for features and components within the class in which the constraint is created. In other classes (e.g. in the class *Wheels* instead of in *Car*) this does not work, because the context of the constraint is not guaranteed.

18.4.2. Fill constraints with data from an Excel file

It is possible to fill a constraint with data from an Excel file. To do so, the *Constraint Wizard* is used. The Excel file has to be formatted in a specific way. The Constraint Wizard construes the values of the first line as cause variables (features), the following lines as corresponding values.

Therefore value combinations, which are already defined in a list, can be read in a fast and comfortable way

?

Workbench/Wasele/Constraint/Constraint editor/Description/Constraint Wizard

18.4.3. Constraints with database links

By setting the option *Constraint with database link* in the create dialog of a constraint, a constraint with contact to a database table can be created.

The advantage of this constraint is that the rules can be read-out from the columns of a database table. The rules can thus be deposited externally and used during runtime. In the knowledge base the link to the database is created and via defining the cause variables in the table headers the connection to the knowledge base elements is defined.

?

Workbench/Wasele/Constraint with database link

18.5. Repetition

- Constraints are tabular combinations of *May* and *MayNot* rules.
- The property *Rules enabled* has to be set on the features and components that are used in constraints.
- Rule explanations can be defined individually for each line of a constraint.

19. Discount calculation

19.1. Intention

With orders of expensive products such as cars, often purchase incentives in form of special discounts are offered to the customer. E.g. 2 % are deducted for a new customer, 6 % for regular customers.

From the side of the car dealers it is of course not desired that the customer independently grants himself a discount. For the customer however, who can spend just a certain amount, the including of discounts is helpful. Therefore he can lead his negotiations with the dealer to a "minimum discount".

The discount has to be entered in an entry field on the form and reduce the end price. The entered discount has to be deducted from the list price that already sums up all currently selected components.

Additionally the discount has to be entered in per cent and restricted to a certain span. Therefore you will learn how formats and value ranges are defined and how you react to entries of the user via an assignment trigger.

19.2. Practice: Create the feature Discount

On the form an entry field has to be created in which the user can enter the discount.

The entered discount has to be deducted from the base price of the car models and the added components. The car knows its parts, because they exist as components. Therefore the feature *Discount* has to be created in the class *Car*.

Open the class *Car*, create a numerical feature with the name "Discount" and initialize it with the static value 0.

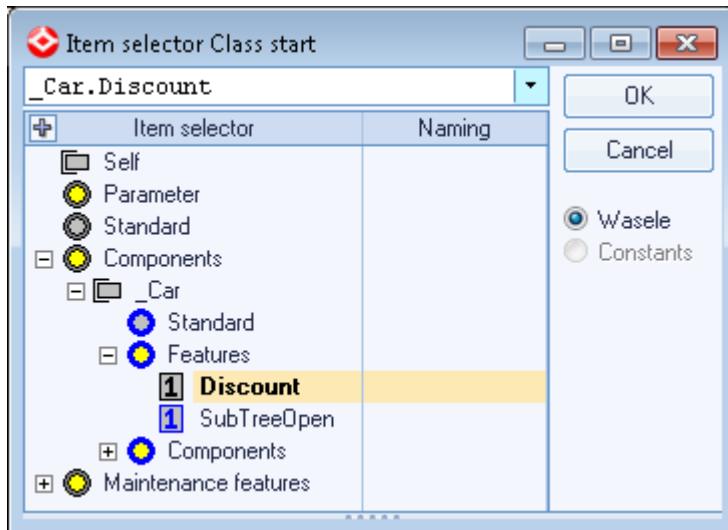


The groupbox "Price info" that already exists on the form and that displays the current list price, is extended by the discount entry.

Create a *static label* and enter the text "Discount". Place the label in the groupbox.

Create an *Editline* for the discount entry. Position it next to the label. Assign the feature *Discount* (located under the component *_Car*) as cause variable.





Note: Instead of a *static label* in which the display text has to be entered statically, you can also use the form element **naming**. This shows the naming of the entered cause variable (feature or component) in the current dialog language. If a naming should not exist in this language, the naming of the main language is displayed. If there are no deposited namings, the name of the cause variable is displayed.



Test your application.

Price info	
List price:	\$15,999.00
Discount:	123456789.99999999

In the field **Discount** any entry with an unlimited number of places is possible. The discount should be restricted to a certain format.

19.2.1. Determine a format

Definition

Formats are characters that determine a pattern for the in- and output of numerical, date- and string features via certain control characters.

With multilingual features the following has to be considered:

If in a multilingual feature a format is not specified in a secondary language, the format of the main language is used.

The discount has to be entered with two digits before and one digit after the comma. The formats of numerical features are specified with the control character #.

In the table in the editor of the feature *Discount* you enter the following character string **##.##** in the column *Format*.



Consider the notation with period! The entry of a comma leads to a syntax error.



This format specification describes a two-digit entry before and a one-digit entry after the period / comma.

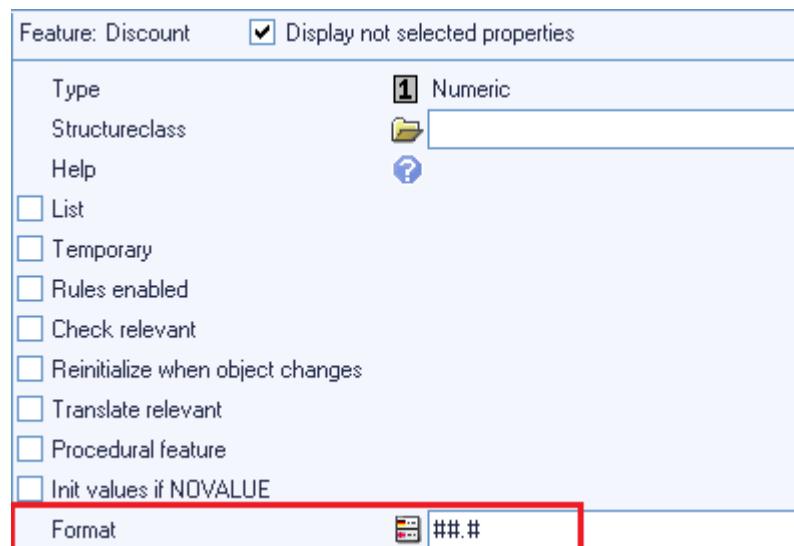


Figure 46: Definition of a numerical format

The double definition of the format is not necessary in this case, because in the secondary language German the same format as in the main language English has to be used.

Workbench/Structure tree/Features/Format

Restart the application and observe the change.



The screenshot shows a SAP Fiori application window titled 'Price info'. It contains two input fields:

- 'List price:' with the value '\$15,999.00'
- 'Discount:' with the value '99.9'

The entry is now only possible in the defined format that limits the number of before and after comma / period digits.

The intention from the beginning of the chapter was a percentage display of the discount. In order to display the discount with a unit of quantity %, this has to be first defined in the frame.

19.3. Practice: Defining the unit of quantity Percent

Create a unit of quantity for “Percent” in the frame, like you have already done it with the power for the engine in chapter 15.3.



Open the TrainingFrame and select and reserve the area *Units of Quantity*. Create the new unit of quantity group *Percent* with the *Unit of Quantity %*.

The screenshot shows the 'Group of unit of quantity' section of the Frame interface. A new group named 'Percent' is being created. The 'Master unit of quantity' is set to 'Percent'. The 'Group name' is also 'Percent'. The 'Naming of group' section includes the American flag icon followed by 'Percent' and the German flag icon followed by 'Prozent'. Below this, a table lists the unit of quantity with 'Name' as 'Percent', 'Factor' as '1', and 'Offset' as '0'. The 'Unit of quantity' column shows two entries: the American flag icon followed by '%', and the German flag icon followed by '%'. There are also icons for saving and canceling changes.

Contrary to the power which is in other countries specified with a different unit and therefore converted, the specification in percent is the same on the whole world. For this reason only one unit of quantity is created in this group.



Open the editor of the feature *Discount* and assign the unit of quantity *Percent*.

The screenshot shows the Feature Editor for the 'Discount' feature of the 'Car' component. The left sidebar shows the feature tree with 'Discount' selected. The main panel shows the 'Values' section with 'NOVALUE', 'ANYVALUE', and 'Individual values' (set to '0'). Below this are 'Ranges' and 'Initialization' sections. The 'Initialization' section has a value of '1.0'. A yellow callout points to this value with the text 'numerical feature Discount'. Another yellow callout points to the 'Initialization' section with the text 'Initialization with 0'. On the right, the 'Feature: Discount' properties are listed: Type (set to '1 Numer'), Help, Format (set to '#.#'), Unit of quantity (set to 'Percent'), Internal access, External access, and Comment. A yellow callout points to the 'Format' property with the text 'Format specification'. Another yellow callout points to the 'Unit of quantity' property with the text 'Unit of quantity'.



Start your application and observe the affect of the specification of a unit of quantity.

Price info	
List price:	\$15,999.00
Discount:	99.9 %

As desired, the discount is displayed in percent. Since an entry and a display have no effect on the list price, now a further feature is created that can accept the reduced end price in Dollar.

Create the currency feature "EndPrice" in the class *Car*.

In the *Groupbox* you create a *static label* with the labeling "End price" and a form element *Currency* for the display of the currency feature. Assign the *EndPrice* as cause variable to the form element *Currency*.



Extract of the form preview

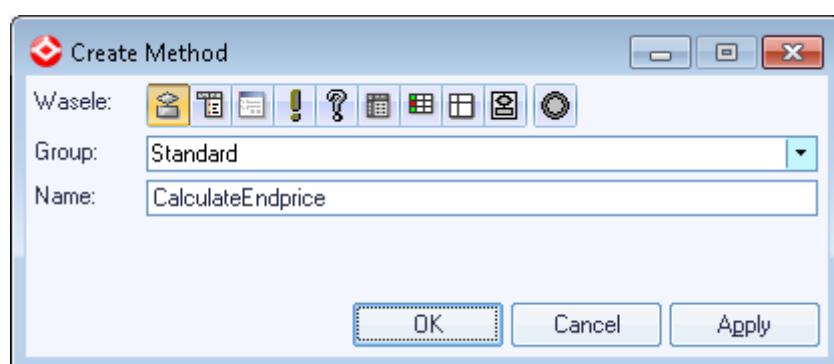
Price info	
List price	123.456,00 EUR
Discount	Editline
End price	123.456,00 EUR

19.4. Practice: Implement the discount calculation

Since currently no function is carried out if the user enters a value in the discount editline, in the next step a function has to be created and called.

Functions should be entered in methods. These have the advantage that they can be used globally and therefore the same functionality must not be coded several times.

In the wasele list of the class *Car* you create a new method with the name "CalculateEndprice".



Enter the following code:

```
EndPrice := ListPrice * (1 - Discount / 100);
```



An important consideration is where the discount calculation is triggered. Should the calculation start after a user action, e.g. click on a button. Or should it be carried out with each new entry in the field of the discount entry and/or a change of the list price?

In this case the second version is selected. The calculation is triggered every time the user enters a discount and exits the entry field or if the list price changes due to adding or removing a component.

To do so, a so-called **assign trigger** can be used.

The assign trigger of a feature or a component is always carried out exactly when the value of the feature or the component changes. This is also valid for the value change via an init value or an assignment rule.

The assign trigger does not have to be specially activated. As soon as a program code is entered in the editor on the tab page *Assign trigger* of a feature/component the icon gets a red flash, e.g. RTF-feature with assign trigger or object class component with assign trigger.

The procedure editor of the assign trigger contains two automatically existing writing system parameters:

LastValue - The parameter *LastValue* is of the same type as the feature for which the trigger is defined. For components it is of the type *String*. It contains the “old” value of the feature before the assignment.

Index - The parameter *Index* contains for list features the index of the last changed list entry.

The assign trigger is entered directly on the feature where a reaction has to be carried out with a change; in this case on the features *Discount* and *ListPrice*.

 Enter the method call *CalculateEndprice()* on the tab page *Assign trigger* in the feature editor of *Discount* and *ListPrice*.

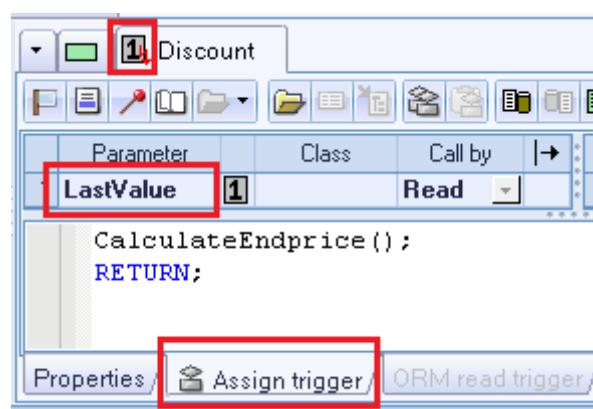


Figure 47: Assign trigger of a feature

 Start your application and test the discount calculation.

You certainly noticed that it is possible to enter a discount of up to 99.9 %. Since this is not really realistic, a value range of 0 to 10 % is introduced.

19.4.1. Practice: Determine the value range

Between the field *Init value* and *Values* of a numerical feature (in this case *Discount*) is a table in which **value ranges** can be entered.

A value range combines a number of values without specifying them explicitly. Which values are within or outside of the section is defined via square brackets (inclusive) and/or parentheses (exclusive).

The value range [1..4] contains therefore the values between 1 and 4. Since a format with one digit after the comma was defined, the value range also includes all numbers between 1 and 4 with a decimal place (e.g. 1.5 and 3.2).

It is possible to include or exclude the first and/or last value. Via the syntax (1..4) the interval from 1 to 4 is described, except 1 and 4.

Value ranges can just as other values be applied to the structure tree and assigned with rules.

Enter the range [0..10] in the class *Car* on the feature *Discount* and apply the range (via double click) as value into the structure tree.



FAQ Basics/Value ranges

Start your application and test the constraint of the value range.



The entered value range is not considered. What could be the reason?

The specification of a value range is a constraint. With other words, there is a rule that forbids all other values (the values > 10 % and < 0 %).

In order to use value ranges and therefore rules, you have to set the option *Rules enabled* for the feature *Discount* as described in chapter 17.2.3.

Set the option *Rules enabled* for the feature *Discount*.



Furthermore a license for the use of the rules has to exist.



Ensure that the license check is implemented in method *New* of the class *start*:

```
LicenseDemand( "camos.Configurator" );
```

With option <i>Rules enabled</i>	Without option <i>Rules enabled</i>																								
<table border="1"> <tr> <td colspan="2" style="text-align: left;">Price info</td> </tr> <tr> <td>List price:</td> <td>\$20,169.00</td> </tr> <tr> <td>Discount:</td> <td>77.0 %</td> </tr> <tr> <td>End price:</td> <td>\$4,638.87</td> </tr> </table> <table border="1"> <thead> <tr> <th style="text-align: center;">Modules</th> <th style="text-align: center;">Price</th> </tr> </thead> <tbody> <tr> <td> <input checked="" type="checkbox"/> New Beetle Cabriolet <input checked="" type="checkbox"/> Soft top </td> <td style="text-align: right;">\$0.00</td> </tr> </tbody> </table>	Price info		List price:	\$20,169.00	Discount:	77.0 %	End price:	\$4,638.87	Modules	Price	<input checked="" type="checkbox"/> New Beetle Cabriolet <input checked="" type="checkbox"/> Soft top	\$0.00	<table border="1"> <tr> <td colspan="2" style="text-align: left;">Price info</td> </tr> <tr> <td>List price:</td> <td>\$20,169.00</td> </tr> <tr> <td>Discount:</td> <td>77.0 %</td> </tr> <tr> <td>End price:</td> <td>\$4,638.87</td> </tr> </table> <table border="1"> <thead> <tr> <th style="text-align: center;">Modules</th> <th style="text-align: center;">Price</th> </tr> </thead> <tbody> <tr> <td> <input checked="" type="checkbox"/> New Beetle Cabriolet <input checked="" type="checkbox"/> Soft top </td> <td style="text-align: right;">\$0.00</td> </tr> </tbody> </table>	Price info		List price:	\$20,169.00	Discount:	77.0 %	End price:	\$4,638.87	Modules	Price	<input checked="" type="checkbox"/> New Beetle Cabriolet <input checked="" type="checkbox"/> Soft top	\$0.00
Price info																									
List price:	\$20,169.00																								
Discount:	77.0 %																								
End price:	\$4,638.87																								
Modules	Price																								
<input checked="" type="checkbox"/> New Beetle Cabriolet <input checked="" type="checkbox"/> Soft top	\$0.00																								
Price info																									
List price:	\$20,169.00																								
Discount:	77.0 %																								
End price:	\$4,638.87																								
Modules	Price																								
<input checked="" type="checkbox"/> New Beetle Cabriolet <input checked="" type="checkbox"/> Soft top	\$0.00																								

If you restart your application and enter a discount outside the value range, you will see the rule violation of the discount at an indication in the component tree (provided that the rest of the configuration is valid).

Since this indication also displays other invalid components of the configuration, the user cannot see where the actual error is; therefore it is useful to indicate the forbidden discount value in a different way.

19.4.2. Practice: End price calculation only with valid values

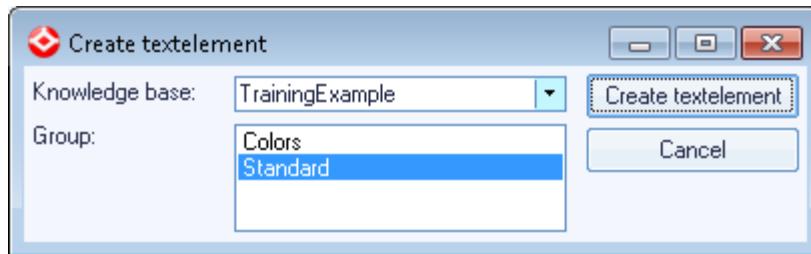
If an invalid discount is entered, an error message with an explanation has to be displayed. Also the discount has to be reset to the last valid value. In case of an error an error message has to be displayed.



To do so, you create a multilingual string constant with the name *!ErrorDiscount*. Specify an English and German message text.



To create a textelement from the entered text, click on the icon in the toolbar of the constant editor. Confirm the Create textelement dialog after you selected the desired knowledge base and group.



The constant editor looks like this afterwards.



In order to reset the incorrect value to the last valid value, the assign trigger of the feature *Discount* has to be modified.

Complete the code analog to the following figure on the tab page *Assign trigger* of the feature *Discount*.



	Parameter	Class	Call by	→
1	LastValue	[1]	Read	↓

```

• IF not IsValid(Discount, Discount) THEN
    WinMessage('Error', !ErrorDiscount);
    Discount := LastValue;
ENDIF;
CalculateEndprice();
RETURN;

```

[Basics/Control Statements/IF-THEN-ELSE-ENDIF](#)

The camos Develop function *IsValid* returns 1 if the specified value *Discount* is allowed for the variable *Discount* by all rules. If this is not the case (return value 0), the error text that is defined in the constant *!ErrorDiscount* is displayed via the function *WinMessage()*.

The function *WinMessage* displays a message with a free definable text. Via an additional parameter the type of the message is specified: Error, question, info or warning, e.g.:

```
WinMessage("ERROR", "This is an error message");
```

[Function reference/Dialog Functions/WinMessage \(MessageType, Message\)](#)



The system parameter *LastValue* contains the last condition of the feature. This can be used to restore the last valid condition of *Discount*.

The end price is only calculated if a valid value is entered or if the value was reset to a valid state.



Test the discount calculation in your application.

19.5. Tips & Tricks

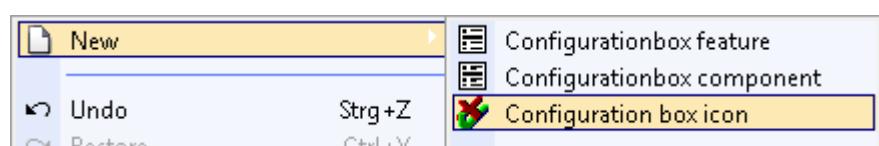
19.5.1. Display the validity of the discount

In camos Develop you can find the form element configurationbox Icon picture to display the validity of a scalar component or a scalar feature. Via the transferred (ruled) cause variable is decided between the Forbidden and Allowed display.

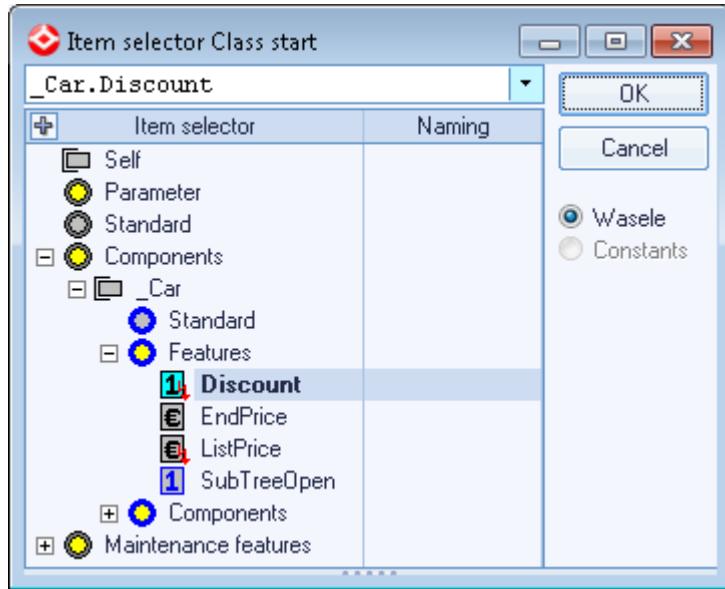
Workbench/Wasele/Form/Form elements/Configuration box/Configurationbox Icon

This possibility can be selected instead of the assign trigger. If the assign trigger still exists, the allowed icon is always displayed because forbidden values are constantly reset to a valid value.

In order to use the *Icon display* in our application, the source code in the assign trigger of the feature *Discount* should be deleted. Then a validity symbol can be created on the *MainForm* via the context menu *New -> Configuration box icon*. This is positioned next to the discount editline.



The feature *Discount* from the component *_Car* is entered in the field Cause variable (via the Item selector dialog).



In the interpreter this has the following effect:

Invalid discount	Valid discount
Price info	
List price: <input type="text" value="\$17,900.00"/>	List price: <input type="text" value="\$17,900.00"/>
Discount: X <input type="text" value="20.0 %"/>	Discount: ✓ <input type="text" value="5.0 %"/>
End price: <input type="text" value="\$14,320.00"/>	End price: <input type="text" value="\$17,005.00"/>

19.5.2. Confirm an entry through Return

After entering a discount, the input field has to be left so the entered value is written into the cause variable, here the feature discount.

If the entry should also be assigned to the cause variable by pressing Return, the property *Assign value with Enter* can be activated for the form element.

19.6. Repetition

- In order to react to user entries, it is possible to use an action in the *Assign trigger* of the feature. The parameter *LastValue* in the *Assign trigger* saves the last condition of a feature / a component.
- Value ranges can be defined for numerical features. Ranges are only effective if the rule processing is active for the feature.
- In order to check the validity of a ruled wasele, the function *IsValid(wasele, value)* can be called.

20. Creating a quotation

20.1. Intention

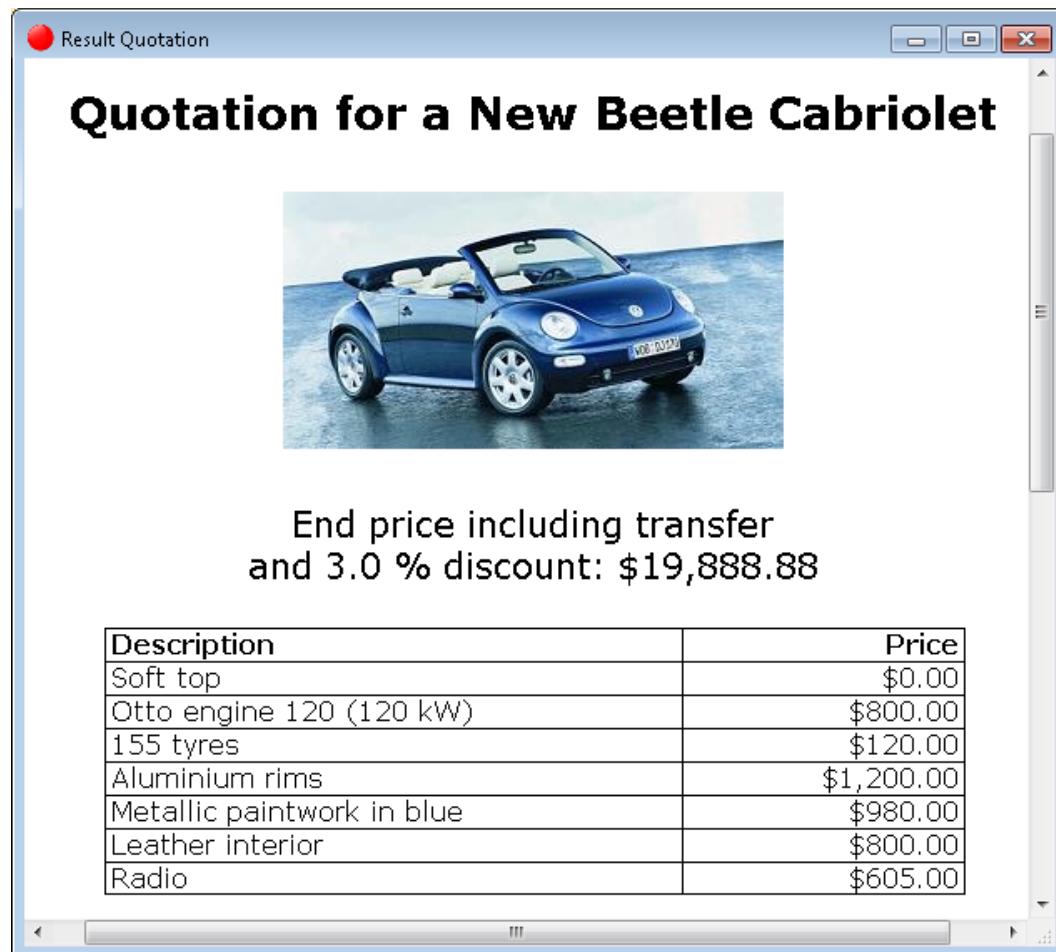
One of the most important targets of each configurator application is to obtain a document in form of a quotation, order form, a bill of materials etc. for the configured article. The user should save and print this document so that he can still use the information from the configuration even without PC and configurator application.

In case of the car configurator a quotation for the assembled model has to be created.

The quotation should contain all relevant information to the product: product image, name of the model, all selected components, their prices, the discount and the total price.

In the following chapter you will learn about the document type Result, the text modules that belong to it and how to use data from the configuration in the result. You will also learn how a result object is put together, generated and opened. The result has to be called via a menu trigger.

At the end of the practice the result should look like this:



20.2. Create a result

In order to take data out of the application and display it like a text document, so-called **results** are used in camos Develop.

Results are administrated in the frame. You can define the type (String, RTF or HTML), the layout, the appearance of the headers/footers, the document language etc. for results.

In classes that contain information/data that has to be displayed in the result, the result is entered under the node *Results* in the structure tree. Via this allocation is defined which objects are able to provide texts to the results.

The actual texts are allocated to the result via text modules (constants with the same type as the result). These text modules are for a RTF-result, constants of the type RTF. In addition to manually entered texts also data from the configuration can be integrated (via cause variables).

The result is generated and opened via the function call *WinOpenDoc()*. I.e. a result object is generated similar to an object of the application which is created by starting the debugger. The generation of the result is carried out as from the object in which *WinOpenDoc()* was called.

Prerequisite is, that on all components whose objects are involved in the result the option “Output to result / printout form” is enabled. During the result generation the text modules of the individual objects are put together.

20.2.1. Practice: Define a result in the frame

Results are defined in the knowledge base properties. A result describes the properties and guidelines which are kept in conjunction with the result generation. These properties are inherited to the text modules and are therefore valid in all parts of this result. In a result e.g. a margin setting can be set that is effective for the completed document.

 Open the knowledge base properties of the knowledge base *TrainingExample*, reserve them and switch to the tab page *Results*.

Create a new result of the type *RTF* via the context menu item *New* in the upper, left table. Assign the name “Quotation”.



Figure 48: Creating a new result in the frame

On the right side a combobox with all available *Languages* is faded in. Since a result can be defined in several languages, it is possible to differentiate.

Select from the dropdown menu of the combobox the icon  besides the country United States.



Under the combobox a further editor is opened which is subdivided into five tab pages.



Options with the definition of the result

Header 1st page – For the first page a separate header can be defined that differs from the header of other pages

Footer 1st page – For the first page a separate footer can be defined that differs from the footer of other pages

Header following pages – Definition of the header of the following pages

Footer following pages – Definition of the footer of the following pages

Layout – Here the margin settings of the document contents, the header and footer are entered. It is also definable if the result is generated in portrait or landscape format. Additionally the paper size can be selected.

Definition

Momentarily the default result layout should be used for both languages.

Therefore you insert the country code 49 (Germany) from the language combobox the same way. Confirm your entries by clicking on the button *Release and close*.



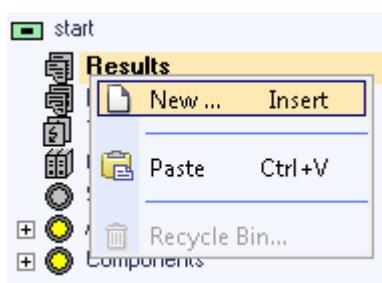
20.3. Practice: Create a test result

In order to display the result *Quotation*, it has to be assigned in each class whose object has to provide data/text to the result.

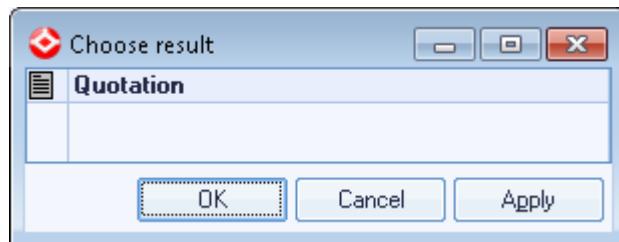
Results are administrated in the node *Results* in the structure tree of a class. To illustrate what a result is, a small example is created first.



Open the class *start* and insert a new result in the structure tree. To do so, you select the context menu item *New...* on the node *Results*.



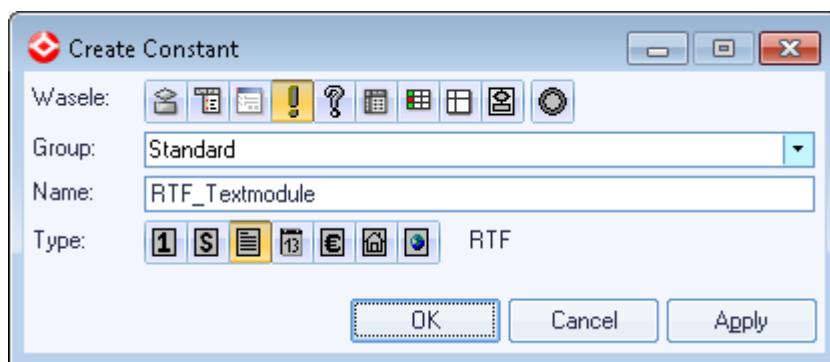
Select the *Result* with the name *Quotation* and confirm your selection with *OK*.



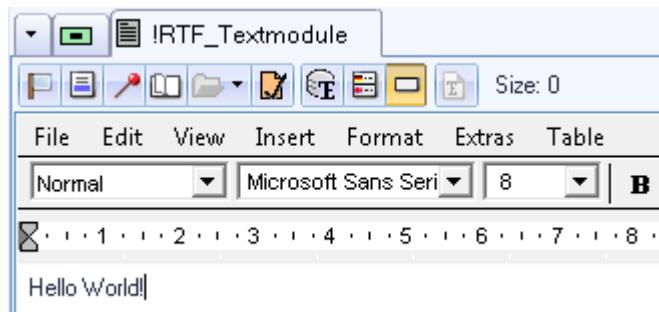
The “Quotation” that is defined in the frame is only the external description of the document. The result does not yet get contents or text modules that are necessary to generate a result object.



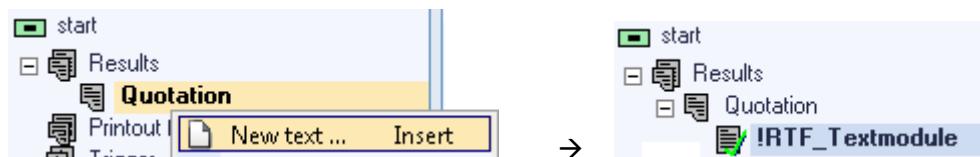
Open the class *start* and create a constant with the name “RTF_Textmodule” of the type RTF (same type as the result).



Change the text mode to *Monolingual text* Enter the text "Hello World!" in the constant editor.



Add the text module via the context menu item *New text...* to the result *Quotation* and select the RTF-constant from the Item selector dialog.



Now only one more step is missing for your first result. The knowledge base contains all parts that are necessary for a result, only the result generation has to be triggered.

In the configuration this has to be realized via extending the existing menu *Maintenance*. The result is generated via the function *WinOpenDoc(result name)* and displayed in a separate window.

[Function reference/Dialog functions/WinOpenDoc \(Result\[, Xpos, Ypos\[, Width, Height\]\]\)](#)

Open the menu *Maintenance* in the class *start* and extend it by the menu trigger "Create quotation". Create a new graphic constant with the name "IconQuotation" and load the graphic *Preview.bmp* from the folder *Graphics* of the training CD. Assign this graphic in the field *Icon* to the menu trigger and activate the option *Icon in toolbar*.

In order to trigger the result generation, you enter in the procedure editor of the menu item this call: *WinOpenDoc ("Quotation")*;

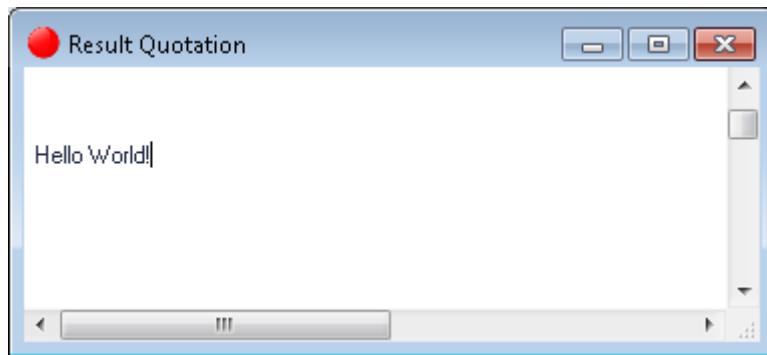


```
WinOpenDoc ("Quotation");
RETURN;
```

Start your application and test the menu item "Create quotation".



The following result window is displayed:



20.4. Fill, assign and display the quotation

Since you can imagine what a result is and which parts are needed now, we can turn to the realization of the real result, i.e. the generation of a quotation for the current model.

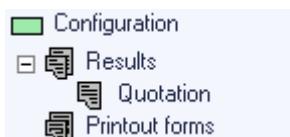


To do so, delete the result *Quotation* and the RTF-constant *RTF_Textmodule* in the class *start*.

For the result of the car configuration you need text modules with values from the classes *Car* (e.g. model, discount) and *Modules* (e.g. the price). Here you can see the importance of the base class *Configuration* again. If this class would not exist, the result should have to be entered in both classes.



Instead you assign *Quotation* to the node *Results* in the class *Configuration*.



20.4.1. Practice: Fill the result with values

In the target specification of the result the information about the model is displayed first. These are deposited in the class *Car*. For this reason the text module has to be in the same class.



Create the RTF-constant “!ResultHeader” in the class *Car*.



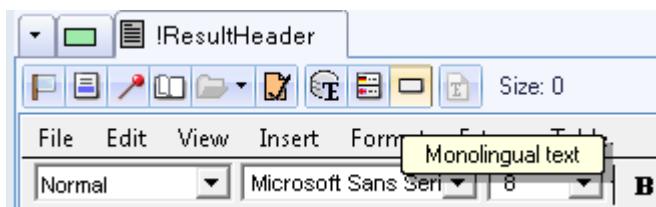
The RFT constant has three text modes. It can contain a Textelement of type RTF, a multilingual RFT text or a monolingual RFT text. In the multilingual modes, the RFT editor can be switched to other languages by the flag symbol.



To train the work with the RFT editor of camos Develop, we first create a monolingual resulttext. Later on the text can be converted into a textelement and of course also be translated



Activate the textmode to *monlingual text* in the RTF constant “!ResultHeader”.



The RTF-constant editor offers the same editing possibilities as other text processing programs. In addition to the known menu items *File*, *Edit*, *View*, *Insert* etc. the use of multilingual RTF-constants has to be mentioned.

[Workbench/wasele/Constant/Constant editor/RTF](#)

[Workbench/wasele/Form/Form elements/Editor/Editors/RTF Editor](#)

In order to access values from the configuration, *Cause variables* are integrated in the RTF-constant. Cause variables are wildcards in the text that are replaced by the current values during runtime when the text is generated.

According to the target stipulation in chapter 20.1 the result should contain the text “Quotation for a...” in the first line. Depending on the selected model its name has to be added. The dynamic decision is carried out via the integration of the cause variable *Self*.

Open the editor of the RTF-constant *!ResultHeader* and enter the text “Quotation for a “ (consider the blank space!).

Add a cause variable at the position after the text. To do so, you have the context menu item *Insert cause variable...*



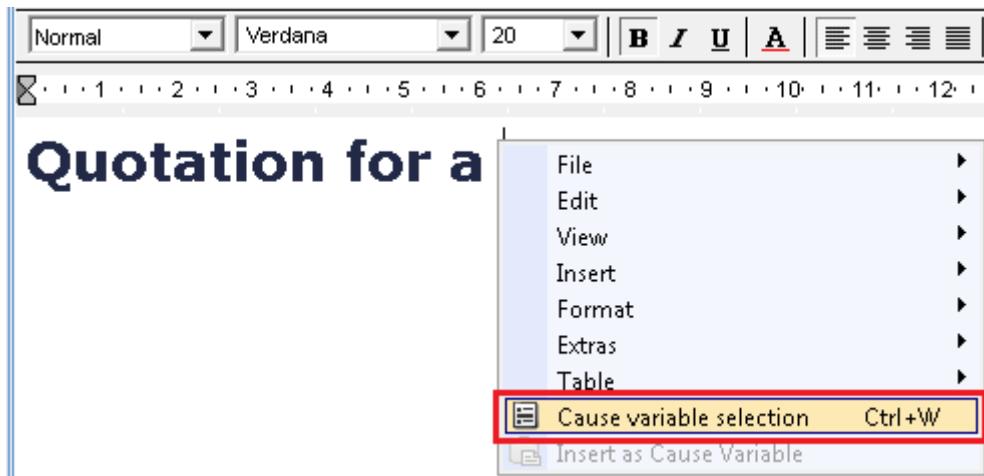


Figure 49: Integration of cause variables in RTF-constants

Since the text module `!ResultHeader` is in the class `Car` and the cause variable has to provide information for the car model, the cause variable `Self` is used. With the call of the function `WinOpenDoc()` the wildcard `Self` is replaced by the actual value, i.e. the name of the currently selected car model.



Insert the cause variable `Self`.



Note: In the RTF-editor cause variables are displayed in braces. Cause variables cannot be edited and additionally indicated by a dark gray typeface.

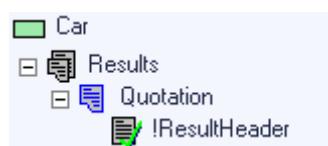
The manual entry of a cause variable in braces that is not inserted via the item selector dialog is not recognized as entry and therefore not allocated with a value during runtime!

The text module has to be first allocated to the result (as in the practice with "Hello World!") before the text can be displayed in the result.



Assign the RTF-constant `!ResultHeader` to the inherited result `Quotation` in the class `Car`.

You can either use the context menu item `New text...` and then select the desired constant or you drag the constant via D&D under the result.

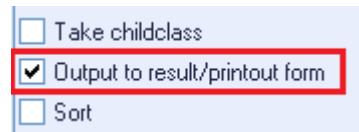


Start the application and test the result generation.



You will probably wonder why the just created text module is not displayed. The reason is that the component `_Car` has not been notified that it is relevant for the result. In order to do this, the flag `Output to result/printout form` has to be activated at the respective component.

Open the class `start` and set the option `Output to result / printout form` on the component `_Car`.



If this flag is set, the components in the structure tree are marked with the icon

Test your application and the creation of the result again. Select different models and check the model name in the result.



Extend the Output to result/printout form by the total price and the discount by inserting the cause variables `Discount` and `End price` with the accompanying text in the constant `!ResultHeader`.



• 2 • 3 • 4 • 5 • 6 • 7 • 8 • 9 • 10 • 11 • 12 • 13 • 1

Quotation for a {Self}

End price including transfer
and {Discount} discount: {EndPrice}

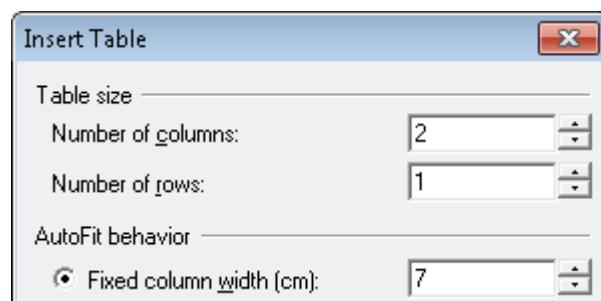
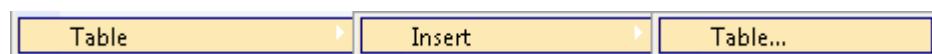
20.4.1.1. Practice: Display components

Have another look at the target result from the beginning chapter. Below the price output you will see a table with all selected components of the configuration.

This table has a header for the column headings and respectively an entry for the components and their price.

The header line of the table is entered in the text module `!ResultHeader` and the table lines in an extra constant which is inherited from the class Modules in all child classes.

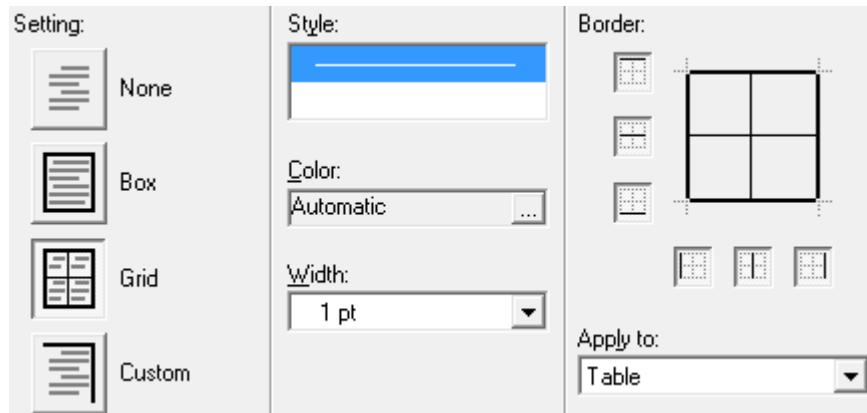
 First you extend the constant `!ResultHeader` by a single-line table. To do so, you use the context menu item *Table -> Insert -> Table...*.



 Set the number of columns and rows as well as the width of the table columns in the dialog *Insert Table*. Confirm with *OK*.

 Fill the cells with the column headings "Description" and "Price" according to the screenshot from chapter 20.1.

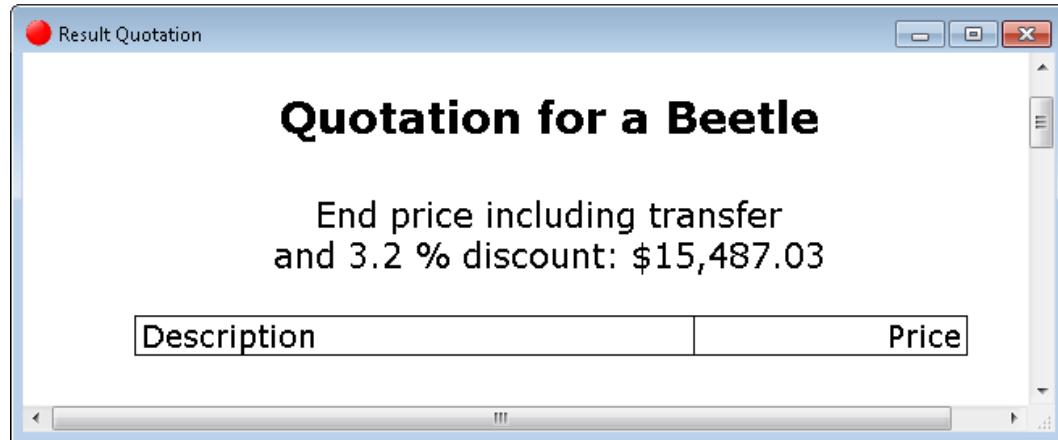
 Select the table and via the context menu item *Table -> Table Properties*. Select the tab page *Table* and therein the Button *Borders and Shading*. On the tab page *Borders* you activate the option *Grid*. Define a width of 1 pt and confirm the settings with *OK*.



Change the text alignment of the price column to *right justified* and, if necessary, the alignment of the table to *Centered*. Therefore call the context menu item *Table -> Table properties*. Set the *Alignment* to *Center* on the tab page *Table*.



Test your result in the application.



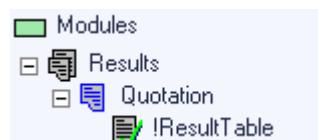
In the next step the components are added. Since it is not known how many and which components the user selects, the table must be built dynamically.

The precondition for a component to be able to appear in the table is that an object of this component exists. Basically all components can be selected, therefore a text module is created in the class *component parts* and therefore inherited to all component parts.

Create a new RTF-constant with the name “!ResultTable” in the class *Modules*. Assign this constant as a result text to the inherited result *Quotation*.



In the next step the components are added.



By these two steps all modules are involved in the result *Quotation*. The constant “ResultTable” is still empty. Here we want the name and the prices of the selected components to be displayed.

Copy the table line from the constant *!ResultHeader* and insert it in the new RTF-constant in class *Modules*.



This procedure guarantees that a continuous table is originated in the result and that there are no columns with different widths.

Please consider that no blank line exists above or below the table line, because otherwise the table would not be continuous.



The contents of the two columns must now be adapted. Each module object must enter its name and its price in the table

Via cause variables you can set the properties of the module object.



Set the cause variable *Self* as name of the module and set *Price* for its price.

{Self}	{Price}
--------	---------

This completes the list of modules: Each object of the type Modules will supply its individual row at the result quotation and the individual lines are automatically connected below each other so that a table is created.

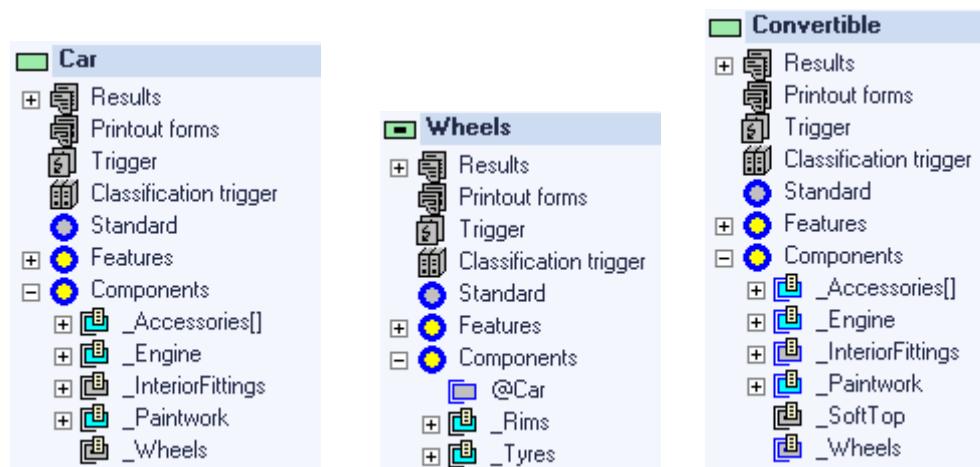
One important step is missing before the result can be generated. Can you remember the beginning of the chapter? Correct: The option *Output to result/printout form* has to be activated on all components.



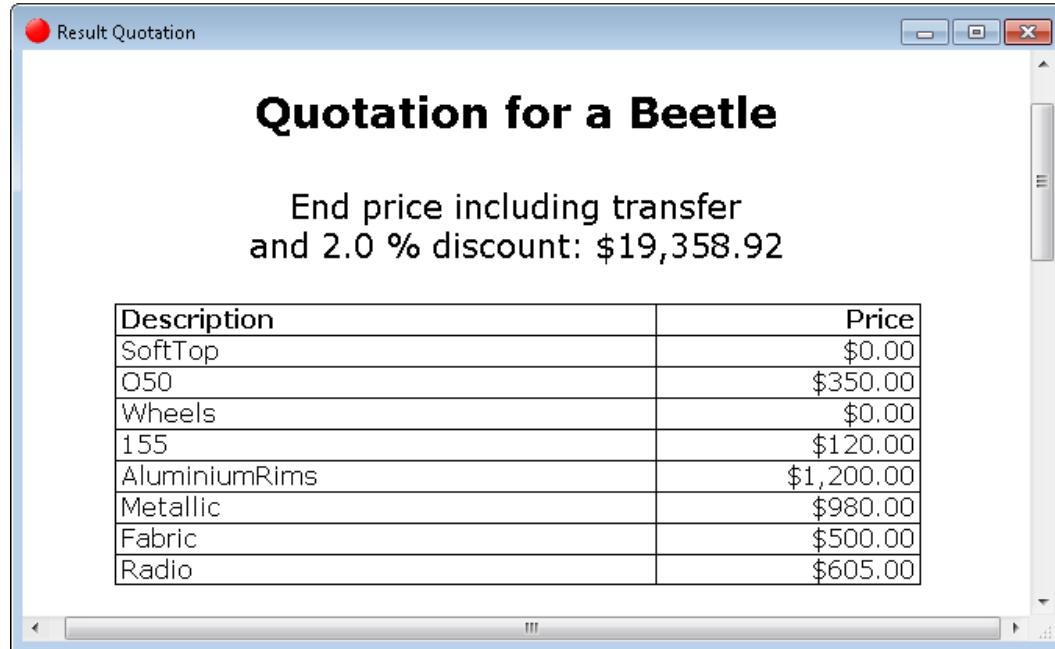
Set the option *Output to result/printout form* on all components in the class *Car*, on the tyres and rims in the class *Wheels* and on the soft top in the class *Convertible*.

You can change this option manually or automated via *find properties*, see chapter 13.5.1

The icon of the components in the structure tree changes to



Start the application, configure a model and select the menu item *Create quotation*. Check if all selected components and their prices are displayed in your result.



Attention, frequent error source: If the option *Output to result/printout form* is not set for a component (e.g. wheels), this object and its child components are missing in the result – even if the *Output to result/printout form* is enabled on the components that are defined in the class *Wheels*.



20.4.2. Practice: Include the product picture of the model

If everything is displayed to your satisfaction, we proceed with the last part of the practice: To display the model image.

The problem is that the graphic constants with the product images are in the class *start*. The text module in which the image has to be displayed is however in the class *Car*.

There are several solution possibilities: prerequisite is a so-called **typecast**.

20.4.2.1. What is a typecast?

A **typecast** is a conversion from one type to another type. In camos Develop a typecast is used to access information from external classes.

When you access *wasele* from objects, that can not be addressed directly by the pointer or the component, because the *wasele* is defined later in a derived class, the object has to be specialized via the typecast: *Object:Classname.Wasele*

For example the SoftTop of the Beetle can be opened with the method *SoftTopOpen()*. This method was only defined in the class *Convertible*, so it can only be used in the class *Beetle* but not in the *HardTop*-classes. To call the method *SoftTopOpen()* without syntax error, it has to be simulated, that the component *_Car* contains an object of the type *Convertible*: *_Car:Convertible.SoftTopOpen()*. If the class name contains blanks, then it has to be put in inverted commas.

When you access constants, the class, that is casted, don't have to exist as an object, because the constants are fixed deposited in the knowledge base. Therefore the declaration is omitted in the syntax, for example "*IconContainer*".!Icon Allowed



Test the use of the typecast in the result with integrating `:start.!ImageBeetle` via a cause variable in the RTF-constant `!ResultHeader`.

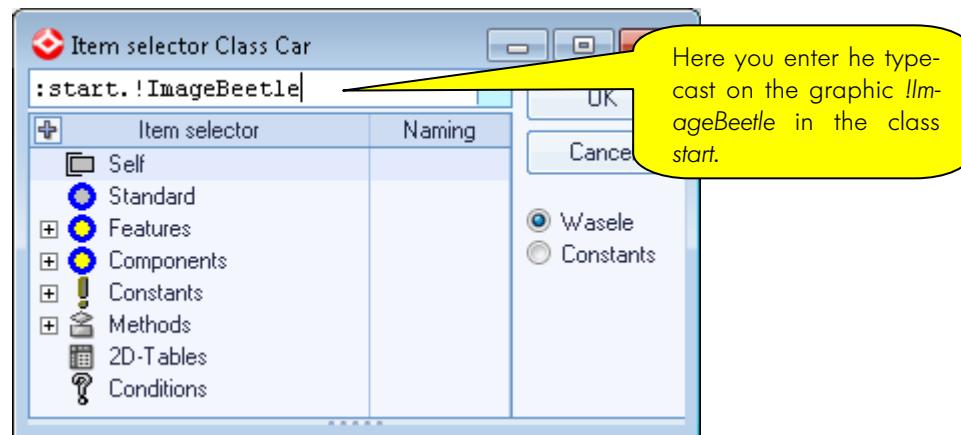
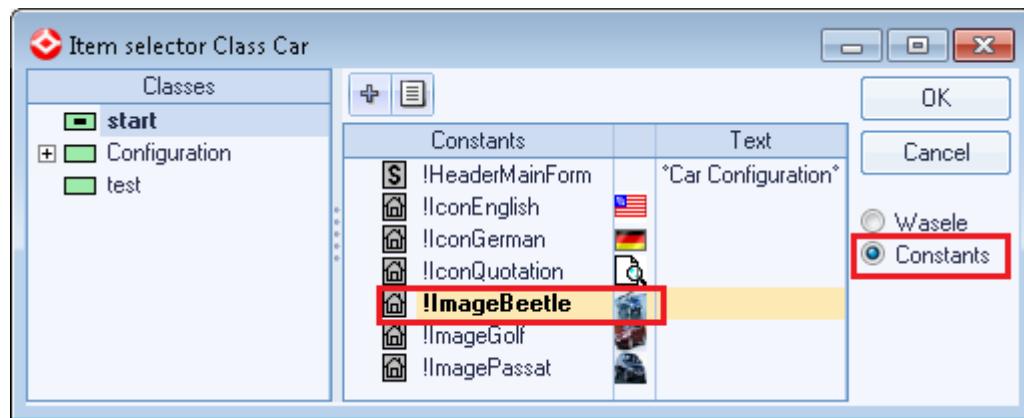


Figure 50: Definition of a typecast in the item selector

You can enter the required syntax manually or switch to the modus *constant* and choose here the *constant* `!PictureBeetle` from the class `start`. This way prevents typing errors, and you do not have to remember the class- or waselename.



... 3 ... 4 ... 5 ... 6 ... 7 ... 8 ... 9 ... 10 ... 11 ... 12

Quotation for a {Self}

{`:start.!ImageBeetle`}



Test the application and the result.

You will notice that the image of the Beetle is also displayed with the configuration of a Golf. Since in the result is not differentiated which model was selected, always the same graphic is displayed.

20.4.2.2. Display the product image by calling a method

The solution would be a mechanism that fades in the correct graphic with regard to the selected model.

To do so, a method is created that queries the model of the car and then returns the matching graphic. More efficient than with IF THEN ELSE queries (as in chapter 19), you can work with the CASE instruction in this case.

Create the method `giveImage()` in the class `Car` and enter the following procedure code. Change the type of the return value to graphic so that the method returns the required data with the correct type.



Parameter	Class	Call by	Variable	Class	Call by
1 Return			1		
2					

External callable
 Side effects allowed
 KNB external

```

CASE Self
IS 'Beetle' DO
  RETURN :start.!ImageBeetle;
IS 'Golf' DO
  RETURN :start.!ImageGolf;
IS 'Passat' DO
  RETURN :start.!ImagePassat;
IS DO
  RETURN NOVALUE;
ENDCASE;
RETURN;
    
```

Figure 51: Procedure editor – Method without side effects

The operators IsA and NotIsA

The operator `IsA` is used to check whether an object is of a particular type, e.g..

`_Motor IsA „Otto“`

This expression returns 1, when the engine object has been instantiated of the specified class (or a derived class). A 1 would be returned, if the current motor is an Ottomotor with 50 KW. If the engine a diesel engine with 70 or 110 kw, a 0 would be returned, since the classes D70 and D110 are not derived from the class Otto.

The following test would provide the 1 for all engine variants:

`Motor IsA „Motor“`

Similarly, there is the operator `NotIsA`, it delivers a 1 when the tested object is not derived from the given class.

?

[Basics/Control Statements/CASE-ENDCASE](#)

?

[Basics/Operators/List-Operators](#)

Back to the method `giveImage()`: it is checked, which type the object `Self` (car) has. Is `Self` of the type "Beetle", then the graphic constant containing the image of the beetle is given back, etc.

Since this graphic is in the class start, it is casted (`:start.!ImageBeetle`). In case of the Golf correspondingly the image of the Golf is returned etc.



Additionally the option `Side effects allowed` has to be deactivated, because only simple expressions are allowed in the context in which the method has to be used (Rtf Editor). I.e. the procedure code is not allowed to affect other objects.



Delete the cause variable `{:start.!ImageBeetle}` in the constant `!ResultHeader` and replace the cause variable by the method call `giveImage()` via the Item selector dialog.

Test the result output with all models.

The screenshot shows a Windows-style dialog box titled "Result Quotation". The main title is "Quotation for a Golf". Below the title is a photograph of a red Volkswagen Golf. Underneath the photo, the text reads: "End price including transfer and 2.0 % discount: \$18,326.00". At the bottom of the dialog is a table with two columns: "Description" and "Price". There is one row in the table with the values "O120" and "\$800.00".

Description	Price
O120	\$800.00

The result provides all information for the configured model. In comparison to the result from target stipulation there are still slight differences.

If you want to bring your result into line with one from chapter 20.1 or if you are not yet satisfied with the display or operation, you can find a variety of optimization suggestions in the following chapter.

20.5. Tips & Tricks

20.5.1. Display the naming instead of the name

Momentarily the result displays the model name with the name of the class.

Quotation for a Beetle

In the target stipulation a different name is displayed? How can that be? What has to be done to get the same display?

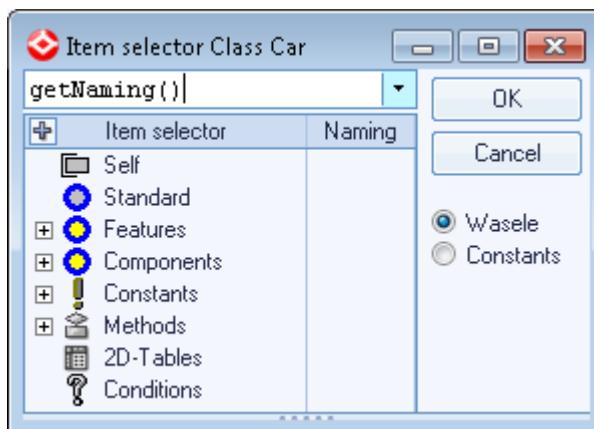
Quotation for a New Beetle Cabriolet

The cause variable `{Self}` was exchanged in the text module `!ResultHeader` by a different cause variable which displays the naming of the class.

For this the function `GetNaming()` is used. This function provides without specifying parameters the naming of the class of the current object in the set dialog language.

 [Function reference/Other functions/GetNaming \(\[Class name\[, waseleName\]\]\)](#)

Replace the cause variable `{Self}` in the constant `!ResultHeader` of the class `Car` by entering the cause variable `{GetNaming()}` via the Item selector dialog.



In the same way the namings of the components can be displayed. To do so, also replace the cause variable `{Self}` in the constant `!ResultTable` by the cause variable `{GetNaming()}`.

20.5.2. Optimize result window

The result is opened – independent of the position of the dialog (= the MainForm) – in the upper left corner. In addition the result window has to be always increased manually so that the content is completely visible.

The meaning of a result is to display/save the Is-condition of a configuration without being able to make further changes at the same time.

20.5.2.1. Position and size

Therefore it would be useful to open the window right away in a determined size and directly above the MainForm.

Momentarily the result is opened via the function call *WinOpenDoc()*. If you search for this function in the online help, you will find out that this function can process further optional call parameters.

 *Function reference/Dialog Functions/WinOpenDoc (Result[, Xpos, Ypos[, Width, Height]])*

The call *WinOpenDoc()* that is carried out in the menu *Maintenance* can be extended as follows:

```
#Parameter: Result name, X-Position, Y-Position, Width, Height  
WinOpenDoc( "Quotation" , 0 , 0 , 800 , 600 );
```

20.5.2.2. Modal windows

If the result window is opened, it has no connection to the MainForm. I.e. that closing or minimizing the dialog has no effect on the result. Additionally further changes on the configuration can be made with an opened result.

This should not be possible. An opened result should always be displayed in the foreground. Additionally the MainForm should be minimized or maximized with the result.

Via the function *WinStartModal()* the result can be opened **modal**. I.e. the result window is bound to the main dialog and therefore always opened in the foreground. If the result is minimized, the MainForm is also minimized.

A so-called form handle is transferred to *WinStartModal()*. The form handle identifies an opened form/result uniquely and is provided by the function *WinOpenDoc()*.

 *Function reference/Dialog functions/WinStartModal (Form handle)*

 Extend the call of the result as follows:

```
WinStartModal(WinOpenDoc( 'Quotation' , 0 , 0 , 800 , 600 ));
```

If the result is opened with this call, but no car object exists at that time, the following error message occurs:



For this reason the result generation should only be possible if a car object exists.

As already introduced in chapter 9.3.1, the editor of a menu item contains the field Enabled with which the state of the menu item is controlled.

In order to make the functionality of the menu item “Create quotation” only possible with a selected model (`_Car` equal ANYVALUE), you can enter a simple expression that checks the existence of an object of the component `_Car` and return either 0 or 1.

Hook:	<input type="text" value="0"/>
Enabled:	<input type="text" value="Car = ANYVALUE"/>

If the expression returns 1, the menu item is active, otherwise inactive.



20.5.3. How to hide individual components

In the result, components that are not charged for the customer are displayed, too. E.g. the wheels or the soft top with the beetle are displayed with 0.00 \$.

If the display of free components should disturb you, there are the following possibilities to fade them out.

20.5.3.1. Reset the Output to result/printout form

Since only components are displayed on whose component the flag *Output to result/printout form* is set, this is also a way to affect the display in the result.

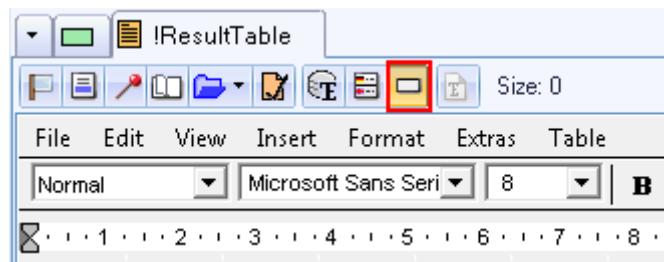
If the option *Output to result/printout form* is reset on a component, neither this component nor the components that are contained in the class that belongs to it are considered in the result.

Therefore this solution would only be possible for the sunroof but not possible for the wheels, because the tyres and rims would then also be faded out.

20.5.3.2. Define a text module without content

A further possibility would be to overload the RTF-text constant `!TableRow` in the object classes `Sunroof` and `Wheels` and to delete the contents (the table row).

If the constant is in the mode *Textelement*, change the type to *Monolingual text*. Then delete the content of the constant.



With this the text module still exists in the result, but – since no contents exist – not visible.

20.5.3.3. MayNot rule on text module

The favored solution alternative for the car configurator is to create a rule on the text modules. The display of the text module is forbidden in the classes in which the text module has to be faded out.

The MayNot-rule is locally defined on the text module in the respective class and has no affect on the components (= child objects) of this class (see 20.5.3.2).



To do so, you open the class *Wheels* and create a MayNot-rule on the text module *!TableRow* (via the context menu item *New rule implicit -> May not*).

A rule always needs a condition. The condition displays the time of the validity. Since the wheels should not be displayed in any case (the display is always forbidden), the expression valid condition is 1.



On the rule origin you create a new expression via the context menu item *New expression* and enter 1 (= always) as condition.

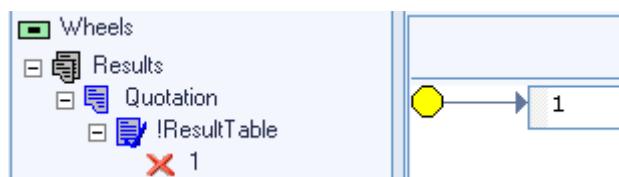


Figure 52: Rule “always forbidden”



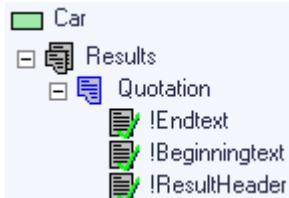
Proceed in the same way with the text module in the class *SoftTop*.

20.5.4. Change the order of the text modules in the result

It is possible that a class has several results with several text modules. In this case the text modules are displayed in the result in the order in which they are sorted under the result node.

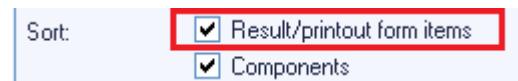
In the below case first the contents of the text module *!Endtext*, then the one of the *!Beginningtext* and finally the *!ResultHeader* would be displayed.

Wrong sorting of the text modules:

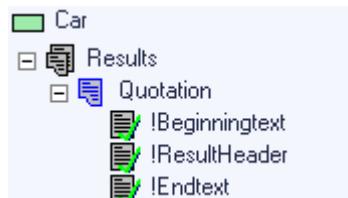


Via setting the option *Sort -> Result/printout form items* in the class editor, you have the possibility to sort the text modules via D&D.

Activate local sort:



Resort text modules via D&D in the right order:



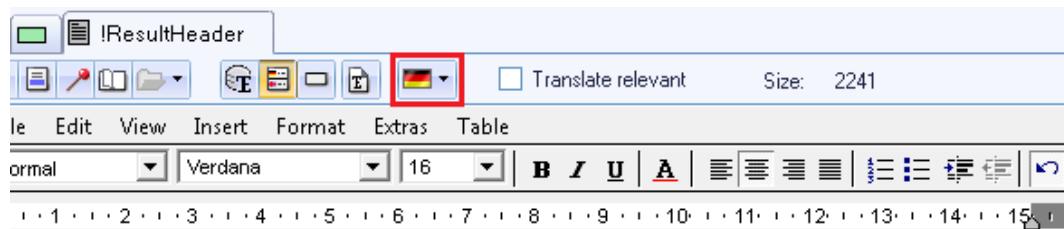
20.5.5. Adjust the result to consider the dialog language

As already mentioned, text modules of a result can also be defined multilingual. Via the country flag of the main language/secondary languages can be switched between the individual editors.

Change the text mode from Monolingual text to Multilingual text. The content from the main language English is maintained in the process.

Activate the secondary language German and translate the text module *!ResultHeader*.





Angebot für einen {getNaming()}



Test the Output to result/printout form with a changed dialog language.

You will notice that the multilingual text constant is not considered with the switching of the language. The reason is, that the function *LanguageDialogSet()* which controls the switching of the language in the dialog has no affect on the result.

In order to switch the language of a result or its text modules, the function *LanguageDocSet()* is used. The respective country code is transferred to this function, corresponding to the desired change.

Since the switching of the language in the result should function analog to the switching of the dialog language, the menu items for switching the language are extended (in the menu Maintenance).

Menu trigger English:	Menu trigger German:
<pre>LanguageDialogSet(1); LanguageDocSet(1); RETURN;</pre>	<pre>LanguageDialogSet(49); LanguageDocSet(49); RETURN;</pre>

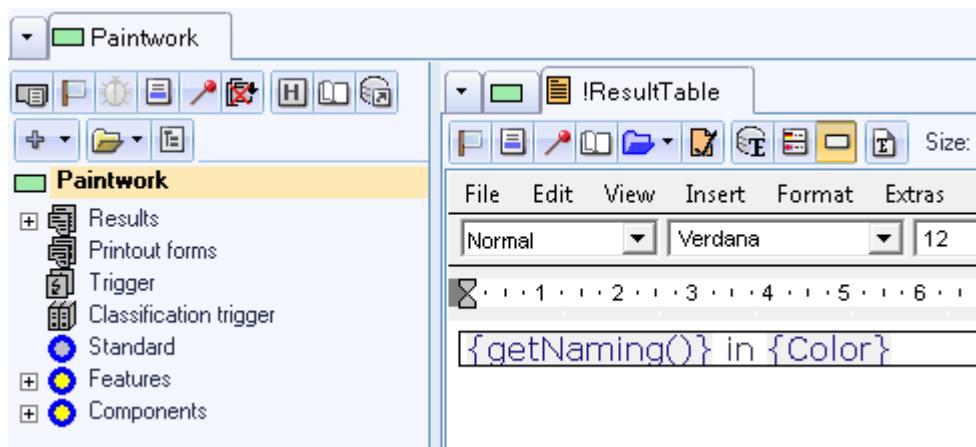
Via the change of the cause variable *{Self}* in the text module *!TableRow* to *{GetNaming()}*, the switching of the language is carried out automatically as far as German namings were deposited.

[Function reference/Language Functions/LanguageDocSet \(Country code\)](#)

20.5.6. Display the paintwork color and engine power

The configuration offers further information that is not listed in the result. There are e.g. the methods *Color* and *Power*. In order to display this information in the quotation, too, only an extension of the corresponding text module is necessary.

Color is a feature of the class *Paintwork*, therefore the text module *!TableRow* is overloaded at this position and extended by the cause variable *{Color}*.



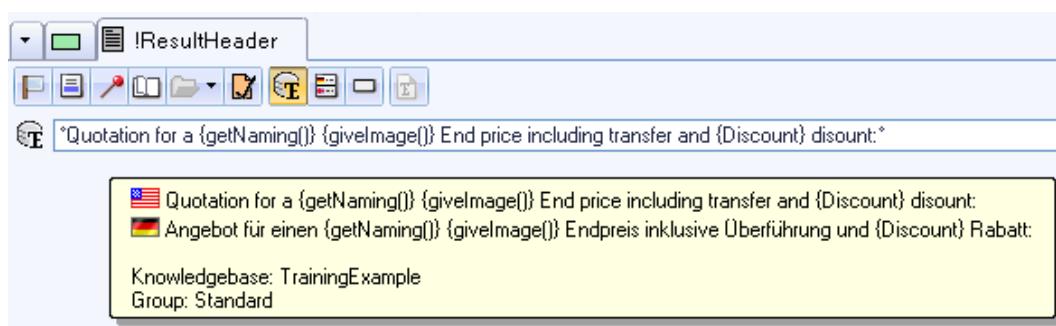
In the same way the text module can be adjusted in the class *Engine*.

20.6. Convert RTF-Text to a Textelement

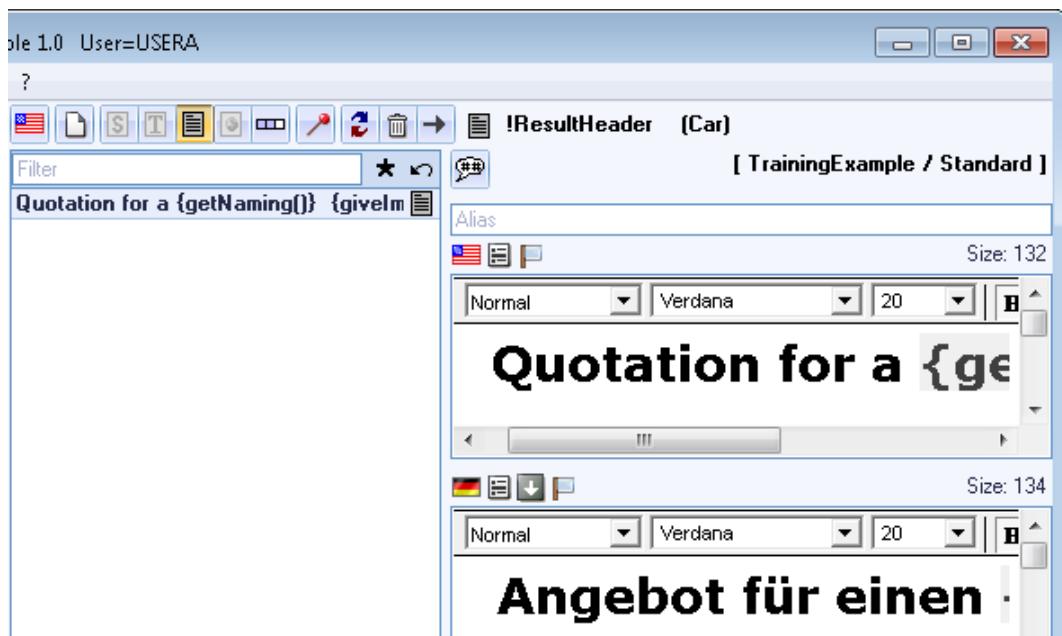
The text modules *!ResultHeader* and *!ResultTable* were processed in the RFT editor. Since all other texts in the knowledge base are deposited as textelements, this should be continued also for the result texts.

To do so the existing RFT text can simply be converted into a Textelement.

Open the editor of the RFT-constant *!resultheader* in the class *car*. Click on the icon *create textelement* and choose in the context of the knowledge base *TrainingExample* and the group *Standard*.



To change the text, you must open the text in the textmanager.



If the Textmanager is opened via the icon in the camos Develop toolbar, the Textelement can be found only when the type RFT is enabled in the toolbar. When you open the textmanger from the RFT-constant, the view is related to the context, therefore you automatically see only the RFT texts, because only those can be used in a RFT constant.

The entering of cause variables is done via the icon . When the textelement was opened in the context you can select the desired EFG from the waseleselect-dialog. Without the context this is not possible, in this case the desired cause variable has to be typed manually.



For the constant *!ResultTable* in the Modules you can convert the content to a textelement in the same way.

Please note that this has to be done manually for all overloaded RFT-constants. Once the wasele is overloaded changes at the original element do not affect the overloaded element any more!

To find out where a wasele was overloaded, the context menu point *display overloaded* is used. Here you can see in which class the wasele was defined originally (grey new-icon) and where it is overloaded (orange new-icon). The current class is displayed with bold font.

Overload list Class: Modules RTF-constant: !ResultTable			
		Context	Remark
□	Configuration		
□	Modules	★	
□	Engine	□	RTF-constant ()
□	Paintwork	□	RTF-constant in

The transformation into a textelement must therefore be done in the classes *Engine* and *Paintwork* aswell.

Open the constant *!ResultTable* in the class *Modules*, *Engine* and *Paint* and transform the content with the icon  *create textelement* in a textelement.



20.7. Repetition

- Results are defined in the frame.
- You can make language-specific adjustments for Layout, formats and headers/footers are.
- Results consist of individual text modules.
- Results can be of the type String, RTF or HTML. Attention: String results do not support cause variables.
- Cause variables in results are replaced by the current value during runtime.
- The result is defined in the structure tree of the class and text modules are assigned to it.
- For each component (and its child objects) that has to provide the result with text modules, the flag *Output to result/printout form* has to be activated.
- If a component does contribute anything to the result, but has child objects that are displayed, the flag *Output to result/printout form* still has to be activated on the component.
- The result generation is started via the function *WinOpenDoc*. With this a check is carried out as from the call object on all objects whose components have the flag *Output to result/printout form*. If it is a result with the transferred name and the allocated text modules are added to the result document.
- RTF-text modules can be defined multilingual.
- The switching of the dialog language has no affect on the document language of the result. To do so, the function *LanguageDocSet(Country code)* is needed.
- Via the function *WinStartModal(Handle)* a result- or dialog window can be opened modally.
- Via the function *WinOpenDoc()* additionally the size and position of the result can be set.

21. Multi-user access

21.1. Intention

If several users use the same camos Develop database, they can theoretically work simultaneously on the same database (so-called multi-user operation). In this case, you have to ensure that a class, the frame, etc. cannot be edited by more than one developer at the same time. Therefore, camos Develop includes an access maintenance for all elements of a knowledge base.

In this chapter you will learn how this access maintenance works and what you have to do (especially if several developers are working on the knowledge base at the same time) in order to be able to edit the individual elements of the knowledge base.

21.2. Basics of the access maintenance

In general, all elements of a knowledge base (its properties, the frame as well as the individual classes) have to be reserved in order to be edited. This is carried out via the icon  respectively . As long as the element is reserved, other users have only reading access to the last released version of the element.

By reserving an element e.g. a class, a copy of this element is created internally. Then the user works with this copy and either cancels the changes (the original status is restored) or applies them (the changed copy is applied as original).



In order to apply the changes of a reserved element, it is released. Since the integration of the ticket system into the camos Develop development environment, classes are either first released in the ticket . With this, the content is public for all developers who also work in this ticket. With this approach, the ticket has to be released to provide the changes for all users. Thus all the included classes will be released publicly and the state of the ticket is changed to "Done". For the next editing of the knowledge base, a new ticket is created and activated.

Alternatively the classes can be released and unlinked from the ticket  to make the changes visible for all developers. In doing so, the ticket does not have to be set finished.

All data is automatically saved in the database in a certain *cycle of saving* (see *User options*, default is 20 seconds). Therefore releasing is not necessary to (temporarily) save data, but it is used to apply changes on elements specific / all users.



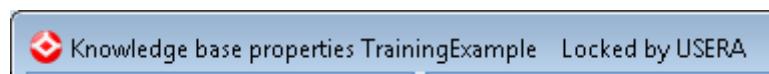
If changes of a reserved element should not be assumed, it is discarded. Here are again two possibilities to do this. Either to *Discard the recent changes*  to cancel all changes since reserving or refreshing the class. The class will be released in the ticket.

The other possibility is to *Discard and unlink from the ticket* . Thereby all the classes are canceled as well, the class will be publicly (for all users) released though.

21.2.1. Access to properties of the knowledge base

If you want to change the *properties of the knowledge base* in order to e.g. adjust the start class in the options, the properties have to be reserved by you first. This happens automatically on opening the property dialog. If another user already edits the properties, all tab pages of the dialog are set to read-only.

On the right upper side of the dialog you can see which user currently reserved the properties - possibly it is your own username.



To apply the changes, click on the button *Release and close*. To cancel the changes, click on the button *Discard and close*.

21.2.2. Access to the frame properties

To edit the settings of an area, it has to be reserved (double click in the first column or context menu). Which users have write permission at the frame, is defined in the frame maintenance, see chapter 28.4.



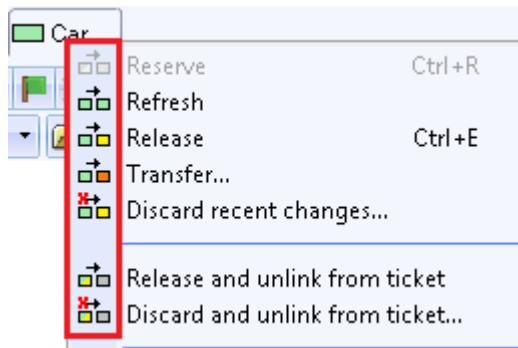
An area that has been reserved by you is displayed green in the first column. If an area is reserved by another camos Develop developer, the area is displayed for you orange pigmented.

21.2.3. Access to classes

Newly created classes are automatically reserved by you. The current status of the class can be recognized by the color of the class icon:

- (green) class is reserved by the current user
- (yellow) class is released in the ticket
- (gray) class is released publicly (for all users)
- (orange) class is reserved by a different user or reserved in another ticket

The reserving, releasing etc. of a class is carried out in the context menu of the class in the class tree or alternatively via context menu of the tab page of an opened class in the workbench:



In addition to the functions *Reserve*, *Release* and *Discard* the context menu of the class has two further functions:

↗ Refresh

If a class is reserved by you, the class state before reserving is visible for all other users. This state is also used for the test interpreter in the knowledge bases of the other users. Via the menu item Refresh the reserved and modified class is updated to the existing state for the other users. It remains however reserved so that you can still edit it.

↗ Transfer

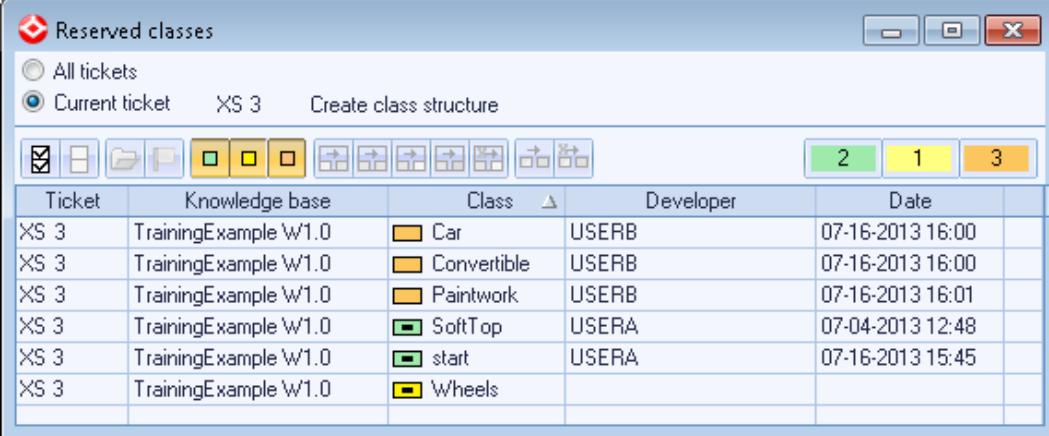
Via this menu item reserved classes can be transferred to another user. Selectable are all users, which are defined as developers in the current ticket. The class is automatically reserved for the user the class is transferred to. The previous developer sees the state of the classes before his changes.

ⓘ Class tree/Class types/Class types/Context menu of the class/Access administration

21.3. Tips & Tricks

21.3.1. Search for reserved classes

Via the icon in the toolbar above the class tree you can search for all reserved and / or released in the ticket and / or from other users reserved classes.



The screenshot shows a dialog box titled "Reserved classes". At the top, there are two radio buttons: "All tickets" (unchecked) and "Current ticket" (checked). Next to them are buttons for "XS 3" and "Create class structure". Below the buttons is a toolbar with icons for various operations like search, refresh, and export. To the right of the toolbar are three colored boxes labeled "2" (green), "1" (yellow), and "3" (orange). The main area is a table with columns: "Ticket", "Knowledge base", "Class", "Developer", and "Date". The table contains six rows, each corresponding to a different class reserved for ticket XS 3.

Ticket	Knowledge base	Class	Developer	Date
XS 3	TrainingExample W1.0	Car	USERB	07-16-2013 16:00
XS 3	TrainingExample W1.0	Convertible	USERB	07-16-2013 16:00
XS 3	TrainingExample W1.0	Paintwork	USERB	07-16-2013 16:01
XS 3	TrainingExample W1.0	SoftTop	USERA	07-04-2013 12:48
XS 3	TrainingExample W1.0	start	USERA	07-16-2013 15:45
XS 3	TrainingExample W1.0	Wheels		

21.3.2. Reserving via hotkey

If you marked a class in the class tree, you can also reserve it via the hotkey **Ctrl+R**. The same is valid if you opened a released class and the focus is within the workbench of the class.

The release is carried out via the hotkey **Ctrl+E**.

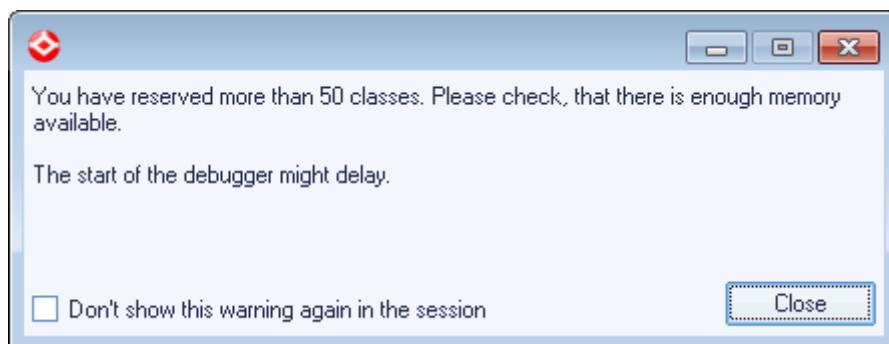
21.3.3. Versioning of classes

Via the modes *Reserve* and *Release/Discard* a kind of versioning of a class can be realized, because there is no Undo function for such extensive changes.

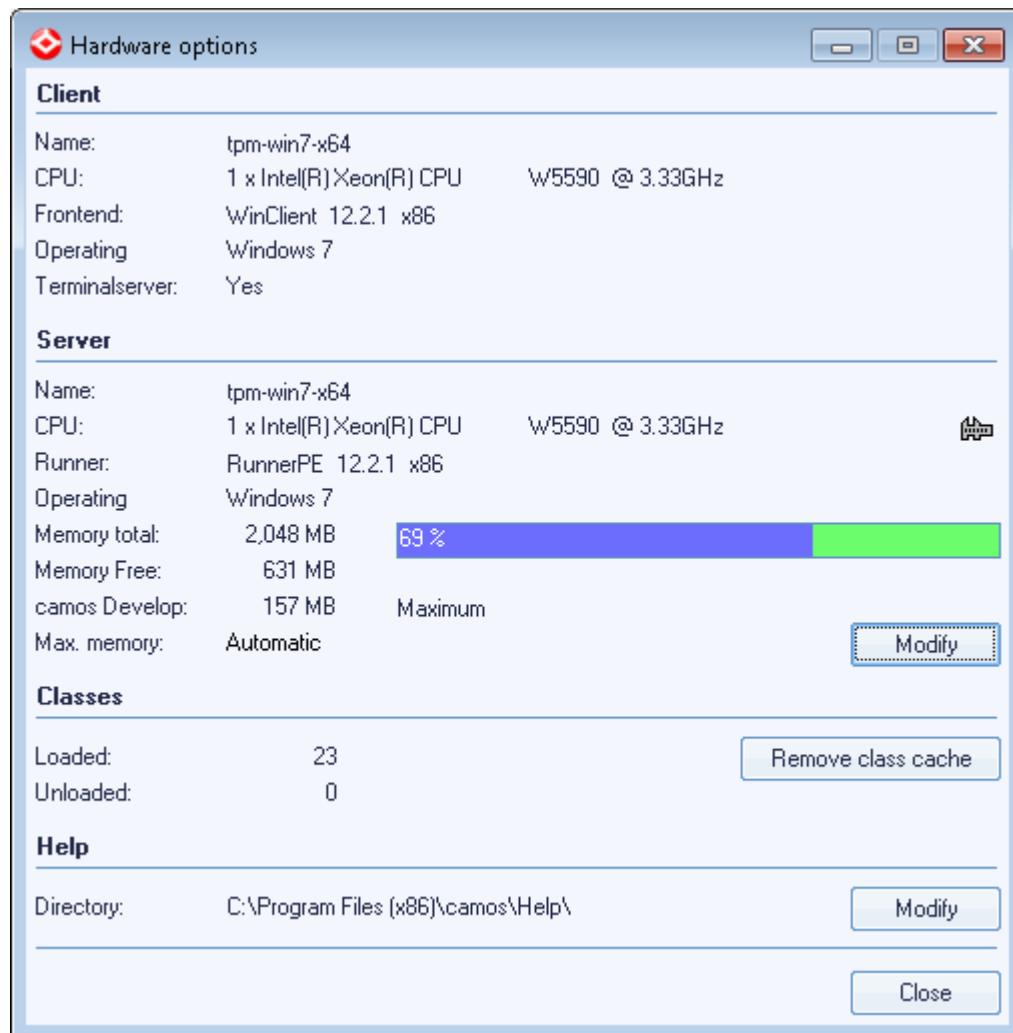
If you plan e.g. a major change on a form and you are not quite sure about the effects it may have, you first release the class in the working state. Then you reserve the class again, carry out the changes and discard them, if necessary.

21.3.4. Optimize the performance of the development environment

Classes that you are currently not editing should principally be released. This does not only save queries with/from other developers. Additionally every reserved class consumes more storage space in the main memory and on the database. Therefore it slows the speed of the development system down, especially with the debugger start. For this reason a corresponding comment message is displayed if you want to reserve more than 50 classes at the same time.



The number of the currently kept or swapped classes in the main memory can be viewed in the dialog *Extras -> Hardware options*.



>Main menu/Extras/Hardware options

21.4. Repetition

- All classes, the frame as well as the properties of the knowledge base are protected via an access maintenance against the parallel editing by several users.
- Before an element can be edited, it has to be (explicitly) reserved for the own user. With reserving a copy of the current condition is created.
- After the changes were carried out, you can decide if these changes have to be released (saved) or discarded (restore original condition).
- The reserving condition of classes can be recognized by the color of the class icon: green = reserved, yellow = releases in the ticket, gray = released publicly, orange = reserved by a different user / in another ticket.
- As long as an element is reserved by a user, other users can only access it reading; the state at the time of the reserving is valid. For other users the changes are only visible or effective after releasing (with classes also updating).

22. Versioning

22.1. Intention

The knowledge base now reached a state in which all minimum requirements to a car configurator were set: The product structure images the three possible models, the configurable modules are visible on the form and usefully ruled. The application surface is multilingual and user-friendly, the price- and discount calculation is implemented and so is the creation of a quotation.

Therefore the application can now be released for the test phase or the pilot use in the sales or in the web shop. In this chapter you will learn how different versions of a knowledge base are created and administrated.

22.2. What are work- and release versions?

The knowledge base in which you are currently developing is the working version 1.0. Working versions are named with a W for example when creating a KIF. The w stands for working version. You can e.g. recognize a working version by the gray icon in front of the knowledge base name in the class tree:



Working versions are the knowledge bases in which you can develop actively. Release versions however fix a certain development state and further changes cannot be made any longer.

Which knowledge base versions exist, can be seen under the main menu *Knowledge base -> Administrate*.

Administration of knowledge bases

Show all knowledge bases Show only knowledge bases from the project TrainingExample 1.0

Knowledge bases	Naming	Category	Version	Runner	Develop	?	LE
Berechnungen			1.1	12.2.0B21	12.2w12	0	0
camos.Basic			1.0	12.2.0B21	12.2w12	0	0
camos.DocViewer							
camos.Toolbox							
Configurator							
Haupt_Wissensbasis							
Hilfe							
Masken							

Figure 53: Administrate knowledge bases

Please note the option *Show all knowledge bases* and *Show only knowledge bases from the project X*. By default only the knowledge bases of the currently opened project are displayed. With this option, only the knowledge bases and knowledge base versions which were included into the project are shown.



This is important when creating a new version. Because the new versions are not added to the current project automatically, they are not visible as long as the option *Show only knowledge bases from the project X* is set.

Prerequisite for the following actions is the set option *Show all knowledge bases*.

All knowledge bases that exist in the database are listed in the left table of the dialog. If a knowledge base is selected here, all versions that exist to this knowledge base are listed in the right table. Work versions are indicated with a hammer and release versions with a red dot .

Additionally is displayed with which version of camos Develop the respective version was last changed. The number of pages for the developer- and user documentation is displayed in the last two columns.

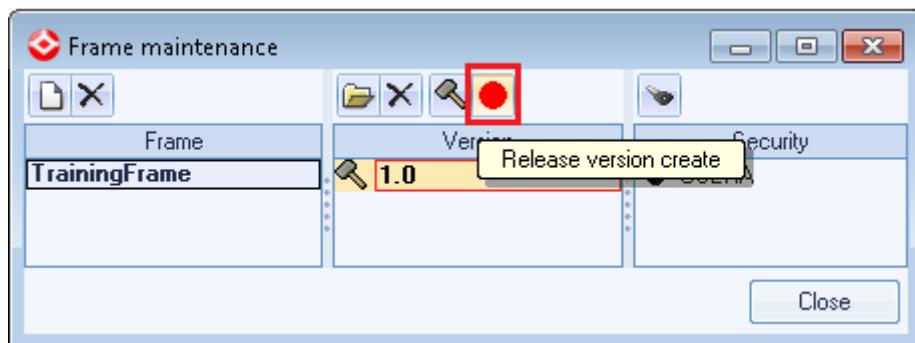
As support for the developers, who work with a lot of knowledge bases, they can define a meaningful naming to the knowledge base. Also you can define categories for the knowledge bases and order them to the categories. Using the filter then it is possible to restrict the list of knowledge bases according to certain categories.

22.3. Practice: Creating a new release version

To fix the current state of development, you create a release version of the knowledge base *TrainingExample*. Because a release version of a knowledge base can only be created with a release version of a frame, the frame has to be released first.



Open the *Frame maintenance* via the main menu *Maintainances*. Select the version 1.0 of the *TrainingFrame* and click on the icon to change this frame into a release version. Close the Frame maintenance afterwards.



The creation of a release-version is only possible when all classes are released. Make sure that the knowledge base is free from syntax errors!

Select all the reserved classes in the class tree (multi selection with pressed Ctrl key) or use the dialog *Reserved classes* to release all the classes.



Open the ticket system via the icon and select the icon *Release ticket*. Thus all classes will be released publicly and the ticket is finished (it will receive the state *Done*).

In the process of releasing the knowledge base, it must be freezed first.

Open the dialog *Administration of knowledge bases* via the main menu *Knowledge base -> Administrate*. Select the knowledge base *TrainingExample* in the left table and click in the right part on the icon to freeze the version 1.0 of *TrainingExample*.



Due to the so called CodeFreeze classes which were already reserved can still be modified, but released classes can not be reserved. That means the developers can complete their work but they can not begin additional changes.

Of course, it is possible to unfreeze a frozen knowledge base. For this the icon is used.

After all changes on reserved classes were applied by releasing the ticket and the knowledge base was frozen, the release version can be generated.

Open the dialog *Administration of knowledge bases* from the main menu *Knowledge base*. Select the knowledge base *TrainingExample* in the left table and generate a Release version out of the selected frozen Working version by clicking the icon .

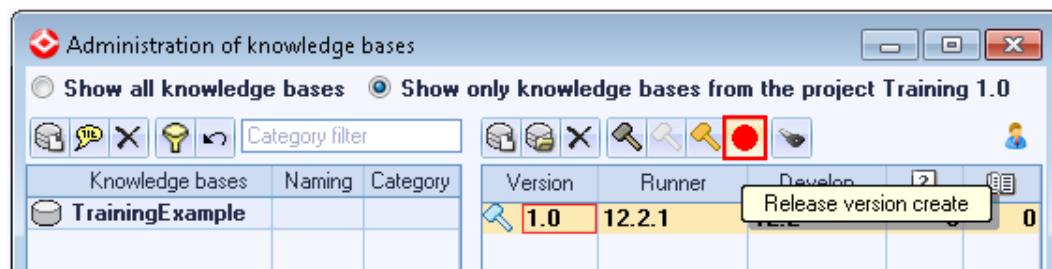


Figure 54: Generating a new release version

You will notice that the knowledge base was closed afterwards. The reason is that it does not exist anymore! With the release procedure the work version 1.0 was changed to a release version 1.0. You could now open this version, but in the release version you have no longer the possibility to reserve classes.

In order to continue developing, after the release process a new work version has to be created from the release version.

22.4. Practice: Creating a new working version



Open the dialog *Administration of knowledge bases*. Activate the option *Show all knowledge bases* and select the knowledge base *TrainingExample* in the left and the *Release version 1.0* in the right table.

Click on the icon in order to create a new work version from the release version.

In the appearing dialog you can decide with which main- and secondary version number the new version has to be created.

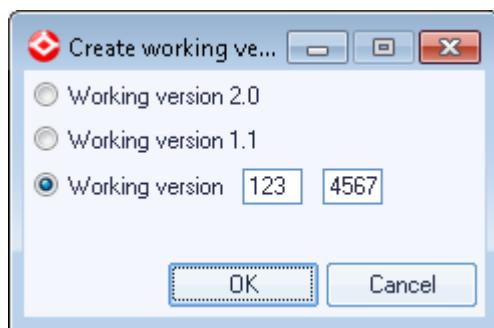


Figure 55: Generating a new working version

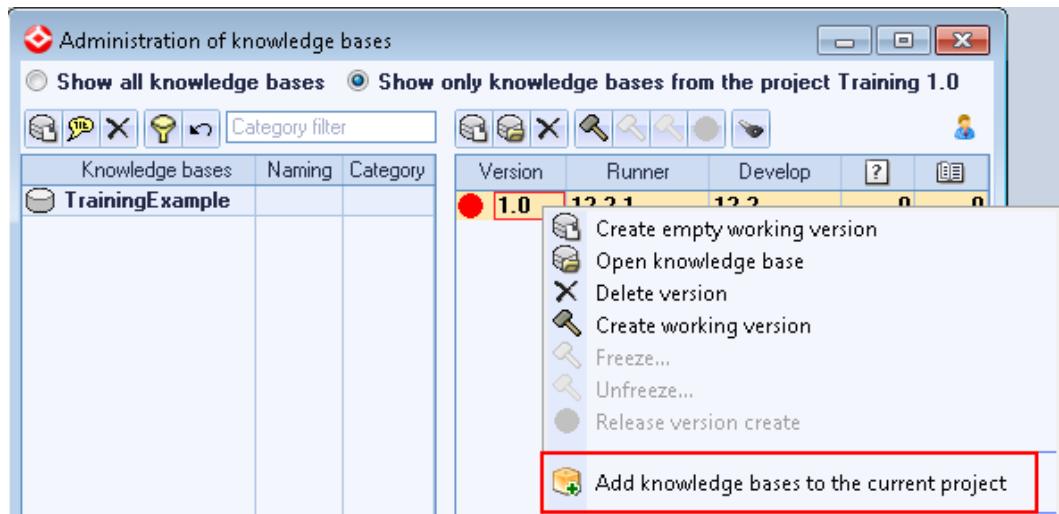
By default a main version jump, e.g. from version 2.6 to 3.0, a so-called major release, is only carried out after extensive and partially not compatible adjustments on the knowledge base. For versions that contain only error recoveries etc., only the secondary version number is increased, e.g. 2.6 to 2.7. You can also specify the version as you like, with this the main version number can be 3-digit and the secondary version number up to 4-digit.

For the knowledge base the numbering is unimportant, you just have to keep track which version is the most current one.



Activate the radiobutton *Working version 1.1* and confirm the creation with *OK*.

With the creation of the working version the selected knowledge base Release version 1.0 was completely copied. To continue the developing the newly created version 1.1 has to be added to the project.



Select the working version 1.1 in the right table and add the knowledge base to the current project via the corresponding context menu item.



Open the knowledge base TrainingExample 1.1 via double click or the context menu Open knowledge base.

In the header and on the bottom of the class tree you can see the new version number 1.1 now.

TrainingExample 1.1

Currently there is no active ticket and all the classes are released. This means, that if you want to change the classes or the knowledge base properties in the following chapters, you have to create a new ticket, set it as the active ticket and reserve the classes via context menu.



Open the ticket system via the icon  and create a new ticket via the toolbar icon New. Define a subject, e.g. "Further development knowledge base" and confirm the dialog with OK. Afterwards activate the ticket by clicking on  Actual ticket set/reset.

To be able to enhance the frame aswell, a new working version of the frame is created.

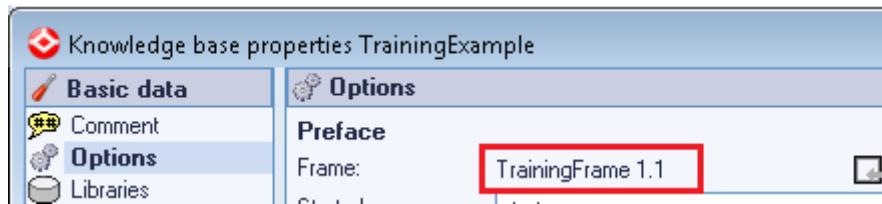


Open the Frame maintenance, select the release version 1.0 of the TrainingFrame, click on the icon  and create a working version 1.1 of the frame.

In order that the knowledge base uses this frame, this has to be defined in the knowledge base properties.



Open the Knowledge base properties of the knowledge base TrainingExample 1.1 and reserve them. Change to the area Basic data -> Options and allocate the TrainingFrame 1.1 via the icon .



22.5. Repetition

- The creation of working- and release versions is carried out in the dialog *Knowledge base -> Administrate*.
- A release version of a knowledge base has to use a release version of a frame.
- Before creating a release version of a knowledge base, the working version has to be frozen.
- All classes have to be released ahead of the creation of a new working- or release version.
- Release versions are created to freeze a certain development state. They cannot be edited later.
- In order to extend the state of a release version, a new work version has to be created from this version.
- The allocation of the main- and secondary version number is up to you.

23. Model-specific components

23.1. Intention

Like the convertible models which are the only ones with soft tops available, there are accessories that can only be selected for the model Passat. The components roof rail, transport box and bicycle carrier should be only selectable for the model Passat.

In this chapter you repeat the creation of classes, list components, values and rules. With the representation of the configurationbox component for the transport accessories you will learn how to access components of different classes.

23.2. Practice: Create model-specific components

Extend the class structure of the modules by the base class "TransportAccessories" with the object classes "RoofRail", "TransportBox" and "BicycleCarrier".



Maintain the prices and the namings of the new components, if necessary.

Since the transport accessories can only be configured for the model Passat, the component is not created in Car but in the class Passat. In addition, a multiple selection should be allowed for the transport accessories.

Create a list component of the TransportAccessories in the class Passat. Add all values to the structure tree by double clicking them.

The selected transport accessories should also be displayed in the quotation.

For this reason you set the option *Output to result/printout form* on the component `_TransportAccessories[]`.

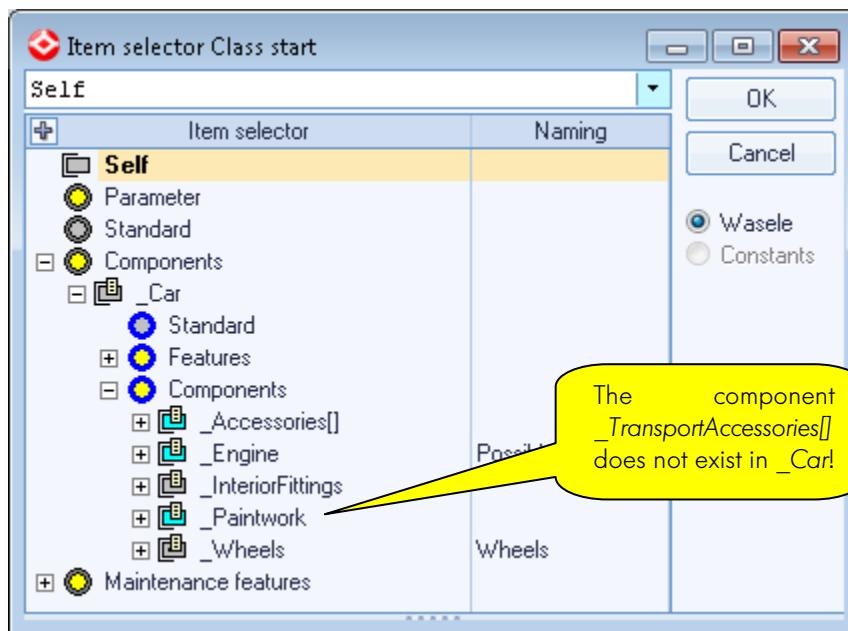
In order to display the transport accessories on the form, a configbox component is created.

Create a *Configurationbox component* on the MainForm and place it at suitable position on the form preview.

Open the item selector dialog in the field *Cause variable* and insert the cause variable `_TransportAccessories[]`.



If you have problems with the last step, please consider the following: Since the class start has a component on Car and the transport accessories are only existing in the class Passat, the component `_TransportAccessories[]` is not offered for selection here.



How can the component still be displayed on the form?

The creation of a component `_Passat` in addition to the component `_Car` is for reasons of redundancy not recommendable. It would also be wrong to shift the transport accessories in the class `Car`. The solution is a **typecast** which you already learned to know in chapter 20.4.2.1.

A typecast is separated by colons and periods, specified in the syntax `<_Component>:<Child class>.<wasele>`.



Note: If you cast on a class / component etc. that contains blanks or numbers, the corresponding wasele have to be enclosed in inverted commas, for example `_Car:"Model Passat"._TransportAccessories[]`.

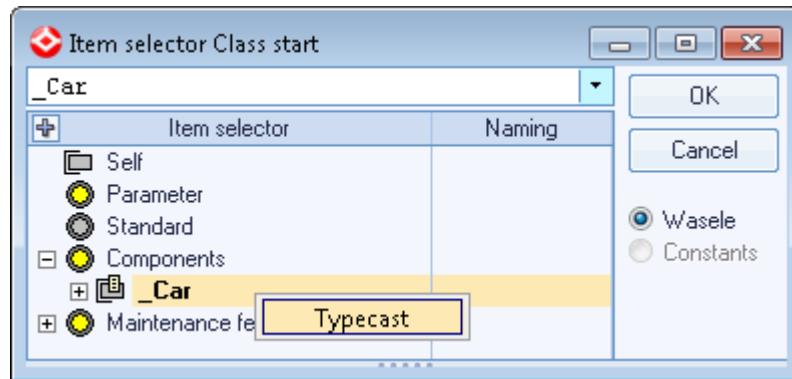
In our case the component `_Car` should be cast to the child class `Passat`. The target, the wasele/component `TransportAccessories[]` is located in the class `Passat`. The complete typecast is therefore `_Car:Passat._TransportAccessories[]`.

FAQ/Hints and questions concerning the interpreter/Access to constants of not-instanced components

The typecast can be carried out via free entry, but it has to be considered that the syntax check does not recognize spelling errors in a typecast that could lead to runtime errors. The safer way is the typecast via the item selector dialog.



Open the item selector dialog in the line cause variable. Via a right click on the component that has to be cast, the context menu with the menu item "Typecast" opens. Click on *Typecast*.



Select the class *Passat* in the dialog *Choose Typecast class*.

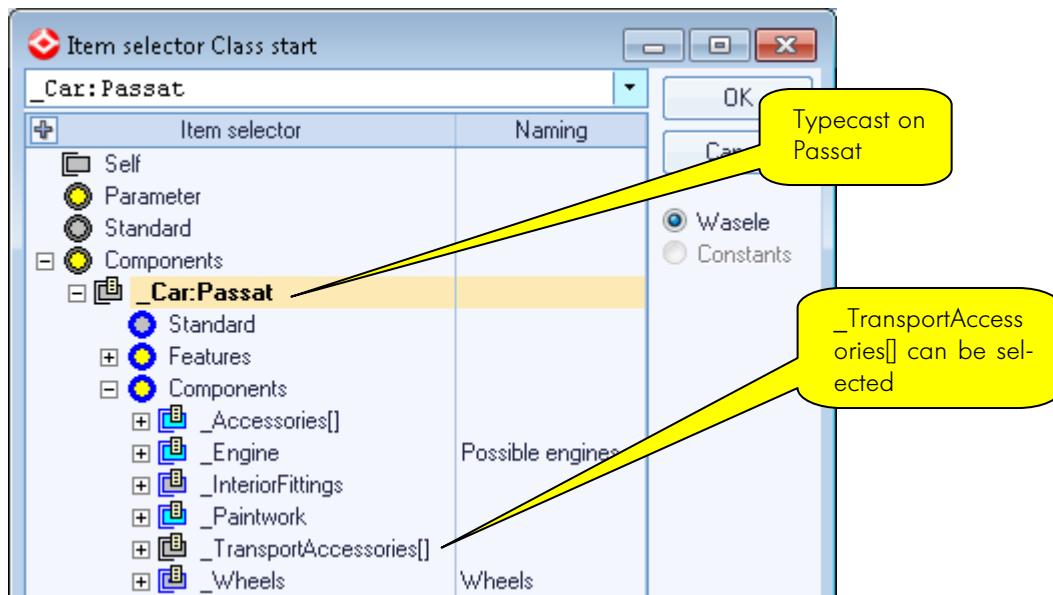


The dialog closes independently after the selection.



Figure 56: Typecast in the item selector

Via the typecast on *_Car* the component *_TransportAccessories[]* that is locally defined in the class *Passat* is shown in the item selector dialog.



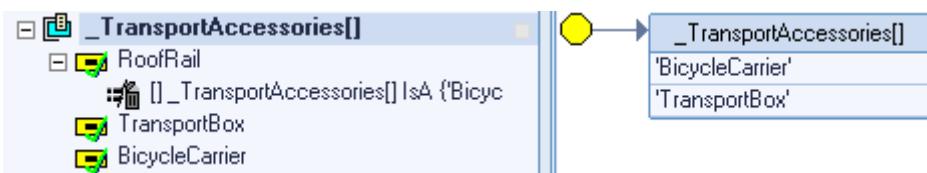


Select `_TransportAccessories[]` and confirm the selection dialog with `OK`.

There should be rules for the component `_TransportAccessories[]`. If the `BicycleCarrier` or the `TransportBox` is selected, the `RoofRail` should be automatically assigned.



Phrase the rule on the value `RoofRail` of the component `_TransportAccessories[]`.



Be sure, that the value `NOVALUE` has been assigned to the structure tree. If not, the user is not allowed to deselect the transport accessories, when the rule flag is set.



Additionally, create a price column, in which you display the *feature* price of the transport accessories parts. Start the debugger and test your application.

Beetle or Golf selected

TransportAccessories	Price

Passat selected

TransportAccessories	Price
<input checked="" type="checkbox"/> <input type="checkbox"/> Roof rail	\$320.00
<input checked="" type="checkbox"/> <input type="checkbox"/> Transport box	\$480.00
<input checked="" type="checkbox"/> <input type="checkbox"/> Bicycle carrier	\$250.00

If the model Passat is selected, the configbox is filled with values; with the selection of a Beetle or Golf it remains blank.

This representation is not advantageous for the models Beetle and Golf, because the meaning of the blank configbox is not clear for the user. So-called subforms can be used to optimize the surface. The use of subforms is described in chapter 24.

23.3. Tips & Tricks

The main advantage of the typecast is that constants for icons, standardized error messages etc. can be swapped in container classes. These container classes are never instantiated as objects, via a typecast however their constants can be addressed.

An example is the graphic constant for the icon New which is used at the most different places of an application. Instead of referring to an existing object (and to create the icon in each of these objects separately), the icon is addressed via a typecast, e.g. :IconContainer.!New

Additionally there is a reduced maintenance- and administration expenditure if many icons have to be changed at the same time or if they are used at different places.

23.4. Repetition

- Typecasts are used to:
 - address locally defined elements of a class that is not defined as object/component
 - address constants that were swapped in container classes

24. Dynamic form structure

24.1. Introduction

Target of this chapter is to optimize the representation of the configurationbox component for the transport accessories of the Passat. Its content is momentarily only displayed for the Passat, for all other models it exists but is disabled.

An appealing solution would be to fade out the configbox for the models Golf and Beetle and to fade it in again for the Passat. The following chapter explains how this can be realized via using the form element subform.

24.2. What is a subform?

The form element Subform serves as a wildcard for another form. I.e. a place in the width and height of the subform on the form 1 is kept free in order to display form 2 during runtime in it.

It is differentiated between static and dynamic subforms. For a correct display both need the name of the form as well as the object in which this form is located.

Definition

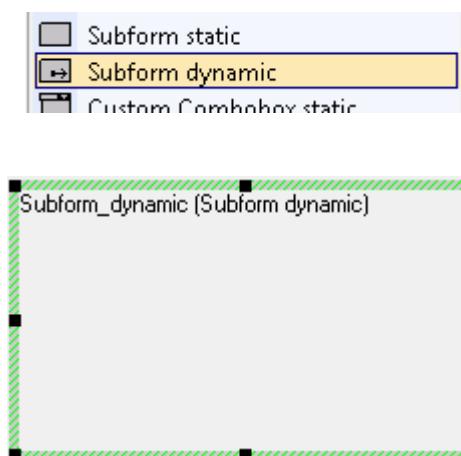
The size of the subform and the size of the form that has to be displayed should principally concur. If this is not the case, you should define how to proceed with the size difference by setting the option *Size handling* in the properties of the subform.



The aim of the following practice is to display only the configbox for the transport accessories if a Passat is selected. For the other two models a label with the inscription “No further accessories available” should be displayed.

24.3. Practice: Using subforms

First you create a *dynamic subform* on the *MainForm*.



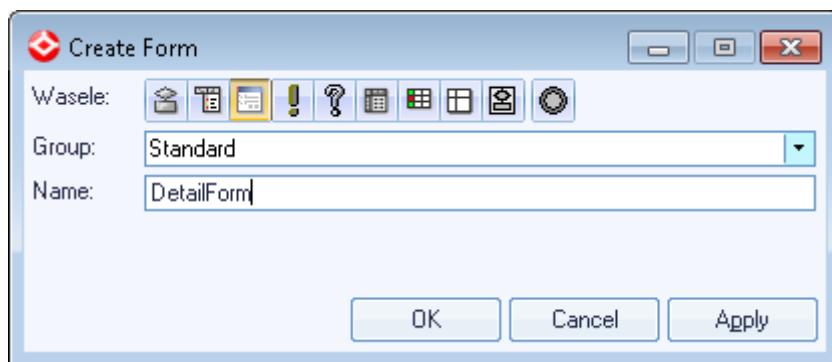


This subform should be somewhat bigger than the configbox for the transport accessories. Adapt the height and width of the subform.

In the area where the dynamic subform is located, the form with the configbox respectively label should be displayed.



To do so, you create a new form with the name "DetailForm" in the class *Car*. Assign the same height and width as of the *dynamic subform* on the *MainForm*.



This form is inherited to all car models.

First we create a label on the form, because this has to be displayed in two classes. In the class *Passat* the *DetailForm* is overloaded and the label is replaced by the configbox component for the transport accessories.



On the *DetailForm* you create a *static label* with the text "No further accessories available." in the class *Car*.

Form preview:



Now the dynamic subform on the *MainForm* has to be instructed which form has to be displayed at its place. To do so, the component of the class is specified in which the *DetailForm* is located. In addition the name of the form that has to be displayed is declared.



Open the *MainForm* and the properties of the *Subform dynamic* in the class *start*. Via the Item selector dialog you select the component *_Car* as the *Form object*. Then you select the *DetailForm* in the field *Form name*.

Name		Subform_dynamic
Type		Subform dynamic
Stylesheet		
+ Geometry		
+ Representation		
+ Handling		
- Specials		
Form object		_Car
Form name		DetailForm

Test your application and the correct appearance of the *DetailForm*.



The *DetailForm* is displayed without header line at the position where the dynamic subform is located. With this there is no recognizable difference between form elements that are directly on the *MainForm* and the ones on the *DetailForm*.

Interpreter:

No further accessories available.

Now we come to the actual purpose of the dynamic subform, the automatic form change during runtime. This is already carried out, but since on each *DetailForm* of the class *Car* or on one of its children the same contents exists, a difference cannot be recognized.

Note: Do carry out all changes on the *DetailForm* now. If the *DetailForm* in the *Passat* is overloaded, changes of the *DetailForm* in *Car* are no longer inherited to *Passat*. Changes that affect all cars would have to be locally reproduced in the *DetailForm* in *Passat*.



Overload the *DetailForm* in the class *Passat* and remove the static label.

Cut out the configbox component for the *TransportAccessories* in the *MainForm* and insert it on the *DetailForm*. Adjust the position of the configbox.



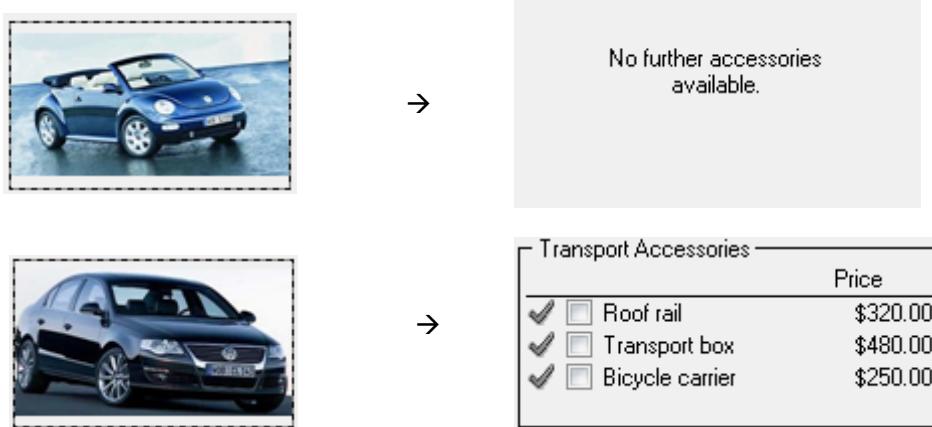
The configbox is not correctly displayed in the form preview. Why?

Due to the cutting and inserting the path of the cause variable is no longer valid. From the class *start* the class *Passat* was accessed via a typecast, but now we are in *Passat* itself. The typecast is therefore no longer correct / necessary.

Select the cause variable *_TransportAccessories[]* via the Item selector dialog.

Test the model change in your application.





24.4. Tips & Tricks

In addition to dynamic subforms there is a second type of subforms: the static subforms. The difference is that dynamic subforms automatically search in the assigned object for the form that is deposited in the form editor and display it.

In dynamic subforms a form object and the form name is deposited, in static subforms only the form name.

Dynamic subform:

Specials	
Form object	_Car
Form name	DetailForm

Static subform:

Specials	
Form name	DetailForm

In static subforms the form object has to be set manually via the function `WinSetObject(Form handle, Form element, Component)`. The local class (= Self), a component or an object pointer has to be entered as parameter `Component`.

Workbench/wasele/Form/Form elements/Subform ...

Function reference/Dialog Functions/`WinSetObject(Form handle, Form element, Component)`

24.5. Repetition

- The form element subform is a wildcard in which other forms can be displayed.
- The size of the subform should correspond to the size of the form that has to be displayed.
- After the overload of a wasele in the child class, changes of the wasele in the parent class do no longer affect the child wasele. This should be considered before an overload.
- Dynamic subforms display the object deposited form.
- On static subforms the form object has to be set via the function *WinSetObject()*.

25. Discount calculation via 2D-table

25.1. Intention

With an immediate cash payment of the ordered model a certain cash discount (in addition to a possibly regular discount) is granted to the purchaser. This amount is calculated on the basis of a by the manufacturer determined value table that considers the selected model and the current end price.

You will learn how to deposit and use value tables in camos Develop. The calculated cash discount reduction has to be displayed in the result.

25.2. The wasele 2D-table

In camos Develop the wasele 2D-table is used to file value tables with two input values and one output value.

A **2D-table** represents a table how it is known from technical manuals. A value is determined on the basis of two input values. The 2D-table is structured in form of a matrix. The column heads contain the value ranges for the first cause variable and the line heads the value ranges for the second one. In the fields inside the matrix are expressions whose values are returned by the 2D-table if the corresponding column and line was determined. It is possible to import data (e.g. from a CSV-file) into a 2D-table.

2D-tables are administrated in the wasele list. They are displayed with the icon  and like methods they are called with two transfer parameters:

```
Return value := Table name(X-value, Y-value);
```

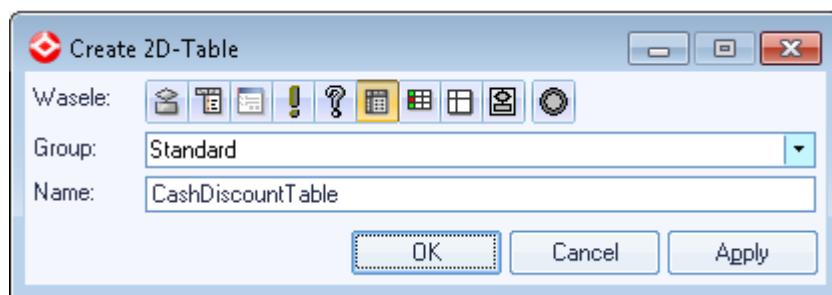
 Workbench/wasele/2D-table

Definition

25.3. Practice: 2D-table for cash discount calculation

The cash discount deduction depends on the amount of the end price and the model of the car. Since the feature *End price* was deposited in *Car* and the models of *Car* are derived, it is useful to deposit the 2D-table in *Car* too.

Create a 2D-table in the wasele list of the class *Car*. Allocate the name “CashDiscountTable”.



The data type of the X- and Y-axis and the data type of the return value is specified in the line above the table. The definition is carried out via selection from the context menu. The value on the X-axis represents the model name and is transferred as string. The Y-value checks for the end price (a currency feature) therefore the entry type is also a currency.

X-value type **S** ▾ Y-value type **€** ▾ Return type **1** ▾

In the table cells the numerical return values are entered that have to be returned with the combination of the X- and Y-value. From this 2D-table we expect the return of the cash discount factor that is deducted as percentage from the end price later. Therefore the return value is numerical.



Fill the table according to the following schema. Consider the specification of the type of the entry values and the return value.

X-value type **S** ▾ Y-value type **€** ▾ Return type **1** ▾ Default: **0**

		X1	X2	X3
		'Beetle'	'Golf'	'Passat'
Y1 < 17500	1.2	1.5	2	
	2	2.25	2.75	
	3	4	5	

Figure 57: Structure of a 2D-table

Example: If the values "Beetle" and 23500 € are transferred to the table, it returns the value 2.

In 2D-tables certain combinations of values can occur that are not covered up by the values in the table. Example: 2 years from now, you will extend the knowledge base by a new car model, but the 2D-table remains unimproved. In order prevent an error with the discount calculation in this case, a default value is entered in the field *Default* that corresponds to an init value. In our case the presetting is 0.



The type of the presetting value has to correspond to the type of the return value.

Through the 2D-table we get the cash discount factor. In the result however the actual value that can be deducted in case of cash payments from the end price has to be displayed. The calculated cash discount amount should be saved in a feature.



Create the currency feature *CashDiscount* in the class *Car*.

Create a method that transfers the *Model* and the *EndPrice* to the 2D-table and that contains the *Factor*. From this factor the cash discount amount is then calculated in the method.

Create a new method with the name “CalculateCashDiscount”. As a local, numerical variable *Factor* is defined.

Enter the following procedure code for the method:

```
Factor := CashDiscountTable(Self, EndPrice);  
CashDiscount := Factor * (EndPrice / 100);
```



The cash discount table is called with *Self* and the *EndPrice*. However, in the 2D-table is defined that a string and a currency have to be transferred. The model (= the name) is transferred as string, because 2D-tables cannot process object pointers. The value of *Self* is automatically converted to the desired data type.

Since the feature *Cash discount* is displayed in the result, the method *CalculateCashDiscount()* has to be called before the result is generated. As known, the result is called via the menu. This is the right place for adjustments.

Add the call of the method *CalculateCashDiscount()* **before** the result is opened. To do so, you open the procedure editor of the menu item *Create quotation* and add the following source code (the comment lines do not have to be applied):

```
# Calculate cash discount  
_Car.CalculateCashDiscount();  
# Call result  
WinStartModal(WinOpenDoc("Quotation", 0, 0, 800, 600));
```



On the quotation, the customer should see what reduction he was granted; therefore the text module has to be extended.

To do so, open the textelement which is deposited in the constant *!ResultHeader* and insert the cause variable *{CashDiscount}* with a corresponding accompanying text.



Test the cash discount calculation in your application.



25.4. Tips & Tricks

Since the feature *CashDiscount* is exclusively used as a cause variable in the result and has otherwise no use, the call of the method *CalculateCashDiscount()* can be directly integrated as cause variable in the result. In this case the function call of *CalculateCashDiscount()* would be automatically carried out with the result generation and the calculated cash discount amount would be directly entered in the result. So the feature *CashDiscount* as well as the function call in the menu item would no longer be necessary and could be removed.

The first step is the replacement of the cause variable *{CashDiscount}* by the method call *CalculateCashDiscount()*.



Open the textelement of the constant *!ResultHeader* in Car and delete the cause variable *CashDiscount*. Then you select via the icon the method *CalculateCashDiscount()* or you enter the method call manually in the upper field.

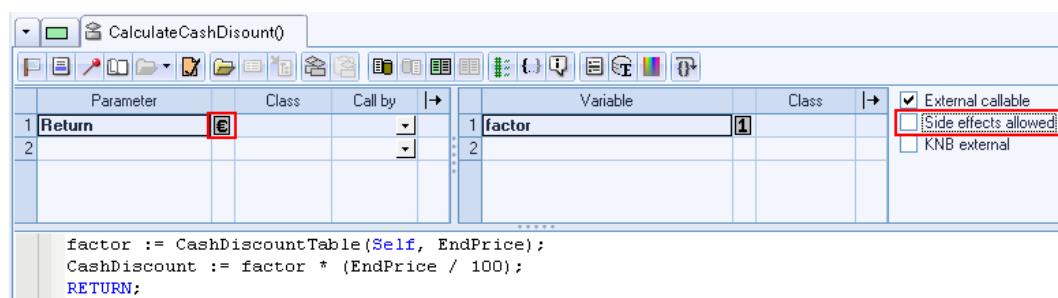
Cash discount: *{CalculateCashDiscount()}*

If the function call has to be carried out from within the result, the method *CalculateCashDiscount()* has to be adjusted for a proper functioning. As you know from chapter 0, only methods

without side effects are allowed in results. For this reason the option *Allow side effects* has to be disabled for the method *CalculateCashDiscount()*.

The calculated cash discount value is now no longer filed in a feature which is then used as cause variable in the result. The result of the calculation is directly displayed in the result. Therefore the value that is calculated via the function has to be returned by the method as return value.

Then also the return value changes which is no longer numerical, but of the type currency.



```

    factor := CashDiscountTable(Self, EndPrice);
    CashDiscount := factor * (EndPrice / 100);
    RETURN;

```

Tip: Even more compressed would be the call:

```
RETURN CashDiscountTable(Self, EndPrice) * (EndPrice / 100);
```

With this the variable *Factor* would be unnecessary.



When you use local variables of the data type numerical, date or currency you must consider, that these are automatically filled with a value at runtime.

- * Numerical variable with the value 0
- * Date variables with the current date
- * Currency variables with the value 0 at the main currency



It is not possible to delete this value. When NOVALUE is assigned to a numerical, date- or currency variable the autovalue will be automatically reassigned.

If the display of the discounts is tested successfully, the call of *CalculateDiscount()* can be removed from the menu and the feature *CashDiscount* can be deleted.



25.5. Repetition

- 2D-tables serve as to determine on the basis of two cause variables a value that is imaged in a matrix.
- 2D-tables are called with two transfer parameters (comparable to methods) → 2D-Table name(X-value, Y-value)
- The value of the field *Default* is returned if a transferred combination is not covered by the 2D-table.
- The column heads of the 2D-table contain the value ranges for the first cause variable, the line heads the value ranges for the second cause variable.
- The matching type has to be set for the transfer values and the return value.

26. Integration of car keys

26.1. Intention

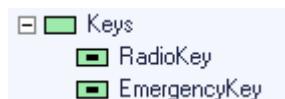
The car configuration has to be extended by the selection of radio- and emergency keys. It should be possible to determine a certain number for both key types. Now the previously used selection that selects (generates) or deselects (deletes) a certain object has to be extended by a quantity specification.

In the following chapter you will learn how camos Develop deals with quantities. In this course you repeat the creation of classes, list components, configuration boxes for components and the depositing of units of quantity in the frame. You will also learn about quantity rules and their use.

26.2. Practice: Extend the class structure by car keys

Keys belong to the group of components. Extend the existing class structure by one base- and two object classes.

To do so, you create the base class "Keys" under *Modules*. To this base class you create the object classes "RadioKey" and "EmergencyKey".



In the class *Car* you create a list component of *Keys* and assign the possible values.



By default one key per type should be selected for each car.

Therefore you initialize the component *_Keys[]* with *RadioKey* and *EmergencyKey* in dragging and dropping the *Values* in the field *Init value*.



Allocate prices for both key types (\$60 for radio key and \$20 for emergency key) and maintain the namings.



In order to take up the keys in the result later, the flag *Output to result/printout form* has to be activated on the component. Additionally the option *Sort* can be selected to determine the order in the structure tree and in the result.

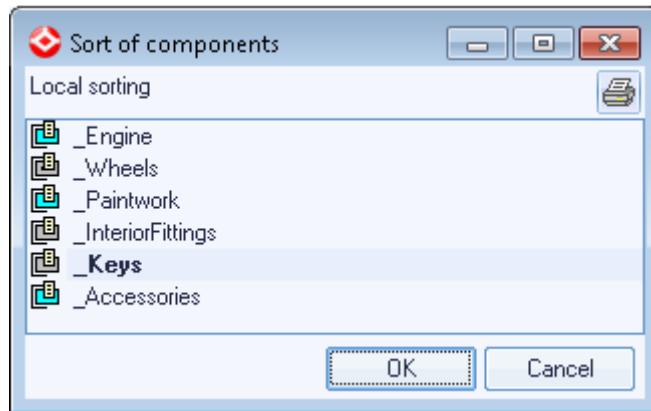


Set the options *Output to result/printout form* and *Sort* in the editor of the list component *_Keys[]*.

The *component sorting* is carried out in the editor of the class *Car*. As already explained in chapter 13.4, the dialog for the component sorting is opened via the icon . There the existing components can be arranged in the desired order via D&D.



Arrange the component *_Keys* in a useful order.



Now the component *_Keys[]* should be displayed on the form via a configuration box.



On the *MainForm* you create a configbox component with the cause variable *_Keys[]* and adjust its size and position correspondingly. Additionally you create a configurationbox column for the price.

Test your application.

Configbox component with cause variable *_Keys[]* in the interpreter

Keys		Price	
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Radio key	\$60.00
<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Emergency key	\$20.00

Momentarily there is no possibility to affect the number of the respective key type. In order to fulfill this requirement, quantities are introduced in the next step.

26.3. What are quantities?

Quantities are always needed if identical objects have to exist several times in a configuration.

Internally camos Develop administrates a quantity specification for each component. If the switchbox *Quantity* on the respective component is disabled, this quantity is during runtime

0, if the object does not exist or it is NOVALUE

1, for the case that the object exists

By activating the option *Quantity* the quantity of a component can take any other numerical values during runtime. With this the *Unit of quantity* of the component class plays an important role. It specifies the *Unit of quantity* to which the quantity refers.

Quantities can (must not) be provided with units of quantity. The allocation of a unit of quantity is carried out on the class of the desired component (= component class) from which the object is created.

With changes of the quantity of an object the system method *ChangeQuantity()* (see chapter 26.7) is running.

26.3.1. Why no feature “Quantity” or no list of keys?

Since the quantity of an object is saved and administrated on the corresponding component itself, the question is unnecessary why the number is not realized via a numerical feature or a list in the key class.

- a) On the one hand this feature would have to be explicitly addressed with each quantity change. With the maintenance via the component the system method *ChangeQuantity()* independently registers the quantity change. This solution is less large-scale, has a lower error potential and uses less storage space.
- b) On the other hand is the number NO property of key.

26.4. Practice: Configure the number of keys

In order to change the number of keys, the option *Quantity* has to be set on the component (see definition in chapter 26.3). With this the table *Init value* is extended by a column for the *Quantity*.

The screenshot shows the Component editor interface for the 'Keys' component. On the left, there's a tree view with 'Keys' expanded, showing 'RadioKey' and 'EmergencyKey'. On the right, there's a configuration panel with several checkboxes: 'Take childclass' (unchecked), 'Output to result/printout form' (checked), 'Sort' (checked), 'Quantity' (checked), 'Help' (unchecked), and 'List' (checked). Below these checkboxes is a table titled 'Initialization' with two rows. The first row contains 'Quantity' (1) and 'Initialization' (RadioKey). The second row contains 'Quantity' (1) and 'Initialization' (EmergencyKey). The 'Quantity' column is highlighted with a red border.

Quantity	Initialization
1	RadioKey
1	EmergencyKey

Figure 58: Component editor – Specifying a quantity

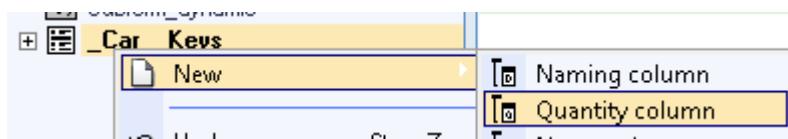


To do so, you open the class Car and set the flag *Quantity* in the editor of the list component *_Keys[]*.

Via activating the *Quantity*, values can be initialized higher 1 in the column *Quantity*. Now it is also possible to extend the configurationbox component by a column which makes the quantity on the form configurable.



Create a quantity column on the configbox with the cause variable *_Keys[]*.



If necessary, change the horizontal alignment of the quantity column to *Right*.



In the interpreter quantity columns are displayed with a white shade in order to indicate the edit possibility. If the focus is in a line of the quantity column, spinbuttons are displayed with which the quantity can be reduced or enlarged. An entry of numerical values via the keyboard is also possible.

Test the quantity column in your application.

Keys		
	Price	Quantity
✓ Radio key	\$60.00	1
✓ Emergency key	\$20.00	1

If the quantity is set to 0, the object is deleted.

Keys		
	Price	Quantity
✓ Radio key	\$60.00	0
✓ Emergency key	\$20.00	1

To change the quantity of an object you can easily use the quantity column. But the user doesn't know, of what unit the quantity is: 5 pieces, 5 dozens or 5 cartons?

26.5. Practice: Define and assign the unit of quantity

The keys are neither specified in horse power, kilowatt or in per cent. Therefore we cannot use one of the previously used units of quantity. For this reason the new unit of quantity "Piece" is created in the frame.

If you don't exactly remember the procedure how to create a unit of quantity, you can read the individual steps again in chapter 15 and 19.

Open the *TrainingFrame*, select the area *Units of Quantity* and reserve it.

Create a new *Unit of quantity* group with the name "Amount". The Master UOQ is "Piece", the naming of the unit is shortened with "pcs.". Change the English naming to "pcs.", the German naming to "Stk." and keep the other system initializations.

Release and close the Frame.



Name	Factor	Offset	Unit of quantity
Piece	1	0	pcs. Stk.

In the class *Keys* you assign the unit of quantity in which the component has to be displayed on the form.

To do so, open the class *Keys* and select the UOQ *Piece* (right-click in column *Unit of quantity*) in the class editor.



Figure 59: Class editor - Assigning a UOQ for components

Confirm the dialog *Choose unit of quantity* with *OK*.





Test your application, especially the output of the unit of quantity.

Keys		
	Price	Quantity
✓ Radio key	\$60.00	12345 pcs.
✓ Emergency key	\$20.00	2000 pcs.

As you can see, the entry in the quantity columns is limited to eight places. A configuration with this quantity of keys (maximum 99999999 pieces) is unthinkable in reality. Therefore the entry should be reasonably limited.

26.6. Practice: Create quantity rules

If the option *Quantity* is set on a component, the rule editor changes additionally.

You already know that the *Rule group* can be entered in the header of the rule editor of values of scalar components. In components with quantity specification there is additionally the field *Quantity*. Via this field quantities can be included in conditions.

Rule group: Index: Quantity: > 0

Therefore quantity rules consist of 2 conditions:

- 1.) an expression / a list
- 2.) a quantity to which the rule refers

May and MayNot rules are only valid for the specified quantity. E.g. a MayNot rule with quantity > 1 and expression 1. This rule forbids all quantities higher 1. 1 is however allowed.

Assign-rules and AssignDelete-rules assign the quantity that is specified in the field *Quantity* to the object.

In order to use realistic conditions for the configuration, the keys are limited with quantity rules to certain numbers of pieces. The following rules should be valid:

- Generally not more than 5 radio keys are allowed
- The allowed quantity of emergency keys varies for each model:
 - Beetle max. 3 emergency keys
 - Golf and Passat max. 4 emergency keys



Try to realize the given rules with your present knowledge by yourself.

You should have proceeded as follows:

- Setting the option *Rules enabled* on the component *_Keys[]* in the class *Car*.
- Creating a May Not-rule on the value *RadioKey*. In the field *Quantity* > 5 is entered. The condition is 1, because this rule is always (and for each model) valid.



Figure 60: Rule editor of a quantity rule

- Create a May Not-rule on the value *EmergencyKey*. In the field *Quantity* > 3 is entered. As condition a list with the cause variable *Self* (i.e. in the object *Car*) and the value 'Beetle' is entered.
- Create a further May Not-rule on the value *EmergencyKey*. As quantity constraint > 4 is entered. As condition a list with the cause variable *Self* and the values 'Golf' and 'Passat' is entered.



After the creation of the rules you check their keeping in the configuration.



If the ruled maximum limit of the number of keys is exceeded, the validity symbol changes from to , in order to display the violation of a rule (example Passat):

Maximum number of keys exceeded:

Keys	Price	Quantity
Radio key	\$60.00	6 pcs.
Emergency key	\$20.00	5 pcs.

Maximum number of keys kept:

Keys	Price	Quantity
✓ Radio key	\$60.00	4 pcs.
✓ Emergency key	\$20.00	3 pcs.

26.7. The system method ChangeQuantity

With a change of the quantity of a key the correct value is displayed in the quantity column. The list price however is only updated with a change from 0 to 1 (and vice versa). Why that?

The method *New* of an object is running when the object is created. The method *Delete* is always carried out with a deletion. None of these methods is capable to register a quantity change of the keys. After the creation, the quantity is changed internally and the new method is not called again. It is the same with the reduction of the quantity.

Since the calculation of the list prices is carried out in these methods, we are forced to adapt the price calculation for the keys and to write an additional method.

To do so, we will use the system method *ChangeQuantity()* and the function *GetQuantity*:

ⓘ Workbench/wasle/Method/Special Methods/Method *ChangeQuantity*

ⓘ Function reference/Other Functions/*GetQuantity* (*Component*[, *Class*])

26.7.1. Practice: Adapt the price calculation for the keys

In order to consider the configured quantity of the keys in the price calculation, the methods *new* and *delete* are adapted. The method *GetQuantity()* provides the current quantity of the transferred object.

Overload the method *New()* in the class *Keys* and change the source code as follows:

```
@Car.PriceAddition(GetQuantity(Self) * Price);
```

Also overload the method *Delete()* and change the method contents as follows:

```
@Car.PriceSubtraction(GetQuantity(Self) * Price);
```



Via adapting these methods the set quantity is considered with the updating of the price after selection and deselection of a key.

Example for *Delete()*: A key object has the quantity 4, the object is deleted by removing the selection. The selected quantity (4) is included in the reduction of the price.

Example for *New()*: With the creation of a key object with an init value higher 1, the list price is correspondingly adapted.

At the moment the price calculation with changing the quantity, e.g. if you increase the quantity on the form from 2 to 3 pieces, is missing. As already mentioned, in this case the system method *ChangeQuantity()* is used. This method is automatically called with changes of the

quantity. To do so, instructions for the price update are entered in the method *ChangeQuantity()*.

The method *ChangeQuantity()* has the internal parameter *NewQuantity* that contains the new quantity; via *GetQuantity(Self)* the previous quantity is determined.

Create a method with the name “*ChangeQuantity*” in the class *Keys*. Enter the following commands in the procedure editor:

```
# Subtract current quantity*price
@Car.PriceSubtraction(GetQuantity(Self) * Price);

# Add new quantity*price
@Car.PriceAddition(NewQuantity * Price);
```

Start your application and test the price update with a change of the quantity.



26.8. Tips & Tricks

26.8.1. Permanent display of spinbuttons

The spinbuttons are (in the default setting) always displayed if the focus is in the corresponding line of the quantity column. In order to fade in the spinbuttons permanently, you can set the option *Spinbuttons always active* in the form editor of the quantity column.

Editor of the quantity column:

AdjustColumnWidth	<input type="checkbox"/>
Spinbuttons always active	<input checked="" type="checkbox"/>

Interpreter:

Keys	Price	Quantity		
✓ Radio key	\$60.00	3 pcs.	<input type="button" value="−"/>	<input type="button" value="+"/>
✓ Emergency key	\$20.00	2 pcs.	<input type="button" value="−"/>	<input type="button" value="+"/>

In the interpreter the spinbuttons are displayed without having a focus in the line.

26.8.2. Display the number of keys and their price in the quotation

The number of the keys is actually added to the total price of the car, but neither the quantity nor the summed-up price of the keys is displayed in the result. Therefore the quantity, the unit of quantity and the total price of the keys should now be integrated in the result.

Overload the constant *!ResultTable* in the class *Keys*. Open the textelement in the textmanager and copy it. Change the copy in the following steps and assign the copy to the constant afterwards.



As you already know, via the function `GetQuantity(Self)` you get the currently selected quantity of the transferred object. Via the function `GetUOQ(Self)` it is possible to query the unit of quantity (= Unit Of Quantity) of an object via function. These functions can therefore be used as cause variables in the result.

 Insert the following cause variables (via the item selector dialog) in the first column:

{GetNaming()} {GetQuantity(Self)} {GetUOQ(Self)}

If necessary, put brackets around the quantity and quantity specification (see above).

In the second column momentarily still the single price of a key is displayed. As in the price calculation of the method `ChangeQuantity()`, this price has to be multiplied with the current number of selected keys.

 Replace the cause variable `{Price}` by the expression `{GetQuantity(Self)*Price}`.

Close the textmanager.

View in the text module:

{getNaming()} ({getQuantity(Self)})	{getQuantity(Self)*Price}
{getUOQ(Self)}	

View in the generated result:

Description	Price
Radio key (2 pcs.)	\$120.00
Emergency key (3 pcs.)	\$60.00

 [Function reference/Other Functions/GetUOQ \(\[wasele\]\)](#)

26.8.3. How to forbid the quantity 0?

Forbidding the quantity 0 means that the deleting of the object is forbidden. In camos Develop this is not possible, because when the quantity 0 is selected, the object does not exist anymore and so no rule could be processed at all.

Solution: React with an automatic new creation to the deletion.

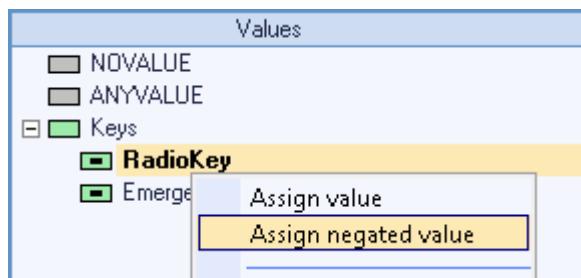
In order to prevent that the quantity 0 is set from the beginning, a new rule is created. This rule has to create an emergency- and a radio key as soon as the respective key is no longer in the list of `_Keys[]`.

For this you can create an *Assign-rule* on the value *Emergency key*. As soon as the emergency key is no longer existing in the list of the keys (*_Keys[]*), this rule should assign / generate a new emergency key. Proceed the same way with the radio keys.

To do so, you first create an Assign-rule on the value *RadioKey*. Drag the list component *_Keys[]* from the structure tree onto the yellow rule-root in the condition editor. By doing this, a condition element list with the cause variable *_Keys[]* is created.



Make a right-click on the value *RadioKey* in the value list on the right side of the workbench. Then select the context menu *Assign negated value*.



That creates the rule condition *_Keys[] NotIsA ,RadioKey*'. The assignment will be executed, when the object in the list *_Keys[]* is not of the type *RadioKey*.

In the field quantity you can define the number of radiokeys that are assigned by this rule. We do not change the default value 1.



Create this rule analog also under the value *EmergencyKey* with the condition '*EmergencyKey*' *Notin _Keys[]*. Start the application and test the behaviour when you change the amount of the keys to the number of 0.



In the application the quantity from radio- and emergency keys is always reset to 1 when setting it to 0. So it is no longer possible to delete the keys.

The operator *NotIsA* returned a 1, when the tested object was not instantiated from the indicated or from a derived class, c.f. chapter 20.4.2.2.

26.9. Repetition

- Internally camos Develop administrates a quantity specification for each object. Quantity 0 means that the object does not exist (NOVALUE). Quantity 1 means that the object exists.

- Via the option *Quantity* on a component the object can be allocated with a quantity higher 1.
- Quantities are used to create a certain quantity of identical objects.
- For values of components with the flag *Quantity* quantity rules can be defined, i.e. the quantity of the object that has to be ruled is considered in the rule condition.
- Via the method *GetQuantity()* the quantity of an object can be determined.
- The system method *ChangeQuantity()* is automatically called with each quantity change.
- The method *ChangeQuantity()* has the internal parameter *NewQuantity*, which contains the new value of the quantity.
- In order to provide components / objects with a unit of quantity, the unit of quantity is assigned in the class editor to the corresponding class.
- In order to change the quantity of an object (on the form), quantity columns are used.

27. Assign components rule-controlled

27.1. Intention

In the car configurator it should be possible to automatically select or deselect several equipment variants for the Golf via the selection of a special model.

A solution via implicit rules on the values is actually possible, but relatively large-scale. Additionally the rules that have to fulfill the same purpose are on the most different places in the knowledge base which makes the maintenance in case of changes more difficult.

Therefore these assignments or deletions should not be realized via individual rules, but via a so-called *Decision table*.

27.2. Definition of decision tables

Decision tables serve as to conveniently enter and display assignment- or assignment/delete rules. Due to one or several conditions defined assignments are triggered.

Similar to the constraints that are a tabular combination of Allowed- and Forbidden (= May and MayNot) rules, decision tables are therefore a tabular combination of assignment- and assignment/delete rules.

Definition

The cause variables that have to be set via a decision table must also have the property *Rules enabled*. If this property is missing in a cause variable, the syntax check in the decision table displays an error.

Decision tables are created in the wasele list of a class. The icon of a decision table is .

27.3. The decision table editor

With a new created decision table you get a blank table with four columns and the two lines B1 and A1. Lines whose description begins with B define **condition part**; lines whose description begins with A are the **action part**. Columns that are described with R are the **rule columns**.

Via the context menu further condition- and action lines as well as new rule columns can be inserted.

Cond./Act.		R1	R2	R3	R4
B 1	Width	< 50	< 50	≥ 50	≥ 50
B 2	Length	[10 .. 150]	> 150	[10 .. 150]	> 150
A 1	_Material	$=$ 'Bar'	'Bar'	'Panel'	
A 2	_Machine	$=$ '1606'	'1608'	'1610'	
A 3	State	$=$!OK	!OK	!OK	!ERROR

In the following the individual parts of the decision table are explained.

27.3.1. Condition part

In the condition part the conditions are formulated. It consists of one or more condition lines whose conditions are per rule linked with AND. A condition line consists of a cause variable and one or more condition values.

If a feature or a component was specified as value, the value can be applied from the value list on the right side.

A blank value field means that the condition line of the rule is interpreted as fulfilled. Such a blank value field is therefore called *Don't care field*.

The complete rule condition of a column results from the AND-link of all conditions of a rule column. For the above imaged Rule R1 you get e.g. the following rule condition: Width <50 AND Length IN [10 .. 150].

27.3.2. Action part

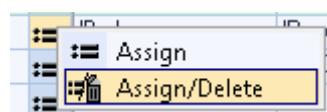
In the action part the assignments are defined. It consists of one or more action lines. An action line consists like the condition lines of a cause variable and one or more values.

Cause variable is a feature or a component. The specification of a feature or a component without the property *Rules enabled* leads to a syntax error.

The value is a simple expression. Its calculated value is assigned to the cause variable. If the value remains blank, the cause variable of the line also remains unchanged.

27.3.3. Rule type

The rule type (column between cause variables and the values) defines if it is an Assignment- or an Assignment/Delete-rule. The rule type can be changed with a right-click on the respective symbol.



27.4. Practice: Special models for the Golf

Three special models for the Golf have to be defined: TrendLine, ComfortLine and SportLine. Each of these special models should have certain basic equipments defined that, once the special model is selected, cannot be changed any more but can only be extended.

Furthermore, the special models should bring price reductions as an incentive to buy certain special packages.

Here you find an overview of the special models, each with equipment and price reduction:

Model	Equipment	Price reduction
TrendLine	Interior fittings fabric, steel rims and accessory radio	400 \$
ComfortLine	Alloy rims, tyres 185, accessories radio and air conditioning	750 \$
SportLine	Metallic paintwork, leather fittings, tyres 205, accessories radio, CD-changer and air conditioning	1000 \$

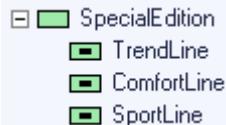
We could e.g. define the special models as a string feature in the *Golf*, but then the price calculation would be too large-scale. And the output in the result would not be carried out automatically.

Therefore it would be a good solution to create classes for the special models and to define the price reductions as negative init values in the classes. Then a component *_SpecialEdition* can be created in the class *Golf* and displayed in the *DetailForm* in a configuration box.

With this all standard mechanisms such as price calculation or Output to result/printout form would work automatically without further assistance. The only thing that still has to be defined is the decision table.

First step is to create a base class *SpecialEdition* under *Modules* and under *SpecialEdition* the object classes *ComfortLine*, *TrendLine* and *SportLine*.

Initialize the prices according to the above table to the object classes. Consider that the prices have to be specified negative, because they are reductions.

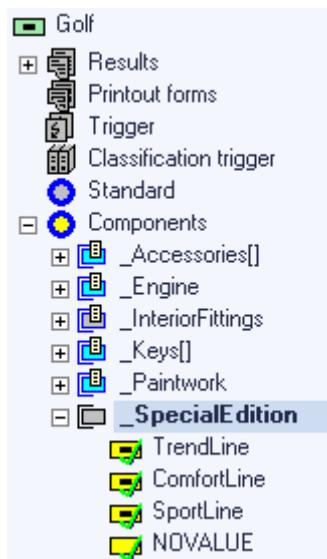


Then you open the class *Golf* and create a new component *_SpecialEdition* from the base class *SpecialEdition*. Apply all object classes and *NOVALUE* (because a *Golf* can also be configured without special edition) as valid values to the structure tree (double click).

Deposit the *NOVALUE naming* “No Special Edition” (property *Translate relevant*) on the component.

The selected special edition and especially the negative price should not be carried out in the module table of the result, therefore you keep the option *Output to result/printout form* disabled. Keep the option *Sort* also disabled, because there is no necessity to change the position of *_SpecialEdition* in the object tree.





The selection of the special model should be carried out via a configbox component. Since only the Golf can be selected as special edition, this is created on the *DetailForm* in the class *Golf*.

 In the class *Golf* you select the item *Overload* in the context menu of the form *DetailForm*.

Remove the label from the form and create a new *configurationbox component* with the cause variable *_SpecialEdition* with the representation *List*.



 Start the application and check if the configurationbox appears in the form and if the list price is correctly reduced with the selection of a special model.

Then you select *New* in the wasele list of the class *Golf* and create a new decision table with the name *SpecialEdition*. Extend and maintain it until it corresponds to the shown table below.

Cond./Act.	R1	2	3
B 1 _SpecialEdition	'TrendLine'	'ComfortLine'	'SportLine'
A 1 _Paintwork			'Metallic'
A 2 _InteriorFittings	'Fabric'		'Leather'
A 3 _Wheels_Rims	'SteelRims'	'AlloyRims'	'AlloyRims'
A 4 _Wheels_Tyres		'185'	'205'
A 5 _Accessories[]	'Radio'	'Radio'	'Radio'
A 6 _Accessories[]		'Air Condition'	'Air Condition'
A 7 _Accessories[]			'CD-Changer'

Figure 61: Editor of a decision table

Test the behavior of the application.



At the moment, the modules are not assigned automatically when selecting a special edition. To find out why, the syntax check is carried out in the decision table.

Call the syntax check in the decision table *SpecialEdition*. Activate the property *Check relevant* for the component *_SpecialEdition*. Activate the property *Rules enabled* for the component *_InteriorFittings*.



Test the application again.

The quotation should now be extended by the display of the special model. Only the name of the selected special model has to be displayed, but no price.

Overload the RTF-constant *!ResultHeader* in the class *Golf*, open the textelement in the textmanager, copy it and paste it.



Change the pasted textelement by inserting the cause variable *_SpecialEdition* between the first line and the product image. Format the text as you like. Assign the textelement with a click on the icon →.



Quotation for a {getNaming()}
{_SpecialEdition}

{giveImage()}

End price including transfer
and {Discount} discount: {EndPrice}

Display in the result:

Quotation for a Golf Individual Comfort Line



End price including transfer
and 1.5 % discount: \$21,763.58

Cash discount: \$489.68

27.5. Tips & Tricks

27.5.1. Process order

The defined rules are checked beginning on the left. In each rule all condition lines are checked from top to bottom. If all conditions of a rule apply, their actions are processed. Then the processing of the decision rule is terminated. This also happens if further rules of the decision table apply.

This has the result that the rule columns in which many condition values remain blank (Don't care fields = the condition is considered as being fulfilled) should be as right as possible in the table so that the more exact specified rules are in front of the more general base rules.

A default value which is automatically set if no rule applies would be solved with a rule column on the very right in which there are no condition rules at all (see screenshot in 27.5.3).

27.5.2. Invalid expressions

If an expression that was specified as condition cause variable cannot be calculated, because e.g. the specified component does not yet exist, the complete decision table is not processed.

If an expression in the action cause variable cannot be resolved, only the line is ignored and the rest of the decision table is processed.

Therefore invalid expressions in the cause variables of decision tables do not lead to error messages, they just cause that the assignments are not carried out.

27.5.3. Use of operators and methods

In the condition lines as well as in the action lines not only direct features and components can be used, but also so-called simple expressions. E.g. calculations can be carried out. In a value for the cause variable *Height* the following term could be set in a condition:

> (Width + Length) / 2

I.e. the height has to be bigger than the average from length and width so that the condition applies.

Cond./Act.	R1	2
B 1 Height	> (Width + Length) / 2	
A 1 StackHeight	:= 4	5

In the same way own methods can be used if they have no side effects (set as flag in the bottom of the method editor). I.e. the method does not change any further wasele, but it only provides the value that has to be compared (in the condition) or to be set (in the action) as return value.

27.5.4. Toolbar

There are icons in the toolbar of the decision table that make the use of the decision table easier:

	No navigation – if this mode is enabled, the edited cell remains selected after pressing the return button.
	Navigation right – after exiting a cell via the return button, the focus is set in the cell right next to it.
	Navigation down – after exiting a cell via the return button, the focus is set in the cell below.
	The scrollbar can be faded in either for the complete decision table or only for actions. This is very useful with longer tables.
	With the focus in the rule columns the values of the respective cause variables are faded in or out.
	Saving as CSV-file. Via this icon the decision table can be saved as csv-file and then formatted and printed in Excel. The type of the action (rule type) is specified with A like Assign or AD like Assign/Delete.

27.5.5. Keyboard shortcuts

For an easier creation of lines and columns in the table there are the following keyboard shortcuts:

<Ctrl> + <Cursor key up>	The decision table is extended by one line above the current cell.
<Ctrl> + <Cursor key down>	The decision table is extended by one line below the current cell.
<Ctrl> + <Cursor key left>	The decision table is extended by one column left of the current cell.
<Ctrl> + <Cursor key right>	The decision table is extended by one column right of the current cell.

27.6. Repetition

- Decision tables are tabular combinations of assignment- and assignment/delete rules.
- Decision tables consist of condition lines and action lines.
- Rules are imaged in columns.
- Cause variables in action lines must have the rules property set.
- Blank cells in the decision table are ignored with actions, with conditions they are considered as fulfilled (Don't care = rule condition fulfilled).
- The processing of the table is carried out from left to right.
- If all conditions of a rule column are fulfilled or "Don't care", the action lines of this rule column are carried out and then the processing of the decision table is terminated.

28. Users and security

28.1. Intention

camos Develop is multiuser-capable, i.e. several users can work parallel on the same database and even on the same knowledge base. In order to guarantee that only the users that are involved in the project have access to the necessary knowledge bases, frames, etc., certain authorizations can be defined.

To do so, first all users must be registered on the database. Then they can be taken up in the authorization maintenance of the corresponding knowledge bases.

In this chapter we deal with the user maintenance and the basic access mechanisms. You will learn how users are created and administrated and how to grant or withdraw them authorizations on the individual parts of the development system.

28.2. User maintenance

camos Develop principally uses the username of the operating system to identify and login on the system. If you are e.g. logged in on Windows with “UserX”, the username “UserX” has to be registered in the camos Develop development database. Otherwise the following message appears with the start of camos Develop:



In order to create a new user “UserY” in the database, you open the *User maintenance* from the main menu *Window*.



Figure 62: User maintenance

On the left side of the dialog you see the existing user groups. Any amount of users can be allocated to a group. The allocation of the authorizations can also be made on individual users, but it is more clearly to subdivide the users in groups.

The maintenance of the users is carried out in the right part of the dialog. You can create a new user via the icon  in the toolbar of the table.

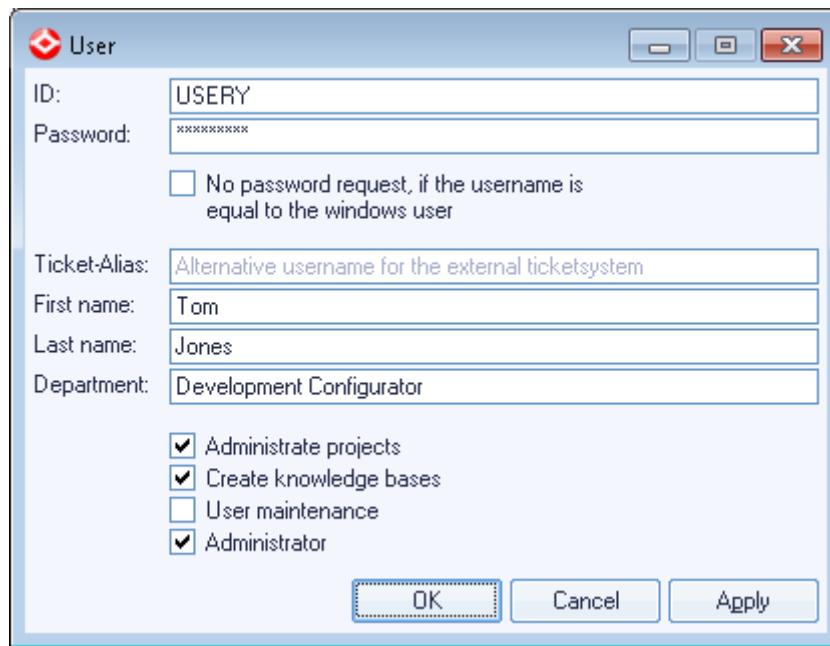


Figure 63: Creating a new user

In addition to the username which by default has to correspond to the Windows-ID, name, first name and department can be specified. If a password is specified, a dialog to enter the password is displayed with the start of camos Develop. If no password is specified, camos Develop starts without query dialog.

Via the option *Administratate projects* you can define, if projects may be created, edited and versioned. In addition you can define if the user is allowed to use the in camos Develop integrated camos DeploymentCenter (additional license required).

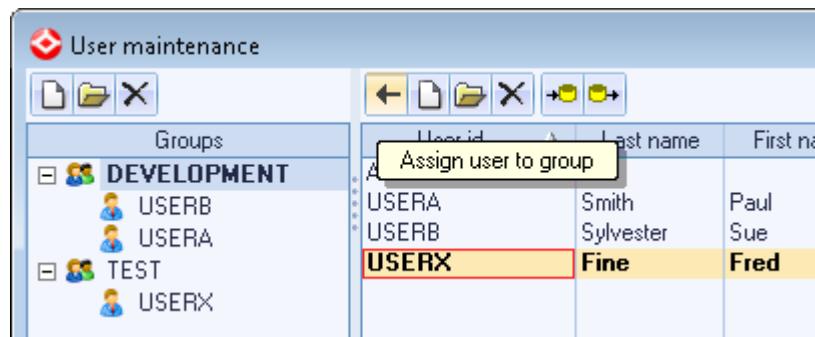
If a user should be allowed to create knowledge bases, he has to have the authorization *Create knowledge bases*.

Via the option *User maintenance* you can decide, if the user has the right to open the user maintenance. Every user that has this authorization, can edit all data of the users. If the option is not selected, then the menu item *User maintenance* is deactivated for this user.

If a user who is not an *Administrator* wants to change his password, he can do this in the dialog *Extras -> User options*.

Via the option *Administrator* is determined if the user has the authorization to delete all knowledge bases and to change their securities, independently if the user has the appropriate rights on the knowledge base or not.

After creating the user with OK, the user can be assigned to one or more user groups. To do so, you have to select the user in the table and the desired group in the left tree and click the icon ←.



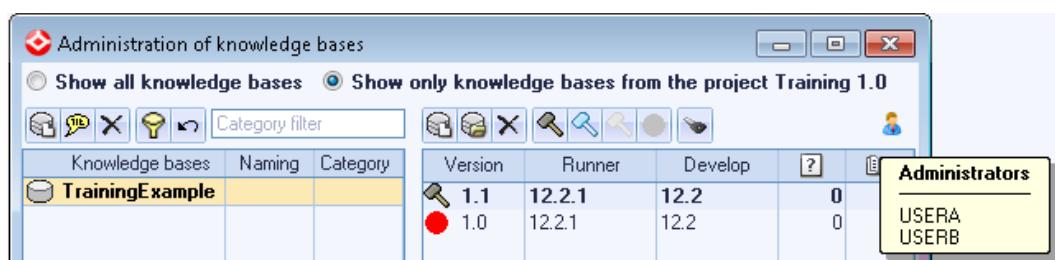
Via the icons in the upper right corner you can ex- and import the user maintenance. This is for example a great advantage when you change the development database. So you don't have to create all users and groups manually.

28.3. Securities on the knowledge base

For each knowledge base you can define which users are allowed to open and edit it etc. This assignment can be made, when the knowledge base is opened, via the context menu item *Knowledge base security...* of the class tree root. The securities can also be changed without opening the knowledge base. To do so open the knowledge base administration via the main menu item *Knowledge base -> Administrate...*.

On the example of the maintenance dialog: In the dialog *Administratate knowledge bases* first the desired knowledge base is selected from the left table. Now you can see all versions that exist on this knowledge base in the right table (for knowledge base versions see chapter 22).

In the toolbar above the left table, you can see the icon and the names of the users and user groups (in braces) which have administrating rights on this knowledge base.



The key-icon stands for *Security...*. You find it everywhere in camos Develop where authorizations are administrated.

In this case the icon opens the dialog in which is defined which users have certain authorizations on the selected knowledge base. The user who created a knowledge base has automatically all rights.

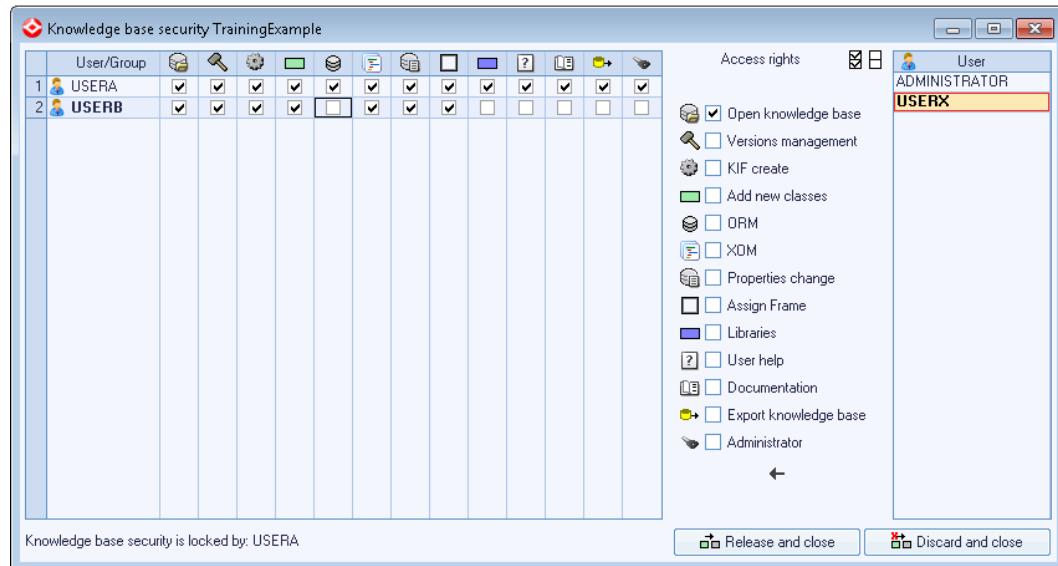


Figure 64: Securities of the knowledge base

In the right table you see all user groups that were defined in the user maintenance. Via the icons or you can switch between the user groups and individual users. Via the icon the user/the group that is selected in the table is applied with all rights that are set in the middle of the dialog.

Class tree/Context menu of the heading in the class tree/Knowledge base security

28.4. Securities on the frame

Authorizations can also be assigned for the frame. However, it is only differentiated between *Edit* and *Security...*

I.e. the frame can be opened by any user. The authorization *Edit* determines if a user is allowed to make changes. The authorization *Security* stands for the administrator authorization. I.e. the user can open and edit the securities of the frame.

The allocation of the frame securities is carried out in the *Frame maintenance*. After selecting the desired frame in the table you will find the icon in the toolbar. On the upper right side of the dialog you also see in braces the users who are administrator of the frame.

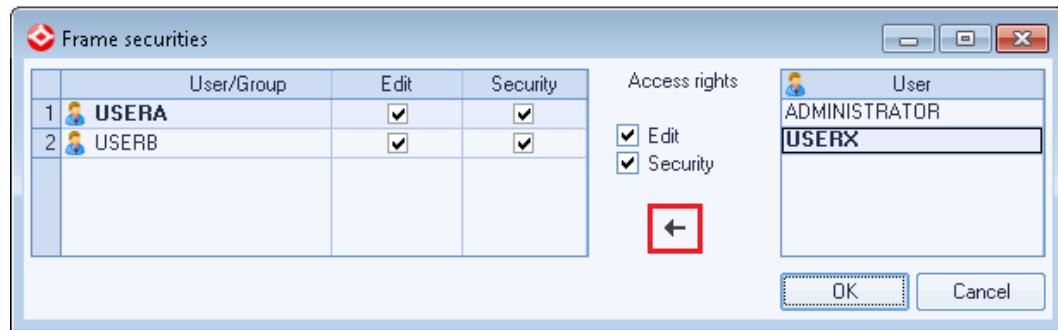


Figure 65: Securities of the frame

28.5. Securities on classes

In addition to the securities on the knowledge base, a user also needs the authorization on a class in order to be able to edit it. Via the context menu item Security... on a class these rights can be administrated.

Four different authorization levels can be selected:

No access

The user cannot open the class. All context menu items that trigger changes on the class are disabled.

Read

The user can open, copy, etc. the class only reading.

Edit

The user can open, reserve, copy, delete, etc. the class.

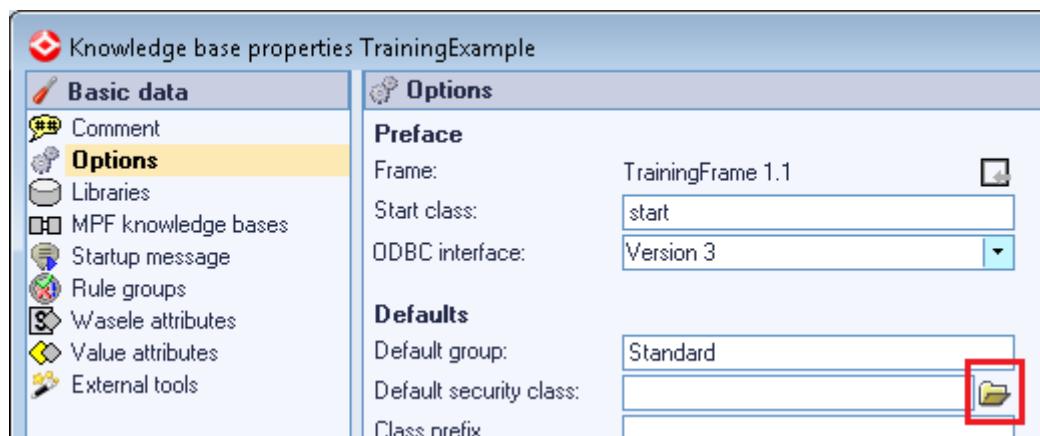
Security

The user can edit the class and additionally allocate rights for this class.

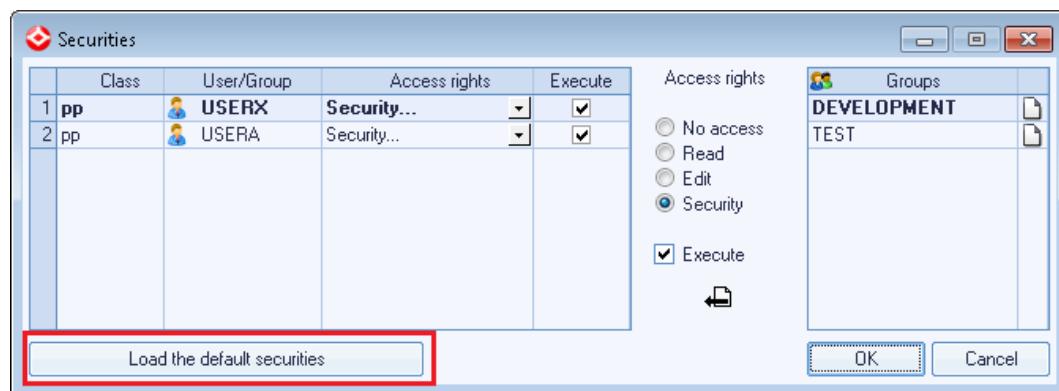
Additionally there can be determined (via the switchbox *Execute*) if the user or the members of a user group have the authorization to start a debug run with this class.

28.5.1. Default securities

In the options of the knowledge base (see chapter 16.4) can be defined under *Default security for class* which users, except the one who creates the class, should have which rights on new created classes by default.



Via the button *Load the default securities* in the security dialog of a class, the existing rights can be overwritten with the rights that are defined in the options.



28.5.2. Collective changing of the securities on classes

Changes on the class securities can be made for several classes simultaneously with first selecting several classes in the class tree (via **Ctrl+Click**) and calling the security dialog on one of the selected classes.

In order to define the authorizations for all classes of the knowledge base, the menu item *Security...* in the context menu of the class tree root is selected. In this dialog you additionally have the option *Overwrite* with which the current allocation for all classes is applied.

In the following figure e.g. the user group "Development" that contains in addition to "UserA" and "UserX" also the "UserY" (see chapter 28.2), is equipped with the *Security authorization* incl. *Execute* and assigned with the enabled option *Overwrite* to all classes.

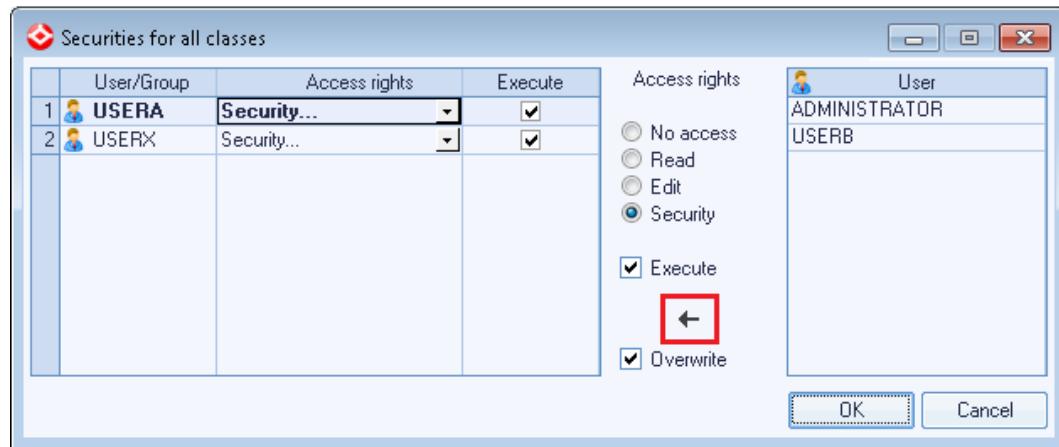
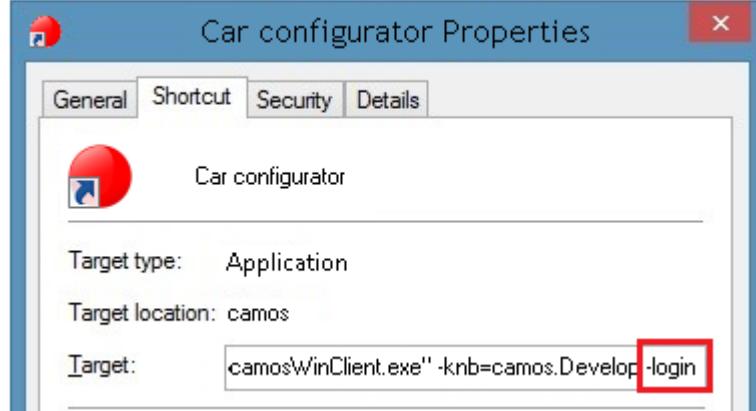


Figure 66: Securities of a class

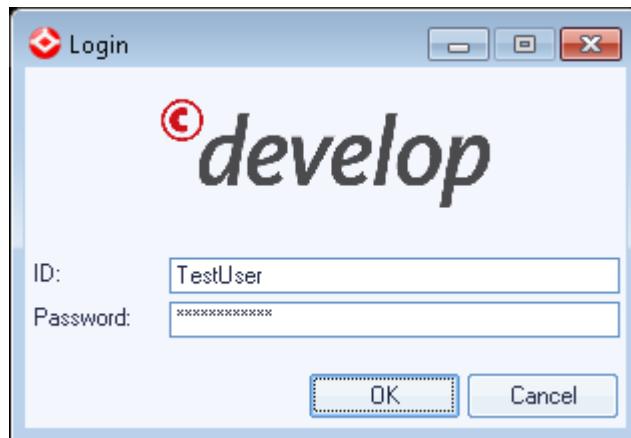
28.6. Tips & Tricks

28.6.1. Login with different username

By default camos Develop uses the name which you login on the operating system. In order to use a different username, you have to specify the start parameter *-Login* in the properties of the icon with which you start camos Develop.



In this case a login form in which you have to identify yourself appears when you login on camos Develop.

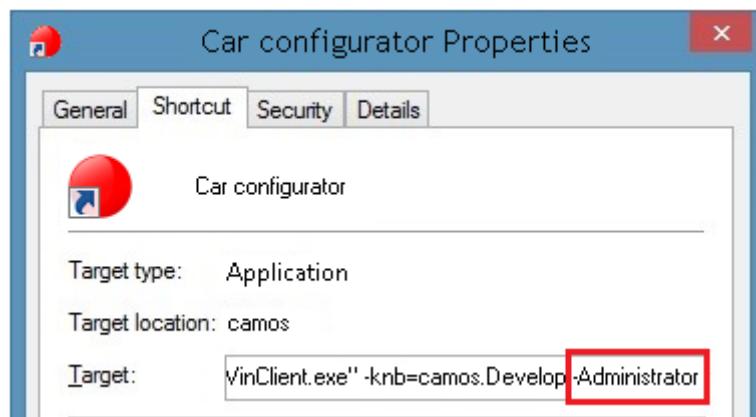


 This user (in the above case "TestUser" has to be also registered on the database and in the user maintenance! If a password for the user is not deposited, the field remains blank.

28.6.2. First start of the user maintenance

Chapter 28.2 describes how to create a new user in the database. This procedure assumes that you are already registered as user. However, directly after the installation of camos Develop this is not the case. Therefore you get the error message "The user is not registered" when you login for the first time.

With the creation of the database tables for camos Develop, the user "Administrator" with the password "camos" is automatically created. In order to get access to the user maintenance and to be able to create further users, you have to login as administrator. This is carried out via the start parameter *-Administrator* which you specify in the properties of the icon with which you start camos Develop.



 Then you start camos Develop and enter "camos" in the password query. Now you are registered as administrator and are able to create the required users, at least your own one, in the user maintenance.

Don't forget to activate the option *User maintenance*. Otherwise your own username cannot open the user maintenance.

In order to login afterwards with your own username, remove the parameter –*Administrator* again.

28.7. Repetition

- Each developer who wants to login on the development system has to be first registered via the user maintenance.
- The user maintenance is located in the main menu *Window*.
- For each knowledge base, frame and class it has to be defined which users have access rights and with which authorization (reading, writing, etc.).
- If a user has the authorization *Security...* he can remove or add other users from the securities.
- In the options of the knowledge base default securities for new created classes can be defined.

29. Import and export knowledge bases

29.1. Intention

In this chapter we deal with the im- and export mechanisms of camos Develop.

The exporting of a knowledge base is necessary if the knowledge base e.g. has to be transferred in another database or if – especially before the update to a new version of camos Develop – a backup has to be made.

In this chapter you will learn how a knowledge base is exported and imported and what has to be considered in doing so.

29.2. Knowledge base export

Before you export a knowledge base, all classes should be released. Classes that are reserved by the own user are exported with the current state. Classes that are reserved by other users are applied to the export with the last released state.

The export dialog is called via the context menu item *Export...* of the header of the class tree.

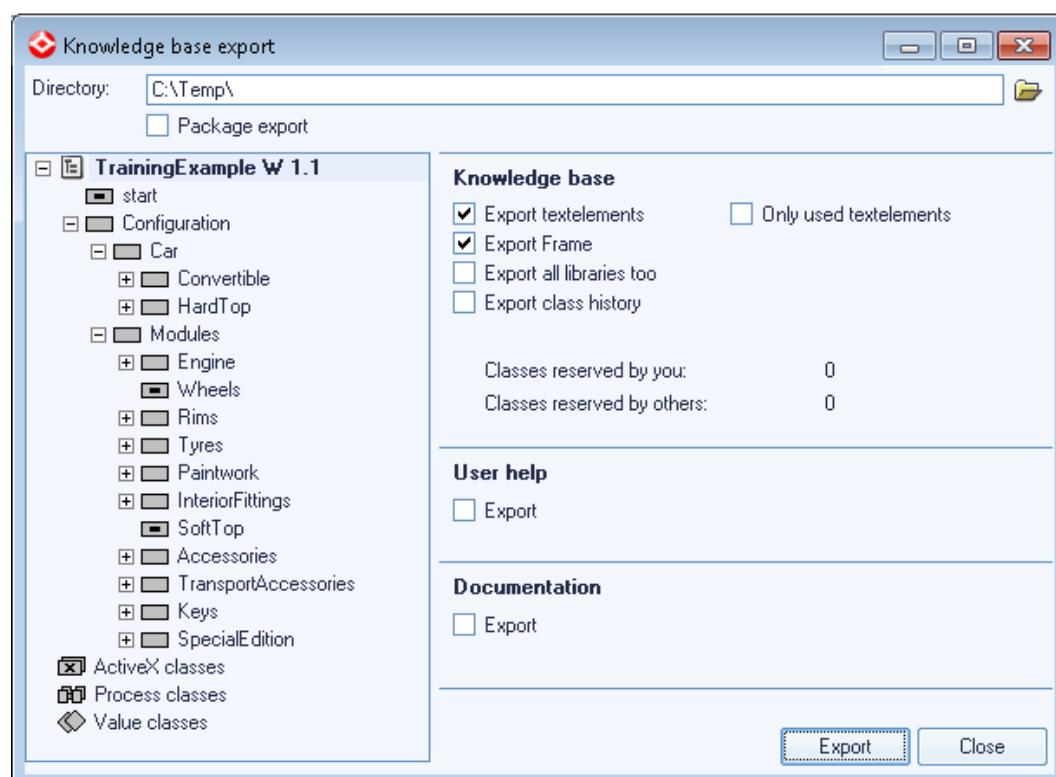


Figure 67: Export knowledge base

In the left part of the dialog the class tree of the knowledge base is shown. By selecting individual classes it is determined which classes have to be exported.

For flagged base classes can additionally be decided if the derived child classes should be exported too. In order to export all classes, the class tree root is flagged.

It can furthermore be determined if the frame of the knowledge base, possibly embedded library knowledge bases and the user help and documentation also have to be exported.

The files that are created with the export have the following file extensions:

Extension	Element typ	Explanation
KBX	Knowledge Base Export	Export of a knowledge base. Contains all classes, rule groups, library allocation, etc.
CTM	camos Textmanager	Export of the textelements of Public, the knowledge base and the linked libraries. This export can also be created manually: Inside of the textmanager, main menu <i>Extras -> Export....</i> This file can be imported into the textmanager (of this or another knowledge base) again.
FMX	Frame Export	Export of the frame that was assigned to the knowledge base. This export can also be created manually: Select a version of a frame in the frame maintenance and select the context menu  Export.
HLX	Help Export	Export of the user help. The file has the same structure as a DOX-file, i.e. it can also be imported as documentation.
DOX	Documentation Export	Export of the documentation. The file has the same structure as a HLX-file, i.e. it can also be imported as user help.

The names of the export files of knowledge bases, documentations and user helps consist of the name of the knowledge base and a time stamp, e.g. the file *CarConfigurator-1.1w-2006-06-06-13-50.dox* is an export of the documentation from the knowledge base *CarConfigurator* in the work version 1.1 and was created on 06.06.2006 on 13:50.

Since frames are not versioned, this specification is not part of the filename of a frame export.

29.3. Practice: Export the car configurator

Release all classes (Release and unlink from ticket) and select the menu item *Export...* from the main menu *Knowledge base*.

We want to export the knowledge base and the textelements and then import it into the same database under a different name. To complete the picture we will also carry out the ex- and importing of the frame.

Mark the class tree root and activate the switchbox *Export Frame* too. Select a directory in which the export files have to be stored, e.g. to C:\Temp\.

Start the export.



Now you will find three files in the defined directory: the knowledge base export (*.kbx), the frame export (*.fmx) and the textelement export (*.ctm).

29.4. Knowledge base import

The import can only be carried out into the work version of an existing knowledge base. In order to avoid runtime conflicts, the knowledge base should have a frame with the same settings that exist in the export. If e.g. the main language is different, this is displayed in a warning message.

The import dialog is opened in an opened work version via the context menu item *Import...* of the header of class tree.

Then the file is selected that contains the data that has to be imported. After pressing the button *Open* the dialog *Import knowledge base* is opened. This represents the class structure that is contained in the export. Exported classes are indicated green, not exported classes are indicated gray. The represented structure is always complete

If the import is carried out into a not blank knowledge base, you can see on the tab page *Existing classes* a list with all classes, that exist two times. In the next step a question message is displayed, how to deal with the existing classes. You will learn more about this option in chapter 29.6.1.

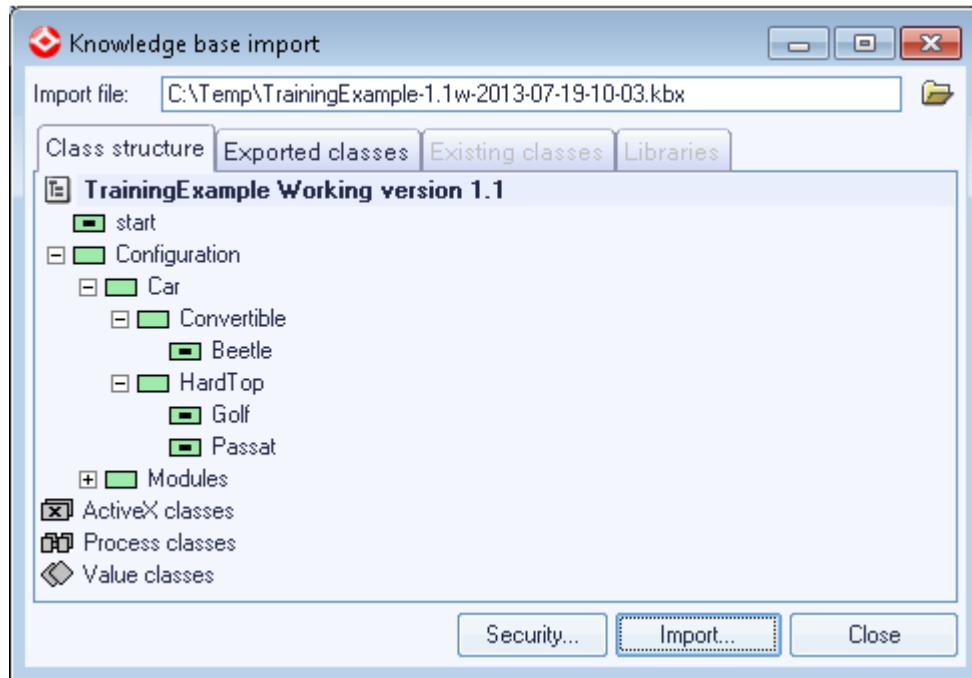


Figure 68: Import knowledge base

Via the button *Security...* you can already define which users should have which rights on the imported classes before the import, because access rights are not saved in the export. The import is commenced with *Start*.

After the import a log of the imported classes with specifications to time and size can be displayed via activating the switchbox *Display statistics*.

The screenshot shows the 'Knowledge base import' dialog box after the import is completed. The 'Import file:' field still contains 'C:\Temp\TrainingExample-1.1w-2013-07-19-10-03.kbx'. The 'State:' field shows 'Done'. A checkbox labeled 'Display statistics' is checked. Below is a table with the following data:

	Class	Time	Size
1	SportLine	0.015	879
2	ComfortLine	0	878
3	TrendLine	0.016	876
4	SpecialEdition	0	595
5	EmergencyKey	0.016	873
6	RadioKey	0	869
7	Keys	0.015	2,972

At the bottom are 'Security...', 'Import...', and 'Close' buttons.

?

Knowledge base/Import ...

?

Knowledge base/Export ...

29.5. Practice: Import frame and knowledge base

We simulate the importing on a completely new and empty database. In order to create a new knowledge base, first a frame is required. Therefore you create a new frame and import the frame Export that was constructed in practice 29.3 in the new created frame.

Open the *Frame maintenance* and create a new frame with the name *BackupFrame*. Select the context menu *Import* →  on the version 1.0 in the middle table. Select the frame-export file in the file selection dialog.

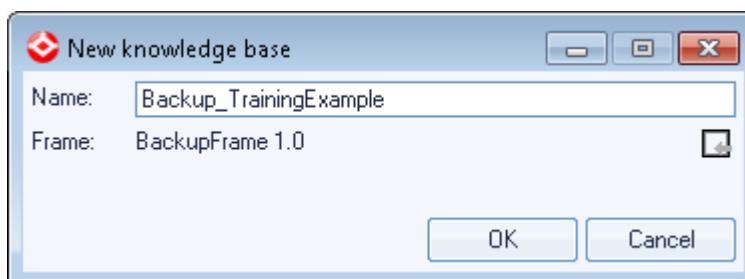
Click *OK* to import the frame with the new name. After the completed import close the frame and the *Frame maintenance*.



Now you create a new knowledge base that uses the *BackupFrame* and import the knowledge base export from practice 29.3.

Select the main menu item *Knowledge base -> New...* to create a new knowledge base. Allocate the name “*Backup_TrainingExample*” and assign the imported frame *BackupFrame*, that has been imported in the last step.

The now created knowledge base is automatically added to the opened project and can be edited right away.



After creating the knowledge base with *OK*, select the menu item *Import...* from the context menu of the class tree header. To do so select the file, that has to be imported, via the icon  and confirm it with *OK*. Confirm the import with *Import*. After the import you can take a look at the statistics if desired.



The classes were released in the ticket after the import. This is visible in the yellow color of the classes. You can either release the classes and unlink them from the ticket or you can release the ticket.

Now only the textelements are missing, which were exported into a separate file during the knowledge base export. The textelements will be imported directly into the textmanager.

Open the textmanger and import the textelements via the menu *Extras -> Import...*. Select the *.ctm file which was created in chapter 29.3 via the icon .



Because the knowledge base in which the textelements should be imported into has another name as the knowledge base where the textelements were exported, the target knowledge base is not known. This is visible because of the ??? in the left, bottom table in the column knowledge base.



Select the knowledge base *Backup_TrainingExample 1.0* from the context menu of the ???.

Start the *Import* with the corresponding button in the dialog.

Knowledge base	Import file	Languages
+ Public	Public	<input checked="" type="checkbox"/> English <input checked="" type="checkbox"/>
+ ???	TrainingExample	<input type="checkbox"/> Deutsch (Deutschland) <input checked="" type="checkbox"/>
	Public	Backup_TrainingExample 1.0

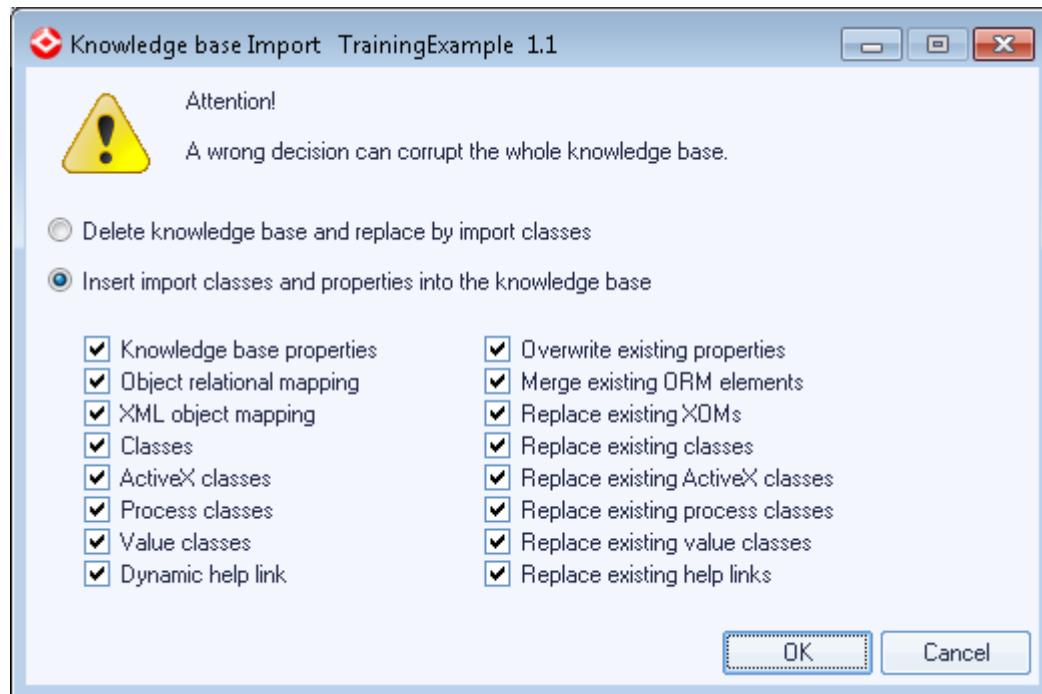
The knowledge base *TrainingExample* incl. frame and textelements was exported and re-imported. Since the two knowledge bases and frames are absolutely identical, you can delete the elements that were created in the last practice:

- 1.) Delete knowledge base: Open the dialog *Administration of knowledge bases*, select *Backup_TrainingExample* in the left table and select the icon from the toolbar.
- 2.) Delete frame: Open the *Frame maintenance*, select the *BackupFrame* in the table and select the icon from the toolbar.

29.6. Tips & Tricks

29.6.1. Import into an existing knowledge base

If the import is carried out into a knowledge base that already contains classes, the following query is displayed:



Delete knowledge base and replace by import classes

This option deletes all classes and properties (except the documentation and user help) of the knowledge base before the import is carried out. The deletion cannot be made undone! Even if the import is canceled later, the classes are lost and cannot be restored.

Insert import classes and properties into the knowledge base

It has to be specified, how it should be dealt with existing elements. It can be decided for every knowledge base property (comment, rule groups, feature-/component-attributes, value-attributes and options), ORM, XOM, the classes, ActiveX classes, Process classes, Value classes and dynamic help links whether they should be imported or existing elements with the same name should be maintained.

29.6.2. Package-Export and –Import

With a complex knowledge base structure it is very comfortable to export the knowledge base with all included libraries. Therefore there is the option *Package-Export* in the export-dialog

Via this export all knowledge bases, libraries, the frame and the Textelements are exported into a single file (ending *.pgx).

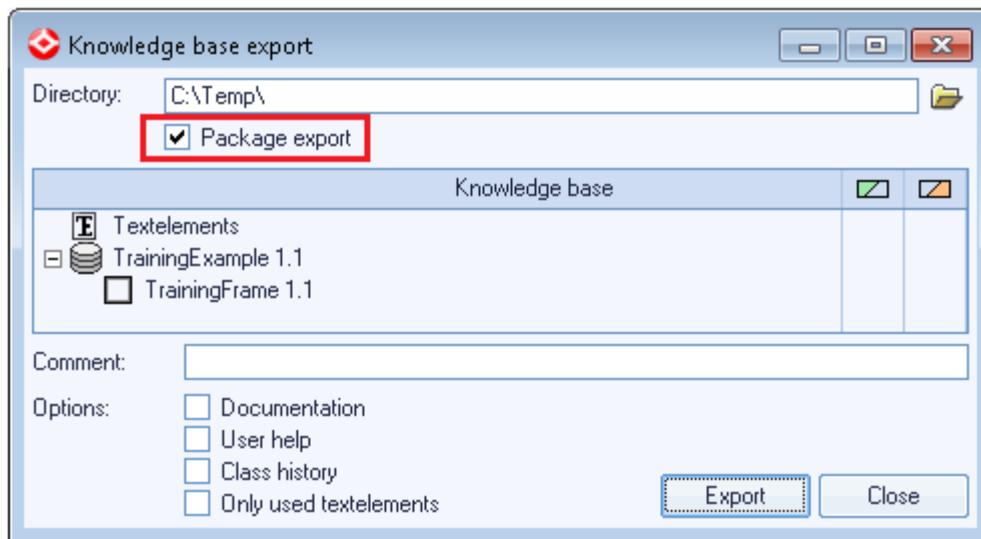
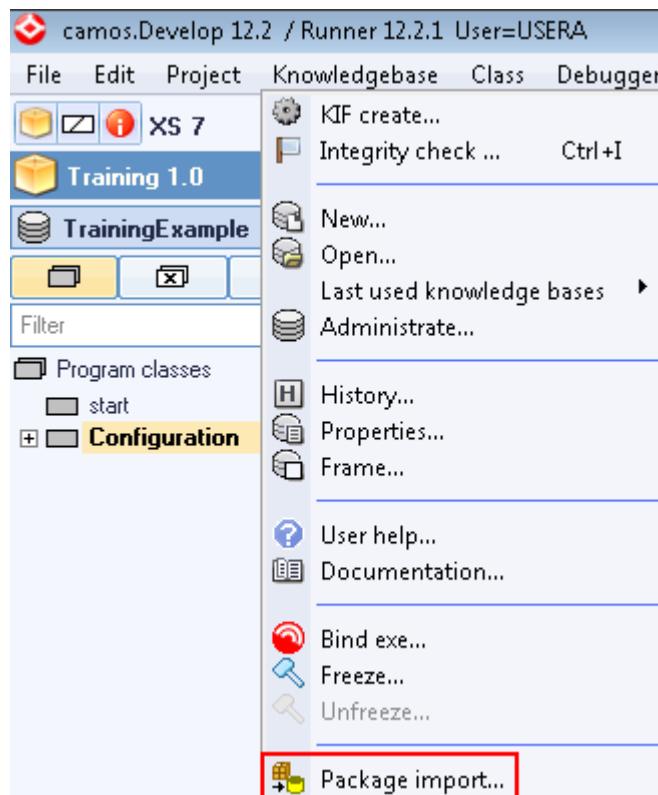


Figure 69: Package-Export

This pgx-file is imported via the main menu *Knowledge base -> Package import...*



After selecting the pgx-file the elements that are contained in the package are displayed. If these elements exist already with the same name and the same version in the data base, they are overwritten. The elements which will be overwritten with the import are marked with the

icon . It is however possible to exclude parts of the package export from the import by using the context menu of the columns *Target* and / or *Operation*.

Knowledge base/Package import

Class tree/Context menu of the heading in the class tree/Export .../Package-Export

In the following figure the TrainingExample should be imported into the same data base. Because the knowledge base and the frame already exist, in the import dialog it is identified via appropriate icons that these elements will be overwritten.

Via the context menu of the columns *Target* it is possible to define another target knowledge base and target frame. Via the context menu of the column *Operation* elements can be excluded from the export or they can be imported into new created elements.

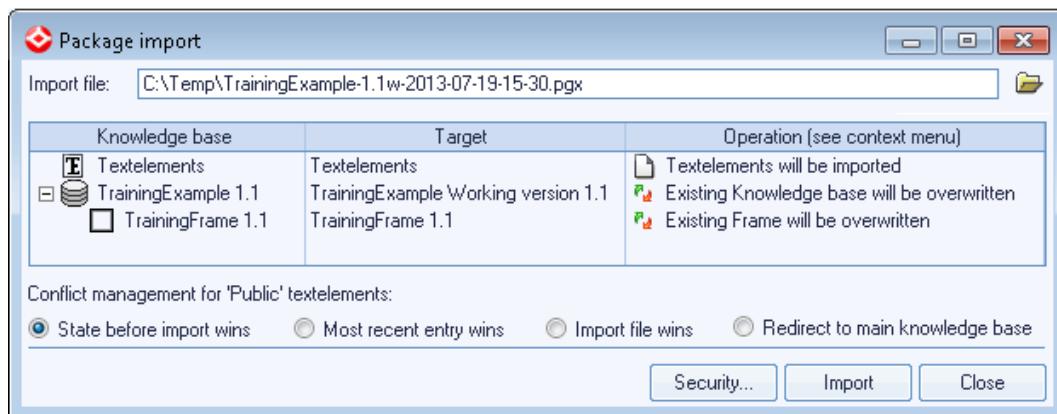


Figure 70: Package-Import

29.6.3. Project-Export

Besides the knowledge base export (*.kbx) and the package export (*.pgx) it is also possible to export a project with all included knowledge bases. The knowledge bases do not have to be linked to each others via libraries.

This possibility is reachable via main menu Project -> Project Export. The project export is divided into the project structure export and the project package export.

The project structure export creates a new *.psx file, which contains the structure of the exported projects. This means that the names and versions of the included knowledge bases will be exported. The project structure export is imported into an existing project. This kind of export will be used e.g. when the project should be "copied" on the same database.

The project package export creates a *.pjx file, which contains all the knowledge bases of the project. The project package export is imported into an existing project as well. With the project package export, the included knowledge bases and their libraries can be imported individually or all together. This kind of export will be used when the knowledge bases of a project should be imported into another database.

29.7. Repetition

- An export can be created from the complete knowledge base, individual classes, the frame, the documentation and user help. This is either carried out collectively in the export dialog of the knowledge base or separately for each element.
- The filename of the export files is put together of the name of the knowledge base, its version and a time stamp. With frames the specification of the version is dropped.
- The import (except for the package-import) that to be carried out into an existing work version of a knowledge base. With this all elements (frame, documentation, user help and knowledge base) have to be imported separately.
- The import can also be carried out into a not blank knowledge base. With this has to be set how to proceed with the existing classes.
- Via the package-export a complex knowledge base structure can be exported into a single file and be imported again.
- Via the project export either the structure of a project or all knowledge bases of a project can be exported into a file.

30. From the knowledge base to the application

30.1. Intention

Up to now the only possibility to start the application is via the debugger in the development system. This means that momentarily each user of the configurator, e.g. the employees in the sales, have to have an installed camos Develop version inclusive the required license and development database.

In this chapter you will learn what has to be done to be able to operate an application without the development system.

30.2. What is a KIF?

In order to be able to start the configurator without an installed development system, a **KIF** is created.

The abbreviation KIF stands for **Knowledge Information File**. The KIF is a binary file that can be started independent of the development system with the camoswinclient.exe (frontend) and the camosRunnerPE (PE = PersonalEdition; local runtime environment) or the camosRunnerSE (SE = ServerEdition; operation via an application server).

During the development phase the application was started and tested out of camos Develop via the debugger. Unlike this, the KIF-file represents a run-capable application for users that are not involved in the development. It is runtime-optimized and no longer connected to the database in which camos Develop files the data during the development.

The execution of the application as a KIF requires a license of the type “camos.Configurator”.

Definition

Any amount of KIFs can be created from a knowledge base (from a work- as well as from a release version).

30.3. Practice: Create KIF

It is recommended to check if the knowledge base contains errors first.

Select the menu item *Integrity check* in the context menu of the class tree root. Activate all options and start the integrity check with the button *Check*.

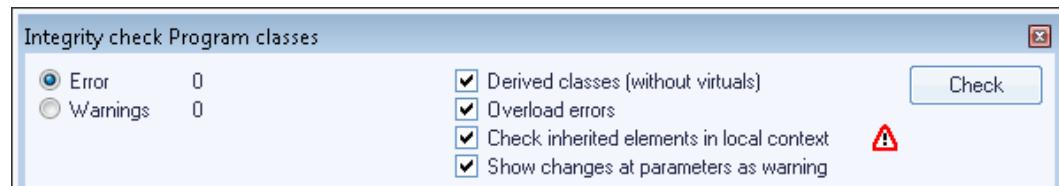


Figure 71: Integrity check of the complete knowledge base

After successfully concluding the check, all reserved classes should be released. This is not absolutely necessary, because the classes that are reserved by you are applied with the current state. Classes that are reserved by other users are written with the last released state. In order to guarantee a uniform development state, all developers should therefore release their changes.

For a collective releasing of all classes you have two possibilities.

Possibility 1:

Select all reserved classes manually in the class tree (multiple selections with pressed Ctrl-button) and select *Context menu -> Release*.



Tip: In order to completely open the class tree, you mark the class tree root and press the button * (multiplication) on the numeric keypad.



Possibility 2:

Select the icon from the toolbar of the class tree. Then the dialog *Reserved classes* is opened. In this dialog you can choose between the following views or you can combine them:

- (green) classes that are reserved by yourself
- (yellow) classes that are released in the ticket
- (orange) classes that are reserved by another user or in another ticket
- (gray) classes that are released publicly



Select all the classes that are reserved by you and select the icon from the toolbar to release these classes and unlink them from the ticket.

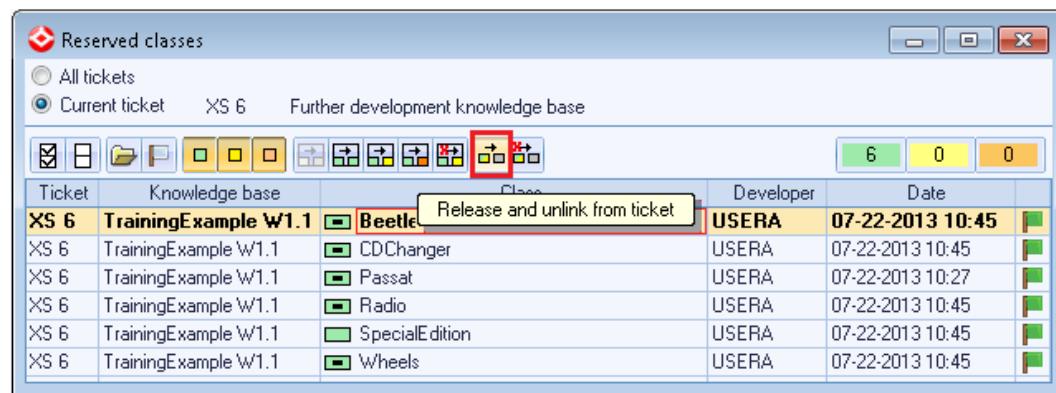


Figure 72: Dialog “Reserved classes”

After an error-free state of the application was fixed by releasing the classes the KIF can be created.

Select the main menu item *Knowledge base -> KIF create* or the icon  from the toolbar of camos Develop.



In the dialog *KIF create* you can select the directory in which the KIF has to be saved via the icon . By default the directory *C:\Program Files\camos\kif* which was created with the installation is preset.

If you work via an application server, the KIF has to be filed in the KIF-directory of the application server!



Specify the default KIF-directory and click on *OK*.

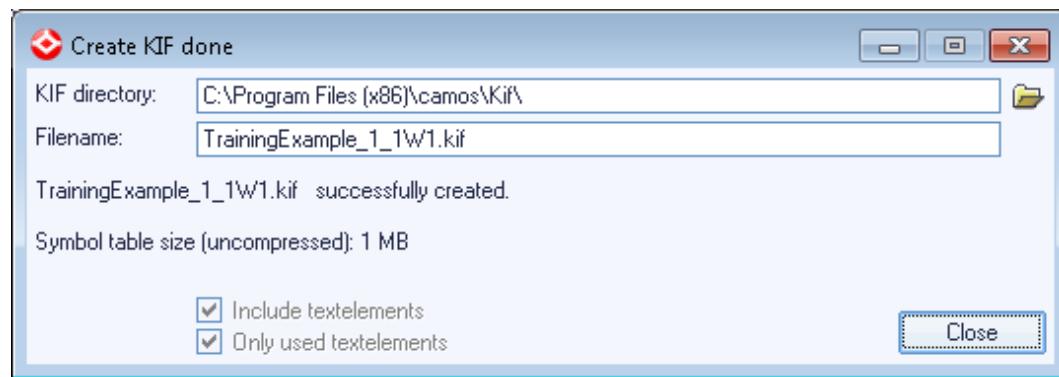


Figure 73: Create KIF

The suggested filename of a KIF consists of:

- the name of the knowledge base, in our case *TrainingExample*
- the version of the knowledge base, here the work version 1.1. Work versions are indicated by a *w*, therefore *1_1W*.
- the number of the KIF that was created from the work version of a knowledge base, in our case the first one.

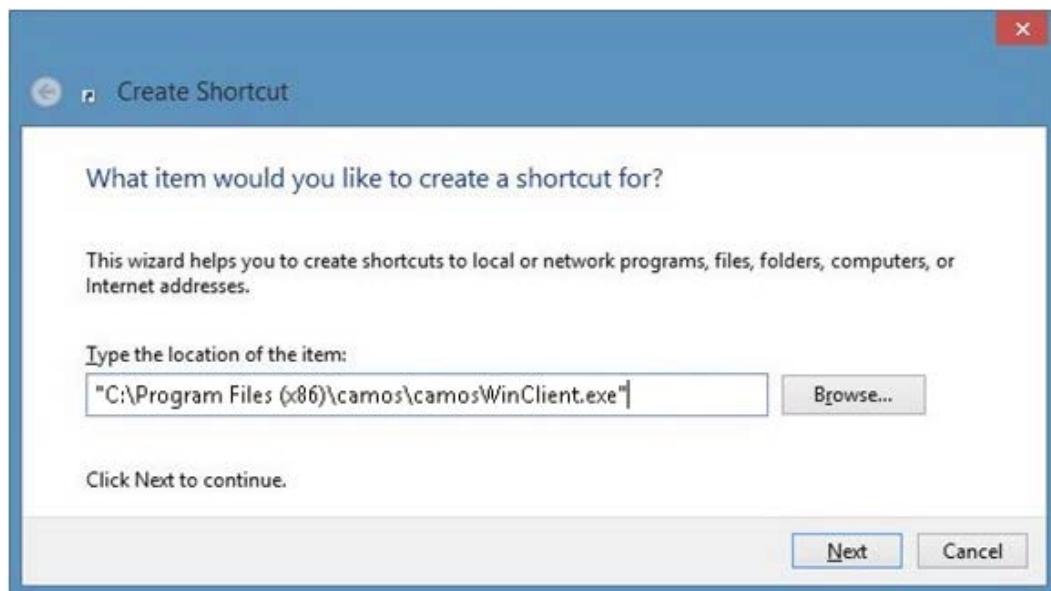
You can also set the filename by yourself.

If a KIF is created from a release version, no numbering is added to the filename, because release versions cannot change anymore. All KIFs of the release version *TrainingExample 2.0* would e.g. always have the filename *TrainingExample_2_0.kif*.

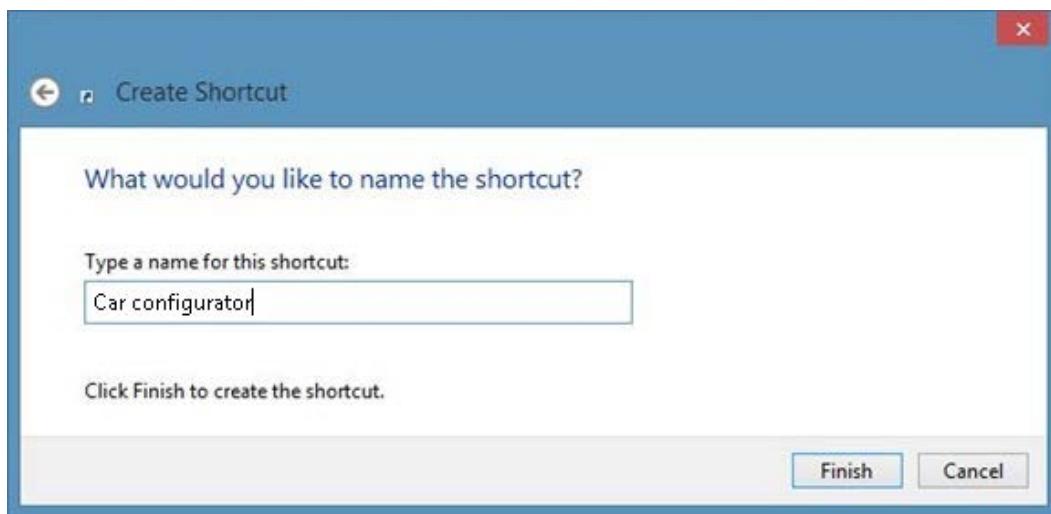
30.4. Practice: Start KIF

Since the KIF contains only the information from the knowledge base, it cannot just be started via double click. It has to be linked with the frontend, the camoswinclient.exe.

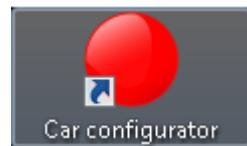
Carry out the context menu item *New -> Link* on the desktop. In the appearing dialog you click on *Browse* and select the file *camoswinclient.exe* from the program directory *C:\Program Files\camos*.



Click on *Next* and allocate any name to the link, e.g. "CarConfigurator". Then you click on *Finish*.

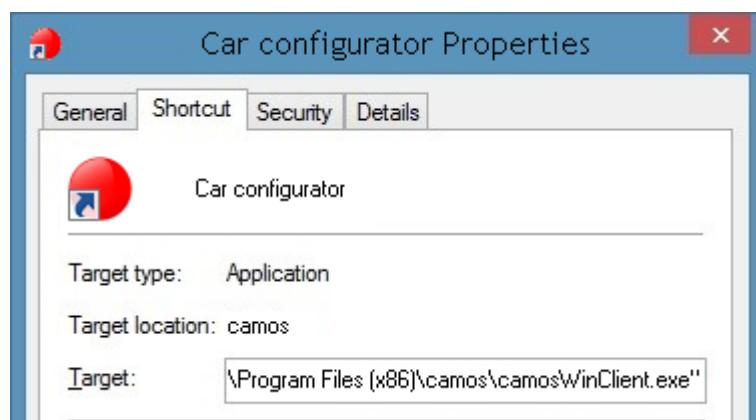


On your desktop you will find the link to the WinClient (camoswinclient.exe):



Via the properties of this link parameters can be transferred to the WinClient. The WinClient evaluates the parameters with the start.

Carry out the context menu item *Properties* on the link. In the field *Target* of the tab page *Shortcut* you can see the path to camoswinclient.exe that was specified with the creation of the shortcut.



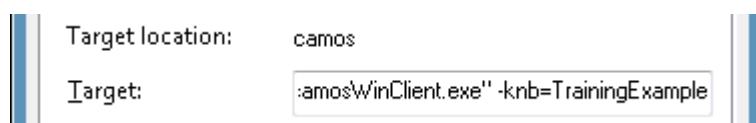
All start parameters are transferred in the syntax <Parameter name>=<Value>. The character – can be replaced by / . If – or / is left out, the parameter is not considered. If the transferred value contains blanks, it has to be enclosed in double high-inverted commas.

The first and only not optional start parameter is *-knb*. KNB stands for knowledge base and tells the WinClient which KIF he has to start. The name of the KIF that was created in the previous practice 30.3 is transferred as value.

Only the name of the KIF (-knb=TrainingExample), optionally the version (-knb=TrainingExample_1_1w1), can be transferred to the start parameter *-knb*. The specification of the complete filename (-knb=TrainingExample_1_1w.kif) would lead to an error message with the start of the link.



Complete the field *Target* by the expression “-knb=TrainingExample”.



Since any amount of KIFs can be created to a knowledge base, it has to be additionally defined which KIF has to be started. This is carried out via the start parameter `-ver`. VER stands for *Version* and can be transferred as follows:

- `-ver=w` starts the latest KIF of the highest work version (**w** = work version)
- `-ver=2.0` starts the KIF of the release version 2.0
- `-ver=3.6w` starts the latest KIF that was generated from the work version 3.6

If the parameter is not specified, the specification of the KIF-name is relevant.

For `-knb=TrainingExample` and a missing ver-parameter the KIF of the newest release version is started.

If `-knb=TrainingExample_1_1w1` was specified, then `-ver` must not be specified, because the version was already specified with the filename. Here the parameter `-ver` has to be omitted.

 Complete the field *Target* by the expression “`-ver=w`” in order to always use the latest KIF of the *TrainingExample*.

Target location:	camos
Target:	<code>'inClient.exe' -knb=TrainingExample -ver=w</code>

Figure 74: Call parameter to start the KIF

 In the last step you apply the changes on the properties with *OK* and doubleclick on the link.
Does the configurator start? Congratulations on your first camos Develop application!

30.5. Tips & Tricks

30.5.1. Further start parameters

 [Basics/Start parameters/Start parameters for camos Develop](#)

30.5.2. How to define your own parameters

In addition to the start parameters that you learned to know in the previous practices, you can transfer own parameters to the WinClient link which you evaluate in the knowledge base.

If e.g. in the application accesses to a database occur whose data source name (name of the ODBC-link) is not fixed predefined in the application, but individually transferred by the user.

To do so, you determine any name for the parameter, e.g. "ArticleDSN". Within the knowledge base you can query the value of the parameter via the function `GetParam(<NameOfParameter>)`, e.g.

```
ODBCName := GetParam( „ArticleDSN“ );
```

If e.g. `-ArticleDSN=DBArticle` was specified in the WinClient link, `ODBCName` contains afterwards the value "DBArticle" and can e.g. be further used for the connection to this database.

Within the development system start parameters can be tested in defining them as parameter sets on the tab page *Settings* of the debugger.

 [Testsystem/Debugger/Settings/Parameter sets](#)



30.5.3. Error message with the start – possible causes

-ver misses or is wrong specified

It is often forgotten to specify the version of the KIF that has to be started. Without the specification of a version (`-ver=w` is missing in call string), the KIF of the release version of the knowledge base that is specified with `-knb` is searched. If no corresponding KIF exists, the error message "KIF xxx cannot be opened! Error code 307 [...]" is displayed with the start of the WinClient.

With the check, if a KIF was generated from a release- or a work version, the WinClient only refers to the existence of the `w` (stands for work version) in the filename. I.e. if a KIF from a work version, e.g. `TestWB_1_0W45.kif`, is renamed to `TestWB.kif`, the version parameter has to be left out.



KIF is in a different KIF-directory

If you saved the KIF to a different directory than `C:\Program Files\camos\kif` (see practice 30.3), you have to inform the WinClient via the start parameter `-kifdir` in which directory the KIF is located. The directory path is transferred, e.g. `-kifdir="C:\Documents and Settings\Administrator\Desktop"`.

Start KIF via application server

If you work via an application server (you have no camosRunnerPE installed), you have to inform the WinClient which computer is the application server. The required start parameter is `-host`. The name of the application server is transferred, e.g. `-host=pc-training`.

The KIF that needs to be started has to be in this case in the KIF-directory of the application server – to access locally filed KIFs from the application server is not possible!



No license demanded

In order to execute the KIF a runtime license is required which depends on the type of the application. For configuration applications with rulework e.g. the license *camos.Configurator* is required. In the *CarConfigurator* the license was already demanded in chapter 17.2.2 in connection with the introduction to the rulework.

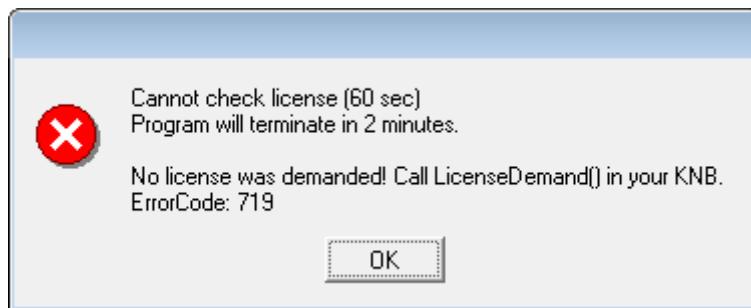


Figure 75: Starting an application without license

 Applications that do not use rules still need a license. If the license is not demanded (calling the function *LicenseDemand*), a corresponding message is displayed 60 seconds after the program start, see Figure 75. After further 120 seconds the application is automatically terminated.

30.6. Repetition

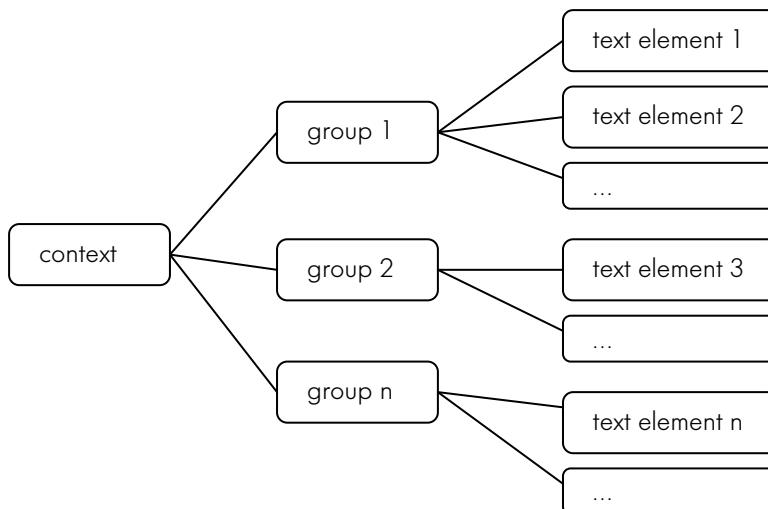
- KIF stands for **K**nowledge **I**nformation **F**ile and is needed to execute an application independent of the development system.
- In order to start a KIF, a link on the file *camoswinclient.exe* is created and the parameters *-knb=<NameOfKIF>* as well as *-ver=<VersionOfKIF>* are transferred.
- In order to execute the KIF, a runtime license is required, e.g. *camos.Configurator*. This license is demanded via the *camos Develop* function *LicenseDemand*.

31. Administration of texts in the Textmanager

In order to use and administrate texts with less effort, there is since version 6 of camos Develop the Textmanager. The Textmanager was fundamentally revised for version 7. Multilingual texts can be created and administrated for several work versions at once with the Textmanager. These texts can be used anywhere in the knowledge base.

Every knowledge base has its own stock of Textelements. The Textelements are always versioned with a knowledge base. This guarantees, that in case of changing a Textelement in the current work version, the Textelements, that are used in the previous version are not affected.

In the Textmanager texts are stored as so called Textelements. Textelements are always assigned to a context, contexts are divided into groups. Contexts are directly connected to a knowledge base. That is if a knowledge base is created, there is a context in the Textmanager with the same name. Within the knowledge base Textelements can only be used of the own context or the context of the integrated libraries.



A special case is the context *Public*. Textelements of this context can be used in every knowledge base (of the same database). Therefore all Textelements that can be used in other knowledge bases, should be stored in the context *Public*, for example “New”, “Erase”, “Insert”, “OK”.

The context Public is not versioned!



The Textmanager is integrated in the development system and can be called in many places in camos Develop. In general the Textmanager can be opened via the icon  in the toolbar or via the main menu *Window-> Textmanager ...*.

31.1. Textelements

Textelements are multilingual texts like constants. Unlike constants they are not created and administrated in classes but globally in the Textmanager. Textelements are not identified by a name, but by a unique GUID.

Textelements can be used in all places, where texts are stored, for example in the form editor, in the procedure editor or as a naming of classes or wasele. Textelements can be of the type String single-line String multi-line RTF or HTML .

For Textelements names don't have to be given unlike to constants and other Wasele. Textelements are identified by their GUID (globally unique identifier). This means that an assigned Textelement can be given another name ex post without losing its uniqueness.

Because the GUID provides no information for the text of the Textelement, the content of the Textelement in the current text display language (menu extras) instead of the GUID is displayed in the development system.

To identify, that a text is not a manual input, but comes from a Textelement, it is bordered with the character ° at the assignment, for example °Schulung°. For longer texts the display is limited to the first 100 characters. This is applied of course only for the development system; during the runtime the complete content of the Textelement is shown without °.

The following figure shows the GUID and the content of the Textelement. This dialogue is accessible via the context menu point *text Show information ...* of a Textelement in the Textmanager.

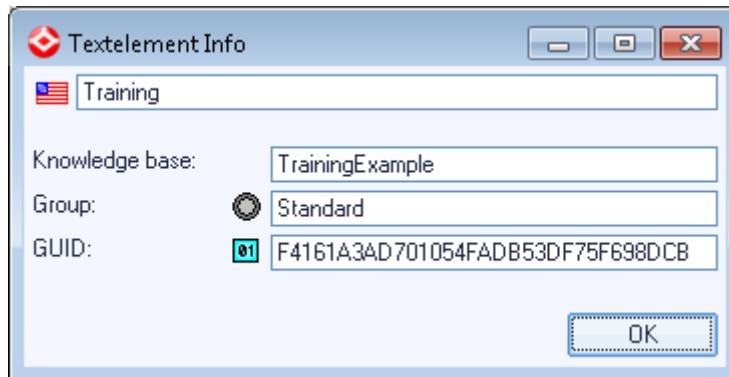
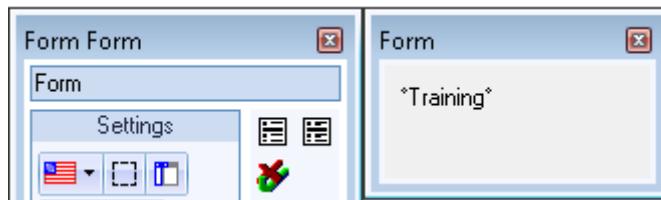
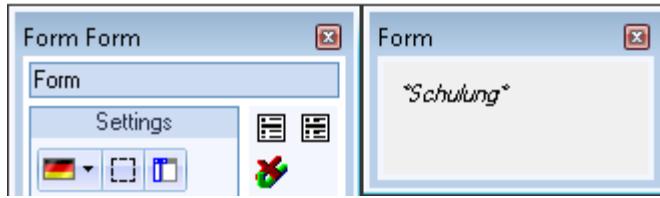


Figure 76: Textelement Info

When this Textelement is assigned in the form editor for example as a text of a static label, this looks like the following figure in the text display language English and German.





31.1.1. Practice: Creating Textelements

In chapter 8.5.2 you have become acquainted with the NOVALUE-naming that can replace the Ø-symbol in the configurationbox with a self-documenting text. In this practice you have stored at several components as the NOVALUE-naming *please select....*

Because identical texts, that occur multiply, imply more administration effort, it is more reasonable to define only one Textelement and use this instead of the redundant namings.

To use Textelements in your knowledge base, the *textinputmode* has to be activated. For that purpose open the knowledge base properties -> Basic data -> Options and reserve them. Activate the option *Textelement* under *textinputmode*.



To create a new Textelement, open the Textmanager via the icon  from the toolbar of camos Develop.



The window of the Textmanager is opened. In the left lower area the context of the knowledge base is selected (here *TrainingExample*). In addition you can see the knowledge-base-expanding context *Public* and the context *All*, in which is shown an overview from all contexts and groups.



To display the existing textelements, a filter expression has to be defined. When there are a lot of textelements, this approach is positive for the performance. For displaying all textelements, define * as filter expression.



In order to create textelements, the desired context is selected and the icon *New* from the toolbar of in the middle area is selected. Thereby the dialogue *Create Textelement* is opened. Here you determine the type of the Textelement, the context and the group. Because the context *Public* has been selected already and there is only the default-group Standard, all the specifications are properly initialised.

When creating a textelement, the text can be defined directly for all the languages existing in the frame.

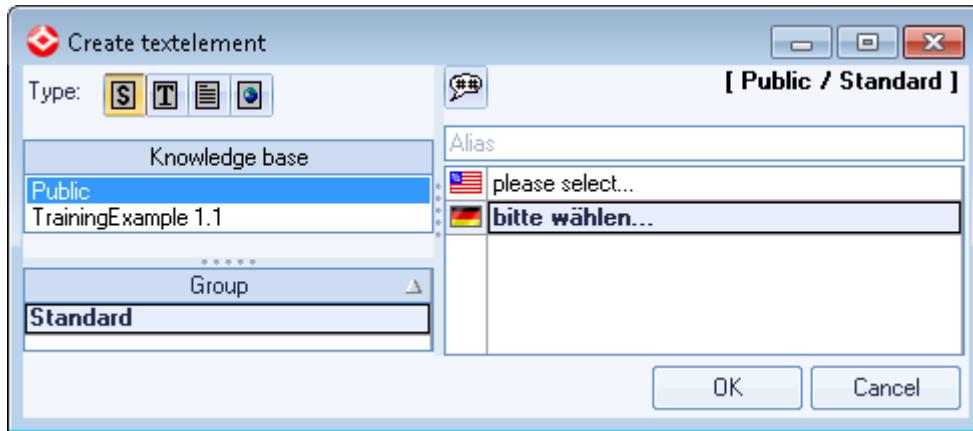


Figure 77: Create a textelement

The new Textelement is shown in the middle table, the focus is automatically set in the editline on the right side. Here the favoured text can be entered in all languages that are available in the frame.



If no text is entered in a Textelement that has been created just now, the Textelement is deleted when the focus changes. The same is applied to Textelements, whose content is deleted in all languages.

31.1.2. Textelements with included cause variables

Texts which should contain cause variables that are replaced by the current value during runtime, can be created as textelements aswell. An example are error messages, which should contain the not allowed value in addition to the error message.

As an example, the already existing message, that an invalid discount was entered (!ErrorDiscount in the class Car), is replaced by a textelement. In addition the value entered by the user should be displayed.

Because the message is knowledge base specific, the corresponding textelement should not be created as a Public textelement but directly in the context of the knowledge base.

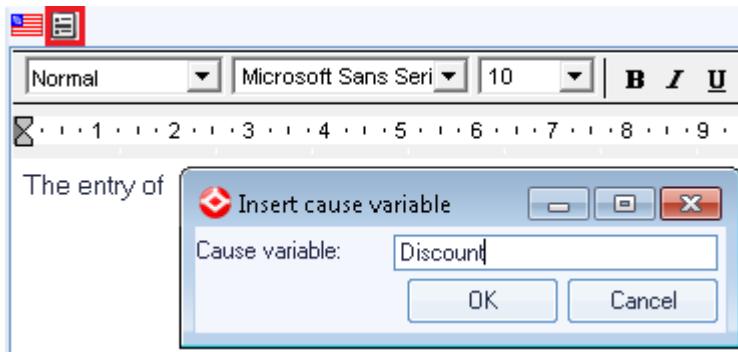


Open the textmanager and create a RTF textelement in the context *TrainingExample*. To do so, select the toolbar icon *New* in the middle area of the textmanager and enter the text "The entry of " in the upper left editor.

Now the entry of the cause variable should take place. Because textelements can be used in many classes, and in various knowledge bases if they are Public textelements, the cause variable can not be selected from a defined selection (Item selector) but has to entered manually.



Click the icon *Insert cause variable* above the editor of the textelement. Enter the text "Discount", because the feature *Discount* should be used here.



After the entry was confirmed with OK, the cause variable Discount is displayed with curly braces and as protected text, like you already know from the exercise 20.4.1 (Creating a quotation).

Add further text to complete the error message. The finished textelement should be: The entry of {Discount} Discount is not allowed! Please enter a value between 0% and 10%.

Enter the text and the cause variable in German and close the textmanager.



Next, the until now used String constant is replaced by the created textelement.

Because the textelement is a RTF textelement (because of the cause variable), this has to be converted into a String. This is implemented via the camos Develop function *RTF2String()*.

Open the class Car in therein the assing trigger of the feature Discount. There you find the call of the function WinMessage, which contains the constant *!ErrorDiscount* as text right now.



Delete the use of the constant and enter as second parameter "RTF2String(" instead. Then insert the recently created RTF textelement via the context menu *Select textelement*. Add a brace ")" to close the call.

The code of the assign trigger will now look like this:

```

IF not IsValid(Discount, Discount) THEN
    # Display error message with the content of the textelement:
    WinMessage('Error', Rtf2String("The entry of {Discount} Dis-
count is not allowed! Please enter a value between 0% and 10%.
"));
    Discount := LastValue;
ENDIF;
CalculateEndprice();
RETURN;
```

Carry out the Syntax check and test the application.

Delete the constant *!ErrorDiscount* which is not used anymore.



31.1.3. Textelements as possible values

Textelements are mostly used as namings. The content of a textelement can also be assigned to a String feature as value:

Str := °bitte wählen...°;

If the option *Multilingual* is activated for the String feature, all translations of the textelement will be delivered to the feature.

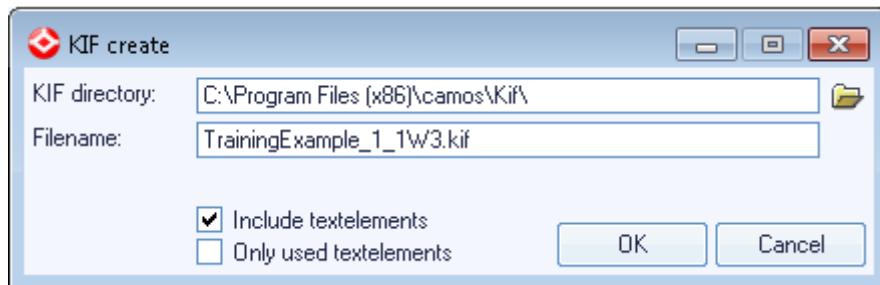
Please note that it is not possible to define a possible value of a feature via a textelement. However you can define a constant containing the textelement as value. This constant is assigned to the feature as possible value.



The textmanager displays only the data types which are allowed at this place. You know this behaviour from the Item selection dialog.

31.2. How to integrate Textelements into the KIF

During the creation-process of a KIF it has to be decided, if the textelements that are used in the knowledge base, should be integrated. By setting the option *Include textelements* in the *KIF create* - dialog the Textelements with the current state are integrated in the KIF. If this option is not set, the texts that come from textelements, are not in the KIF.



Beside the option of integrating the Textelements fixed in the KIF, they can be loaded during runtime dynamically from the database. In this case the option *Include textelements* can be deactivated in the *KIF create* – dialog.

The KIF has to be started via the parameter *-dsnTxE* with the ODBC-link to the data base of the knowledge base (for example *-dsnTxE=DevelopDB*). This procedure has the big advantage, that in case the content of a Textelement has been changed, not a new kif has to be distributed to all users.

Besides the option *Include textelements* you can decide if only the used textelements should be included. This reduces the size of the KIF, however the analysis while creating the KIF might take a bit longer.

?

Main menu/Knowledge base/Create KIF

?

Basics/Start parameters/Start parameter for camos RunnerPE/SE and for WinClient, WebClient and HTMLClient

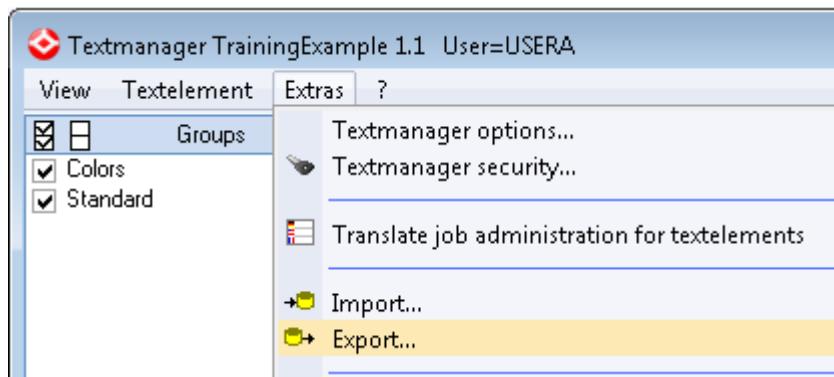
31.3. How to ex- and import Textelements

During a package-export (see chapter 29.6.2) the associated Textelements are automatically exported with the knowledge base in one file.

During a normal export it has to be specifically declared, that the Textelements should be exported. This is done by setting the option with the same name during the *Export knowledge base* - dialog. By setting this flag an additional file is created in the specified export directory with the file extension *.ctm during the export (see chapter 29.5).



To export only the Textelements of a knowledge base (without the classes), select in the Textmanager the main menu item *Extras* -> *Export...* .



In the dialog *Textmanager Export* the export can be limited, that means that you can decide, which contexts (= knowledge bases), groups and languages should be exported.

Textmanager/Main menu of the Textmanager

You can access the import via the main menu item *Extras* -> *Import ...* . Thereby you have to notice, that the name of the context is only initialised with default values, if the target-knowledge base has the same name as the original knowledge base. If the export includes for example Textelements from *Public* and from the context *TrainingExample* and you have opened a knowledge base with a different name, only the Textelements of the context *Public* of the source data base are imported in the context *Public* of the target data base without manual adjustment.

To import the Textelements into a knowledge base, that has a different name, the name of the target knowledge base has to be set during the import-dialog. For that purpose deselect in the left bottom in the column *Knowledge base* the row, in which is displayed ??? . Select from the

context menu one of the allowed contexts: *Public*, the new knowledge base and if necessary the assigned library knowledge bases.

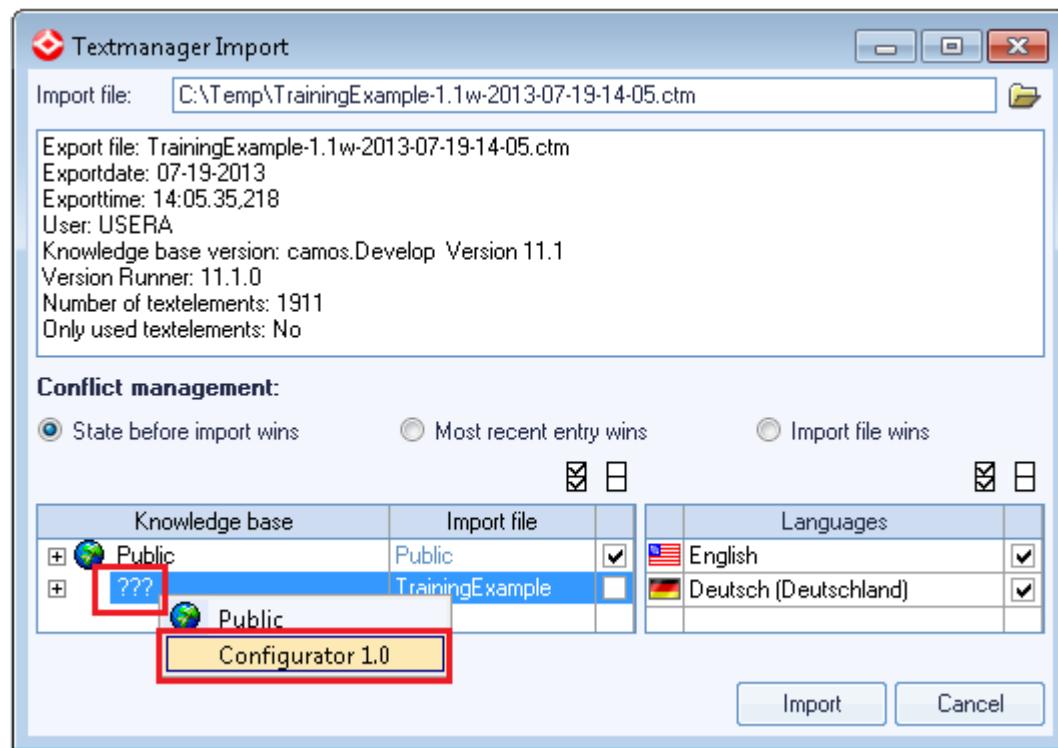
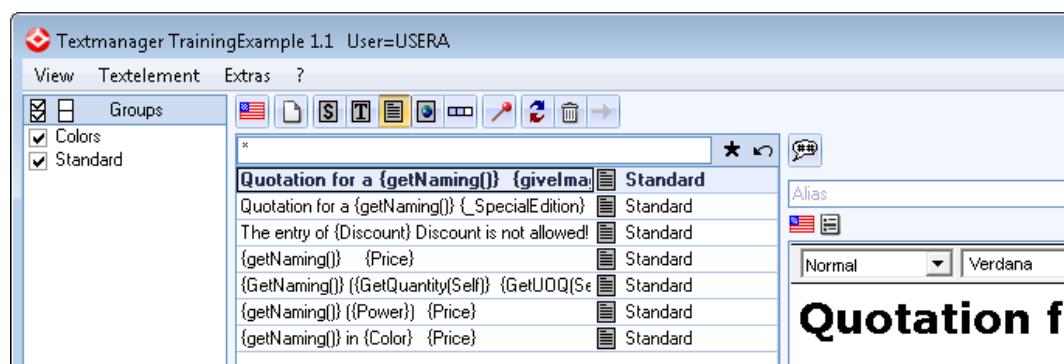


Figure 78: How to adjust the context name at the import of Textelements

31.4. Namings for textelements

The name of a textelement is, like you have already learned, the first 100 characters of the included text. Because it can be difficult to distinguish RTF textelements which are part of the quotation, shows the following screenshot.



You can see, that there are textelements, which begin with the same characters. When using long quotation texts it can happen that the textelements just differ after the 100th character.

For those thestelements there is the possibility to define an Alias optionally. The monolingual Alias serves as the naming and will be displayed in the textmanager instead of the name. The search will of course consider the defined Alias.

Define an Alias for the RTF textelements, e.g. "Quotation header Standard" and "Quotation header with Special Edition".



After creating the Alias the display in the textmanager could look like this:

Text Element	Type
Error message Discount	Standard
Quotation header Standard	Standard
Quotation header with Special Edition	Standard
Table column Standard	Standard
Table column with color	Standard
Table column with Power	Standard
Table column with quantity	Standard

31.5. Tips & Tricks

31.5.1. Save a filter

In order to see textelements in the textmanager, a filter expression has to be defined. To avoid setting the filter each time you open the textmanager, you can "pin" the filter via the toolbar icon . Saved is the selected context, the group, the textelement types and the defined filter expression.

If the textmanager is opened in a place where the filter does not fit, the filter is removed. E.g. the textmanager is opened in a RTF constant and the data type single line String is pinned, the filter is removed, so that RTF textelements are selectable.

31.5.2. How to replace multilanguage texts with Textelements

If you use the procedure that is described in the previous chapters, you can replace all the texts, that have been entered freely and constants of your knowledge base with Textelements. In case you want to convert all texts of an extensive knowledge base at once into Textelements, there is a possibility to do this via a tool that is integrated in camos Develop.

Via the main menu item *Extras -> Tools -> Replace multilanguage texts by textelements ...* you have access to the tool with the same name. The knowledge base or the selected class is scanned for free text input with this tool. A Textelement can be created and assigned during this dialog for all places of discovery.

You can find the exactly described procedure in the online-help:

[Main menu/Extras/Tools/Replace multilanguage texts by textelements...](#)

31.5.3. How to set the mode for new string constants

In the knowledge base properties you can find on the tab page *options* the section *mode for new string constants*. There you can define, what kind of input mode should be set automatically in new created String constants.

The choice is between *Textelement*, *Monolingual text* and *Multilanguage text*.

 [Knowledge base/Properties/Options](#)

31.6. Repetition

- Textelements are used like multilanguage constants in any place in the knowledge base as providers for naming and texts.
- Textelements are administrated in the textmanager.
- Textelements can be deposited expanding the knowledge base in the context *Public*.
- Textelements are clearly identified via the GUID, the content can change during the development without the Textelement losing its uniqueness.
- A naming for a textelement can be defined via an Alias.
- Textelements are bordered with the characters °° in the development system.
- Textelements display the current value during the runtime in the selected dialog language.
- Textelements are versioned with the knowledge base.
- The context Public is not versioned.

32. Common requirements for an application

32.1. Intention

The training target – the development of the car configurator – is completed by creating the KIF in chapter 30.

Most developers ask how the appearance and the usability of the configuration can be further improved. Some of these frequently requested changes are described in this chapter:

32.2 Navigation in the class tree

32.3 Dealing with the

32.4 Improving the user interface via tab pages

32.5 Alignment of form elements / form alignment

32.6 Display of the result on the form

32.7 Extend the result by address and creation date

32.8 Transparent graphics

32.9 Individual

32.10 Basic properties of form elements/abstract representation

Before you carry out changes, we recommend to create a new work version, e.g. 1.2w (see chapter 24.4), in order to secure the current condition of the knowledge base.



32.2. Navigation in the class tree

Via the key * (multiplication) on the numeric keypad a class- or structure tree can be completely expanded. This can also be used with the form elements *tree* and *component tree*.

Via the button + (plus) on the numeric keypad only the next level is opened, the cursor button right has the same effect. With – (minus) or the cursor button left this level is closed again.

If the focus is on a tree element, the navigation in the tree is possible via the cursor keys up and down.

32.3. Dealing with the Recycle Bin

In camos Develop the menu item *Recycle Bin* is in the context menu of (almost) every element. The Recycler is indicated by the icon .

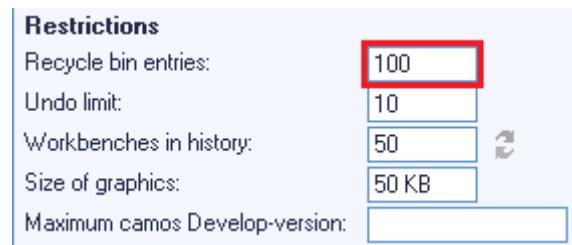
If an element is deleted, it is collected in the *Recycle Bin* of the superior element.

There is a global *Recycle Bin* in the class tree. This means, no matter where in the class tree you call the *Recycle Bin*, all deleted classes are displayed and can be restored. This is possible via the option *Show all deleted classes of the knowledge base*.

Recycle bin of Convertible				
	Name	Parent class	Date	User
1	Leather	InteriorDecoration	07-23-2013 16:08	USERA
2	Passat	Hardtop	07-23-2013 16:08	USERA
3	Radio	Accessories	07-23-2013 16:08	USERA

Please consider that there is no global Recycler for knowledge base elements, so you can restore deleted elements only in the Recycler of the accompanying group/base class/class etc. Therefore you have to remember at which place you delete e.g. a feature.

The number of the Recycle Bin entries can be determined in the *Properties of the knowledge base* on the tab page *Options*. By default the quantity is set to 100.



If the Recycler is deactivated at the place from where it has to be called, it does not contain any elements. Via the icon the Recycler can be emptied and therefore deactivated. All elements that were contained in it are irretrievably deleted.

Recycler entries can be restored via the icon .

Recycle Bin			
	Name	Date	User
1	TempHnd	23.07.2013 16:13	USERA

All entries of a Recycler in the structure tree and the wasele list are definitely deleted with releasing or cancelling the class.

The Recycler entries in the class tree are not automatically deleted with terminating camos Develop, releasing a class or generating a release version.

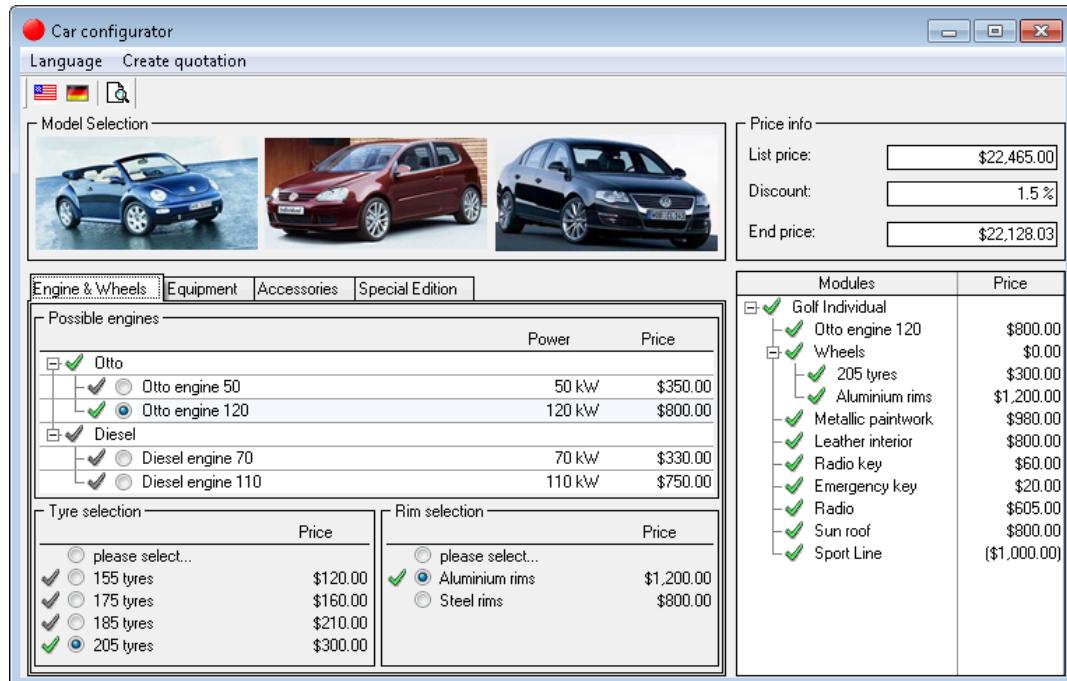
The deleting of classes has to be confirmed in general. For wasele this has to be activated manually via the option *Ask for deleting a wasele* in the user options.

32.4. Improving the user interface via tab pages

Due to the numerous form elements to select and display the configuration, the form appears very unstructured. It would be better to part the components in separate categories to handle them separately.

A form element that is used for most of the configurations is the tab frame. Tab frames consist of a number of tab pages that are separately placed in the foreground to show and edit their contents.

In this chapter the user interface of the application should be optimized as follows.



32.4.1. Practice: Division into tab pages

The configuration can e.g. be subdivided into the groups *Engine & Wheels*, *Equipment*, *Accessories*, *Transport accessories* and *Special Edition*.

Most of the form elements that are currently on the *MainForm*, will be placed on the *DetailForm* in the class *Car*. After the general changes for all models in the base class are carried out - the model-specific components are reinserted in the individual object classes.

On the *MainForm* only the model selection, the price- and discount display and the component tree should be kept. All further parts are shifted as follows. Please consider that the subform still has to be displayed next to the component tree.

Open the *MainForm* and enlarge the dynamic subform until it approximately corresponds to the size of the completed tab. Orientate to the figure of the finished condition at the beginning of the chapter.

Delete the contents of the *DetailForm* in *Car* and allocate the same size as with the *dynamic sub-form*. Create a tab frame and under this a tab page. Adjust the height and width to the form size.





 Create a new textelement with the content “Engine & Wheels” and deposit it as text for the first tab page and set the same name. Create two further tab pages for “Equipment” and “Accessories” the same way and set text and naming.

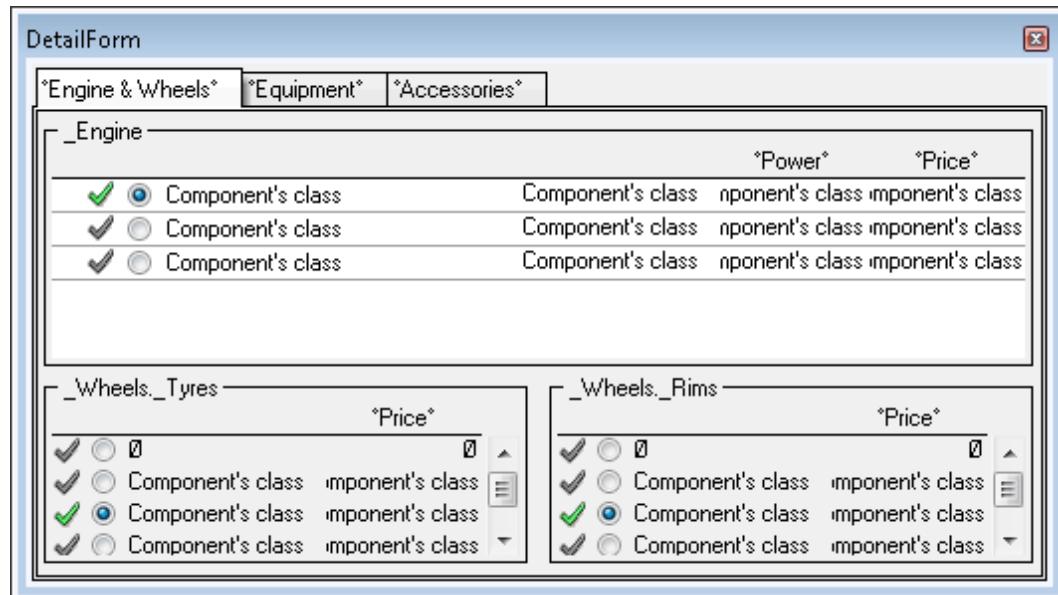


 Cut out the configboxes for the engine, the tyres and rims in the class *start* on the *MainForm*. Insert them in the class *Car* in the *DetailForm* under the tab page *Engine & Wheels*. Open the form preview and arrange the elements and adjust the sizes, if necessary.

 Please consider that the cause variables are no longer valid, because the form elements were shifted to a different class. So the path “_Car” has to be removed on all cause variables (e.g. “_Car._Engine” now becomes “_Engine”).

 Change the path of each cause variable.

With this the configuration boxes are visible again in the preview:



 Cut out the form elements for the *Paintwork*, *Paintwork color* and *Interior fittings* from the *MainForm* and insert them in the *DetailForm* under the tab page *Equipment*.

Adapt all cause variables and arrange the elements in the preview.

Repeat this procedure for the *Keys* and the *Accessories*. Locate them on the tab page *Accessories* and set the correct cause variables.



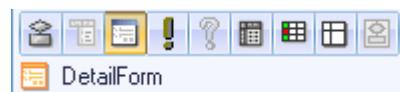
Start your application and check the display and the alignment of the form elements.

Since the *DetailForm* will be overloaded in the object classes in the next step, an additional change in all *Car* classes would be correspondingly laborious.

The *DetailForm* was overloaded in the classes *Golf* and *Passat*. All changes we made in the class *Car* were therefore not further inherited. In order to make the changes visible in the object classes, the overload is taken back.

Attention: With this all model-specific components will be lost. Either you copy the form elements first or you have to create them again.

DetailForm overloaded



Via the context menu item *Delete* or the key *Del* the overload is taken back and the inherited condition is restored.

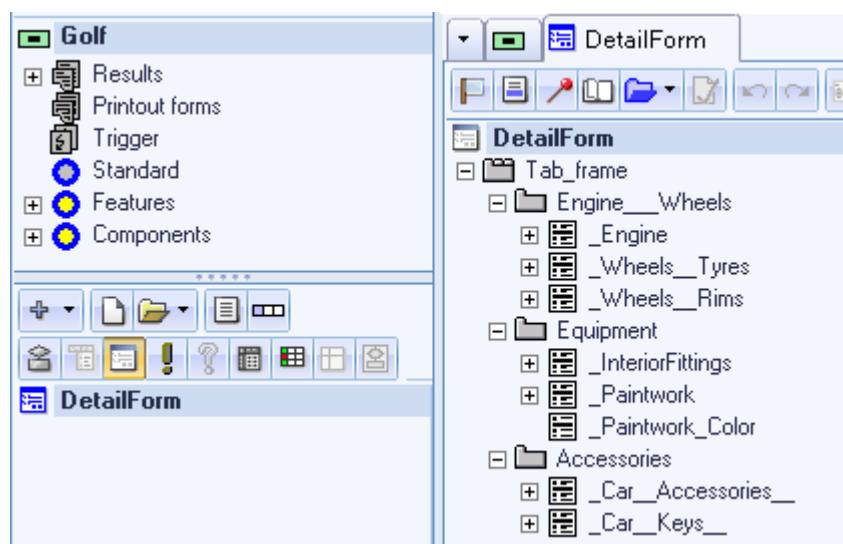


The form icon changes the color from orange to blue.

Open the editor of the form *DetailForm*.



You will notice that all form elements from *Car* are present.

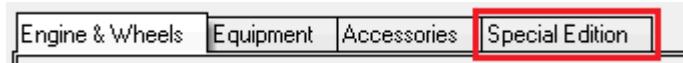


The configbox for the *SpecialEdition* of the *Golf* should be displayed on a separate tab page.



Overload the *DetailForm* again. Create a new tab page with the name “Special Edition” and then you create a configurationbox of the component *_SpecialEdition*.

Test your application. With the model selection of the *Golf* the additional tab page is faded in.



Carry out this adaptation analog in the class *Passat* for the *Transport accessories*.

Test your application again.



32.4.1.1. Display the validity symbol on tab pages

Via the introduction of the rule processing in chapter 17 the current state of the configuration can be checked for validity. A rule violation is displayed in the component tree with a validity symbol on the respective component.

Through the introduction of the tab pages it is no longer obvious at the first sight on which tab page an invalid configuration is originated. Because of this the option *Validity symbol* is available for the form element *Tab page*. In order to switch this, you open the *Tab frame* on the *DetailForm*. On the tab pages under the tab frame this option can be set within the group *Representation*.

If you set the option *Validity symbol* you can additionally define if all icons or only *MayNot*-icons should be displayed.

Name		Engine & Wheels
Type		Tab page
Help		
Stylesheet		
+ Geometry		
- Representation		
Visible	<input checked="" type="checkbox"/>	
FG color		COLOR_SYSTEM
BG color		COLOR_SYSTEM
Open page fg color		COLOR_SYSTEM
Open page bg color		COLOR_SYSTEM
Frame		No border
Graphic		
Validity symbol	<input checked="" type="checkbox"/>	
Show MayNot symbol only	<input type="checkbox"/>	

Effect in the interpreter:**32.4.1.2. Options of the tab frame**

Some settings can also be made on the tab frame itself. E.g. the tab pages can be displayed on the bottom side of the tab via setting the option *Tabs down*.

Representation	
Visible	<input checked="" type="checkbox"/>
Font	<input checked="" type="checkbox"/> Standard
Frame	<input type="checkbox"/> Line border
Border color	<input checked="" type="checkbox"/> (0,0,0)
Tab style	<input checked="" type="checkbox"/> Standard
Tabs down	<input checked="" type="checkbox"/>
Multiline	<input type="checkbox"/>

In the interpreter this option would have the following effect:



Via the option *Multiline* the tab pages can also be displayed in several rows.



Set option *Multiline* in the interpreter:



Otherwise a scrollbar is faded in.

Not set option *Multiline* in the interpreter:

**32.5. Alignment of form elements / form alignment**

As you might have noticed there are further properties for the form elements that were not yet mentioned. For most form elements their alignment can be set in four different directions. This

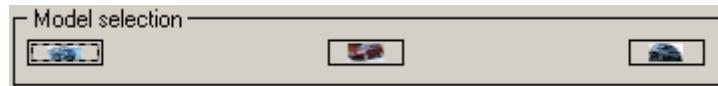
property can be set under the group *Geometry* in the editor of the form element or via the graphic of the base properties next to the form element preview.

The easiest way to illustrate the meaning of this property is on the configurator application.



Start the debugger and drag the form smaller and bigger.

With the groupbox of the model selection (e.g.) the following effect occurs with the reduction:



...and this with the enlargement of the form:



How a form element has to behave when changing the size of a form or the higher-order form element is determined by the option *Alignment*. The set alignment can be changed via selecting the alignment icon.

For the sides (in the following called “Panel side”) of the form element (left, right, top, bottom) the different settings have the following effect:

Alignment left

- ◀ The distance to the left panel side (X-position) remains fix.
- ✚ The distance to the left and right panel side changes proportionally.
- ▶ The distance to the right panel side remains fix.

Alignment right

- ◀ The distance to the left panel side (X-position + width) remains fix.
- ✚ The distance to the left and right panel side changes proportionally.
- ▶ The distance to the right panel side remains fix.

Alignment top

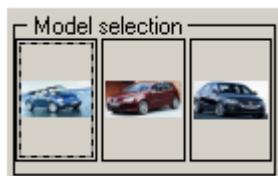
- ↑ The distance to the upper panel side (Y-position) remains fix.
- ↔ The distance to the upper and lower panel side changes proportionally.
- ↓ The distance to the lower panel side remains fix.

Alignment bottom

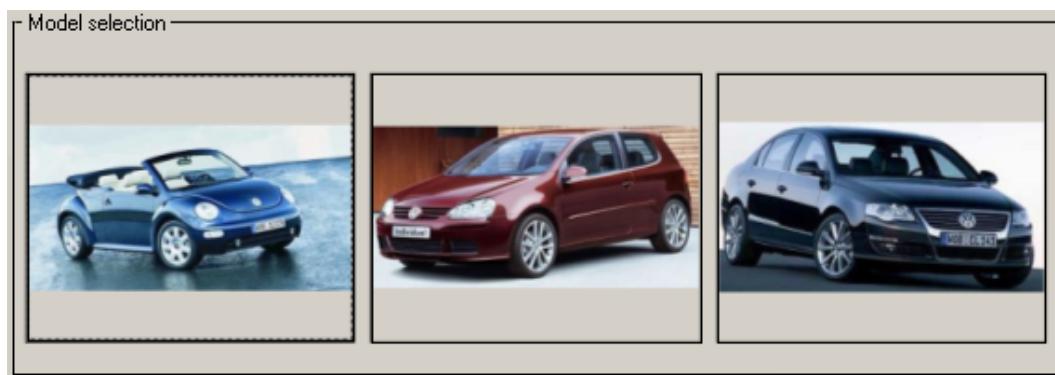
- ↑ The distance to the upper panel side (Y-position + height) remains fix.
- ↔ The distance to the upper and lower panel side changes proportionally.
- ↓ The distance to the lower panel side remains fix.

Try out different settings in their effect.

By setting the alignment to proportional changes on all four sides the following happens e.g. with reducing the *MainForm*:



and with the enlargement this effect can be observed:



This is in any case already an improvement to the default setting.

Think about which behavior is the most useful for the groupbox and adjust also the alignments of the other form elements (price groupbox, configboxes).



32.5.1. Tips & Tricks

32.5.1.1. Search for alignment

Via the search the knowledge base can be browsed for properties e.g. of features, components, forms and constants. To find for example all form elements, that have the alignment “Right” for the left border, select the menu item *Find* from the main menu *Edit* and the search options have to be set correspondingly:

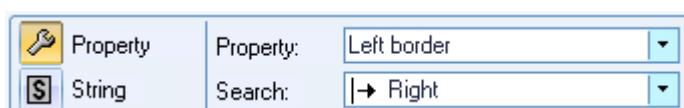


Figure 79: Search dialog – Property “Left border”

The search result can be refined by searching in the search result. To do so another Search tab page is faded in by the icon .

[Basics/Search](#)

32.5.1.2. Forbid size adjustment

In order to give a user not even the possibility to reduce the form beyond recognition, the setting *Resizable* in the form editor can be switched. For the property *Resizable* there are in addition to the default setting *Resizable* also the options *Enlargeable* (form can be dragged bigger, but not smaller) and *Not resizable* (form size cannot be changed).



32.6. Display of the result on the form

Instead of displaying the generated result in a separate window, you have the possibility to do this in the form itself. For this purpose the form element *Result* is available.

Results that are directly placed on the form provide information about the configured product immediately, even before the selection procedure is concluded.

If the result should be displayed on the *MainForm*, then the form can be enlarged and the form element can be brought onto the form.

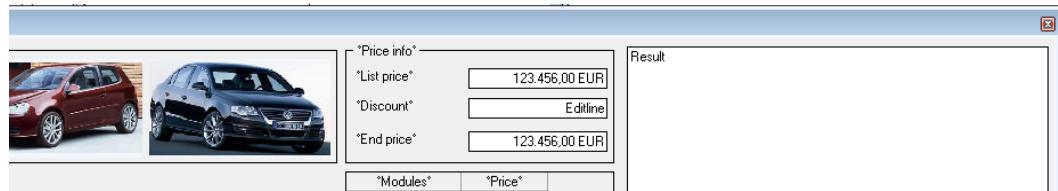
In our case there is a further possibility and that is to display the result on an own tab page. If you already subdivided the form into tab pages, this change would have to be reproduced manually in all car classes that overload the *DetailForm*.

Since the immediate display during the configuration should be demonstrated, we take a closer look at the first possibility.

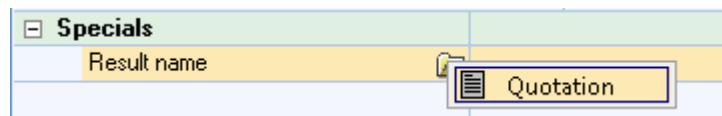


To do so, drag the *MainForm* in the width to get sufficient space for the Output to result/printout form.

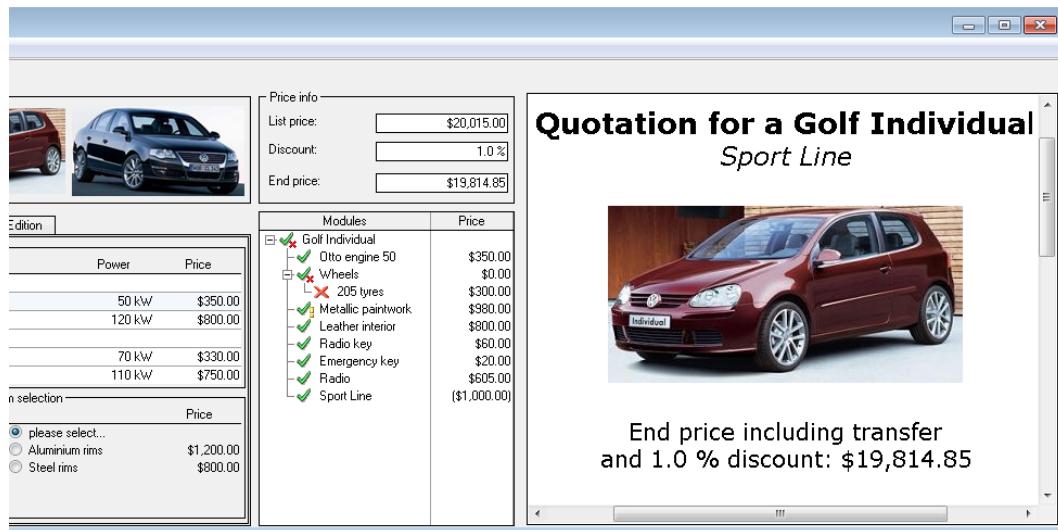
Create a form element *Result* via the context menu of the form root or the form preview and adjust the size.



Now the result name is deposited in the editor of the result. Via the icon a list with all existing results is faded in from which the desired result can be selected.



Choose *Quotation* in the field *Result name* and test the application.



You see that this change can also bring disadvantages. On the one hand the result is displayed even with an invalid configuration. On the other hand the form has to be extremely enlarged in order to get a reasonable output.

A further important point of criticism is the bad performance of the application, because the result is automatically updated after each value change. Therefore the use of the form element *Result* is principally not recommendable.

Delete the form element *Result* and adjust the size of the form.



32.7. Extend the result by address and creation date

The result should be extended by a footer that contains the name and the address of the dealer as well as the current date.

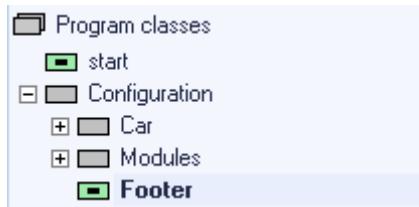
For this it is not possible to insert a corresponding text module in an existing class. If this text module would be added to e.g. the class *Car*, it would be faded in before or after the text module *ResultHeader*. A further text module in the class *Modules* would be added in front of or behind each component.

Therefore an auxiliary class is created. This class is not involved in the configuration of the product itself; it has just the task to provide a text module. This class is applied as component in *Car* and therefore treated as car component.

Since the auxiliary class has no features and components at all, only a RTF-constant, it is not displayed on the form for selection and it has no influence on the price calculation.



Create an object class with the name “Footer” under *Configuration* in order to inherit the result allocation.



Create a RTF-constant with the name “!Footer” in the class *Footer* and enter a fictitious dealer address in the editor.

If you enter some blanks in front of the address, the text is displayed with a certain distance to the component table.

Via the function *GetDate()* the current date can be read out.



Insert the function call *GetDate()* via the Item selector dialog as a cause variable in *!Footer*.

?

Function reference/Date Functions/GetDate ([Time])

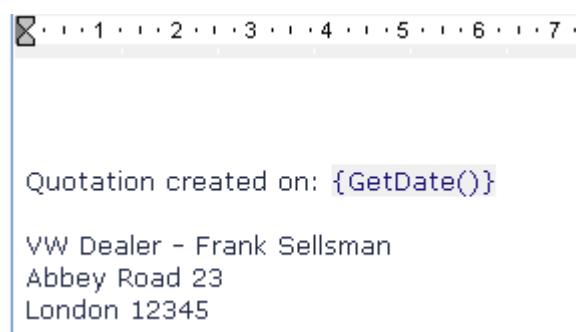


Figure 80: Show date and dealer in the footer

Convert the defined text into a textelement via click on the icon  in the toolbar of the constant editor.

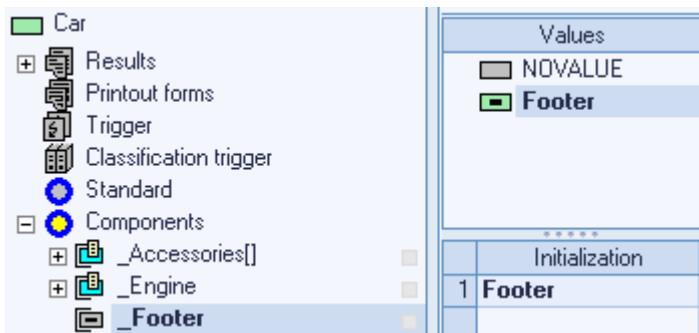


In order to integrate this text module in the result, you insert the constant *!Footer* as a text module under the inherited result *Quotation*.



In the next step a component of *Footer* is created in the class *Car*.

To do so, you drag and drop the class *Footer* under the group *Components* in the class *Car*. Initialize the component with itself (*Footer*) in order to automatically generate an object of the auxiliary class with the creation of a car.



Activate the option *Output to result/printout form* on the component *_Footer*.

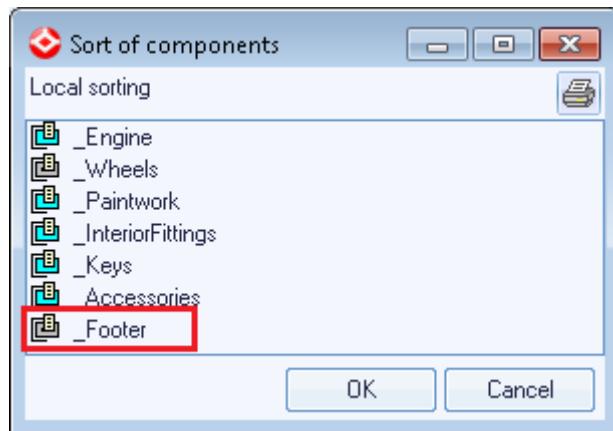


Since the order of the texts in the result is based on the order of the objects in the object tree, it has to be ensured that the component *_Footer* is at the last position in the object tree.

The position of an object in the object tree can be determined via the dialog *Sort of components* (see chapter 13.4).

Set the option *Sort* on the component *_Footer* and change the component sorting of the class *Car* (icon  in the class editor) according to the following figure.





 Check also in all from *Car* derived classes if a component sorting exists (icon  is black) and shift the component *_Footer* to the last position. If the sorting of components is inherited, the icon is blue. To overload the inherited sort, set the property Sort of Components for this class.

Start the application and check the output and correct position of the footer in the result.

The screenshot shows a quotation document with the following content:
Quotation created on: 7/24/2013
VW Dealer - Frank Sellsman
Abbey Road 23
London 12345

The current date is inserted in the result and the address data is displayed under the result table.

 For the quotation is displayed continuously in German, translate the content of the constant *!Footer* respectively the content of the textelement also into German.

32.7.1. Fade out objects from the component tree

The object of the component `_Footer` is now displayed in the component tree, too:

Modules	Price
[-] New Beetle Cabriolet	
[+] Soft top	\$0.00
[+] Otto engine 120	\$800.00
[-] Wheels	\$0.00
[+] 175 tyres	\$160.00
[+] Aluminium rims	\$1,200.00
[+] Radio key	\$60.00
[+] Emergency key	\$20.00
[+] Footer	

In order to fade out certain objects, e.g. only internally needed auxiliary classes, from the display in the component tree, there are several possibilities.

In the form editor of the form element component tree e.g. a *Filter class* can be specified under the group *Specials*. If a filter class is defined, only the objects that are derived from this class are displayed.

Since `Footer` is derived from `Configuration`, `Configuration` cannot serve as filter class. The specification of the class `Modules` is also not possible, because otherwise the car model (not derived from `Modules`) would not be displayed any longer.

A further possibility is the definition of a *Filter expression* under the group *Specials* in the form editor of the component tree. For each object as from the entered *Root* this expression is automatically evaluated and the result is checked: if the expression returns the value 1, the object is displayed, otherwise not.

This can e.g. be solved via a feature that has by default the value 1 (object is displayed). For all objects that have to be faded out, this feature is overloaded and set to 0.

In the class `Configuration` you create the numerical feature “visible” and initialize it with the static value 1.



With this the car- and module objects are by default visible.

In order to fade out the object of `Footer` in the component tree, the init value of the feature `visible` in the class `Footer` is overloaded and set to 0.



In the component tree the expression that was deposited in the column *Filter expression* controls if an object is displayed or not.

Only objects should be displayed in the object tree, whose feature `visible` has the value 1. This condition can be formulated with `visible =1` or shorter `visible`.



Through this only the objects are displayed that have the feature *visible* with the value 1.

Specials	
Root	<input type="checkbox"/> _Car
Filter class	<input type="checkbox"/>
Filter expression	<input type="checkbox"/> visible

The expression *visible = 0 or not visible* would display all the objects, whose feature *visible* has a value unlike 1. Attention: if the father nodes are hidden via the *filter expression*, the child nodes also can't be displayed! In the training example the component tree would be empty for *not visible*, because the root object *Car* has an *visible* value of 1.

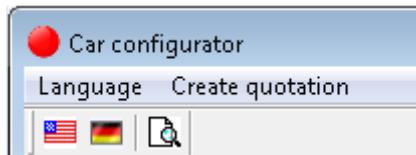


Test if the component tree is correctly displayed in your application.

If required, use the same procedure to fade out the objects for soft top and special model.

32.8. Transparent graphics

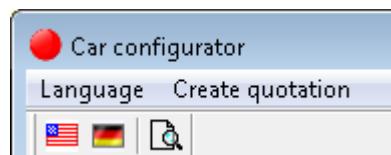
Let us have a look at the icons in the toolbar. They are surrounded by a white border in order to keep the recommended size for icons of 16x16 pixels



In order to "remove" the white area, the color white can be defined as transparent in the graphic constant editor.



To do so, you open the editor of the graphic constant *!IconQuotation*. Select the tool pipette  and click with it in the white area of the graphic. Then you set the current color transparent in clicking the icon  from the dropdown menu on the right side. Now all white pixels are displayed transparent, i.e. the background shines through.



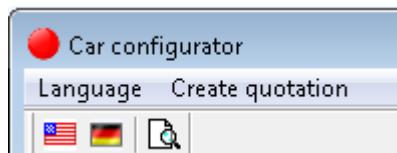
Since the white sheet on the icon still has to be white, the area that actually has to be transparent has to be filled with a different color that is used nowhere else in the graphic. This filling color is then defined as transparent.

To do so, you select e.g. the color pink  from the section *Colors*. With the tool filling bucket  you afterwards color the area that has to be displayed as transparent. For a better handling you can enlarge the graphic via the icon  from the toolbar or the context menu of the graphic.



Figure 81: Transparent graphics

Now only the new color has to be defined as transparent color and the result is as desired. Define a matching transparent color for the other icons as well.



Always just one color can be defined as transparent. You can see which color is transparent in the second colored box in the upper toolbar of the graphic constant editor. Via the X-button on the right next to it you can delete the transparency.



32.9. Individual title bar

When you start your application, the configurator icon (red dot) is by default displayed in the title bar. The icon can be changed via calling the function `SystemSet("DefaultIcon", Graphic)` for all forms of the application or via `WinSetGraphic(Form handle, Form element, Graphic)` for a certain form.



Function reference/Dialog Functions/WinSetGraphic (Form handle, Form element, Graphic)

Function reference/System Functions/SystemSet/SystemSet ("DefaultIcon", Graphic)

In the title bar a different icon, e.g. your company logo, has to be displayed. Create a new graphic constant with the name `!OptionalProgramIcon` in the class start. Load any 16x16 pixel sized graphic.

The function to set the icon is called in the method New.

```
LicenceDemand("camos.Configurator");
winhandle := WinOpen(' MainForm');
SystemSet('DefaultIcon', !OptionalProgramIcon);
RETURN;
```



The icon is set via the function `SystemSet(...)` on the `MainForm` as well as on the result window (and all further windows).

As you can see, there is a text (name of the result) automatically displayed in the title bar for the result window but not for the `MainForm`. The form title can be deposited in the editor of the form (form root) in the field `Header`.

Name	MainForm
Type	Form
Help	
Stylesheet	
+ Geometry	
+ Representation	
- Handling	
Readonly	<input type="checkbox"/>
In taskbar	<input checked="" type="checkbox"/>
Menu	 Administration
Toolbar	<input checked="" type="checkbox"/>
Large buttons	<input type="checkbox"/>
ViewShot relevant	<input checked="" type="checkbox"/>
- Texts	
Header	 "car configurator"



The naming that is entered in the editor is displayed on the *MainForm*. Additionally the header, e.g. the selected model, the username etc., can be dynamically set via the camos Develop function *WinSetTitle*.

 [Function reference/Dialog Functions/WinSetTitle \(Form handle, Title\)](#)

Via removing the option *Title bar* in the properties of the form root, you can fade out the title bar completely.



Representation	
Close on focus lost	<input type="checkbox"/>
BG color	 COLOR_SYSTEM
Window frame	 Standard
Window border color	 COLOR_SYSTEM
Title bar	<input checked="" type="checkbox"/>
Client type	 Standard

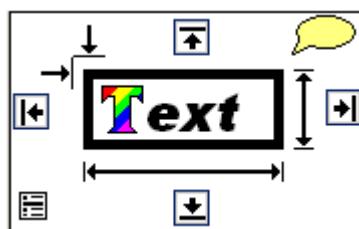
In this case a Close icon (X) does no longer exist, i.e. you have to create a possibility to close the window manually; e.g. a menu item “Terminate” that calls the function EXIT.



32.10. Basic properties of form elements/abstract representation

Above the property table of the form editor is a splitter page that displays the abstract view and the single preview of the currently selected form element. Via the icon  a *Scaled preview* of the complete form can be displayed. This is useful for the orientation, where each form element is located on the form. But the scaled preview is read-only, this means form elements can only be created or shifted in the normal form preview.

The abstract view consists of the following graphic:



Via a click on different elements of the graphic, settings for a form element can be directly edited. If selection possibilities exist, they are displayed.

You have setting possibilities for the following properties:

Alignment

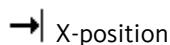
Via a click on the icon a popup menu appears via which the desired alignment (left, top, bottom, right) can be set.

Height

A click on this icon jumps in the line *Height* of the property table.



A click on this icon jumps in the line *Width* of the property table.



A click on this icon jumps in the line *X-position* of the property table.



A click on this icon jumps in the line *Y-position* of the property table.

Foreground color

Via a click on this icon a popup menu appears. This menu contains all user-defined colors that are available in the frame as well as the system colors.

Font

Via a click on the icon a popup menu appears with all fontaliases that are available in the frame.

Background color

Via a click on the icon a popup menu appears with all user-defined colors that are available in the frame as well as the system colors.

Border

Via a click on the icon appears a popup menu from which the desired border type (no border, 3D-border or line border) can be selected.

Tooltip

A click on this icon jumps in the line *Tooltip* of the properties table.

 Cause variable

On the bottom left side of the abstract view is the icon for the Item selector dialog with which the cause variable that has to be assigned to the form element can be selected. Alternatively the cause variable can also be dragged and dropped on the icon.

Depending on the form element the name of the property *Cause variable* differs. On the form element Graphic the corresponding field is e.g. called *Graphic*, on tables *Selected*, on subforms *Form name*, etc. The icon  is valid for the bright-yellow shaded line in the properties table.

Note: The display of the basic properties is different depending on the form element. This means that certain settings are only possible with some form elements. That is the reason why the presentation of the graphic changes as well.

 Workbench/Wasele/Form/Form editor/Element editor/Preview modes/Base Properties / Abstract

33. Hotkeys and Shortcuts

Many functions in camos Develop can in addition to the context menu or an icon also be called via hotkeys or shortcuts.

33.1. Hotkeys in the Class Tree

Ctrl+S	Syntax check
Ctrl+I	Integrity check
Ctrl+R	Reserve (also in form editor)
Ctrl+E	Release (also in form editor)
Ctrl+F	Search dialog
Ctrl+C	Copy
Ctrl+X	Cut
Ctrl+V	Insert
Ctrl+N	Rename
Ctrl+L	LookUp
F3	Continue search (after LookUp)
Alt+1	Restores the smaller width of the class tree after the width has been changed
Alt+2	Restores the wider width of the class tree after the width has been changed

33.2. Hotkeys in wasele lists, groups and classes

Ins	New (not with groups in the structure tree)
Del	Delete
Ctrl+O	Overload
Return	Open

33.3. Hotkeys in the procedure editor

Ctrl + Cursor key down	If you are in the parameter- or variable table, this hotkey opens the list of the datatypes for the selected parameter/variable. The individual datatypes can also be selected via hotkey, each with the first letter, e.g. D for date or S for string.
Ctrl+H	Search and replace
Ctrl+Z	Undo

Ctrl+Y	Restore
Ctrl+A	Select all
Ctrl+Blank	Codecompleter (suggestion list of all possible functions, methods, parameter, features etc. will be opened)
Ctrl+W	Insert a cause variable via the Item selector dialog
Ctrl+G	Extend cause variable, i.e. if the cursor is behind a component or an object pointer, then this is considered as the beginning of a path. The path is extended via the cause variable that is selected in the Item selector dialog.
Ctrl+E	Search bracket
Ctrl+M	Complete keyword (displays the list of the functions, beginning with the previous entered character string)
Ctrl+K	Change lines/paragraphs into comments
Ctrl+2	Encloses the expression in which the cursor is located with simple inverted commas
F1	Opens the camos Develop online help. If the cursor is on a camos Develop function, the function reference of the corresponding function is opened.
F4	Add expression to the watchlist (only in debug mode)
F9	Set/Delete a breakpoint of the line in which the cursor is located
F12	Jump to the method/class/constant in which the cursor is located
Alt+T	Insert a textelement (the textmanager is opened)
Ctrl+F2	Set / delete bookmark
F2	Jumps into the next line with a set bookmark
Shift+F2	Jumps into the previous line a set bookmark

33.4. Hotkeys in the decision table

Ctrl+Cursor key up	new line above
Ctrl+Cursor key down	new line below
Del	delete line (focus on numbering column)
Ctrl+Cursor key right	new column right
Ctrl+Cursor key left	new column left
Ctrl+Del	delete column

33.5. Hotkeys in the debugger

F5	Start
F10	Step Over
F11	Step In
Ctrl+F11	Step Out
Ctrl+F5	Restart

33.6. Hotkeys in the RTF Editor

Ctrl+I	Font style italic
Ctrl+B	Font style bold
Ctrl+U	Font style underlined

33.7. Navigation in the class tree, tree and component tree

Cursor keys up/down	marks the next class in direction of key
+ / Cursor key right	opens the next tree level
- / Cursor key left	closes the subtree up to the marked class
* (multiplication character)	opens the complete structure as from the marked class
Home	jumps to the first node
End	jumps to the last node

33.8. Navigation in the rule editor

The elements in the condition graph can be moved via the cursor keys:

Cursor key	Focus changes to the next element in the direction of the key
Ctrl+Cursor key	Slides the selected element 10 px. into the direction of the key
Alt+Cursor key	Slides the selected element 1 px. into the direction of the key

If the focus is in an editor, e.g. when editing a condition element expression none of the above mentioned hotkeys is triggered.

33.9. Navigation in the form preview

Cursor key	The focus changes (in arrow direction) to the next element in the form tree
Ctrl+Cursor key	Moves the element by the number of pixel that was specified in the toolbar (in arrow direction)
Alt+Cursor key	Moves the element by 1 pixel (in arrow direction)

33.10. Other hotkeys

Ctrl+Alt+I	completes the list feature in the cause variable of a table column by the term 'Index'
F7	activates/deactivates the spell check in text- and RTF-editors
Ctrl+Shift+Alt+F2	to display the statement in database tables
F1	opens context-related the camos Develop online help
Shift+F1	adds a question mark to the mouse pointer. A click on a form element will display its linked help text (provided that a link exists and the camosDocViewer was integrated in the knowledge base).

33.11. Identify form elements

In applications in which forms exist in a nested or complicated class structure, a certain form element can be selected in the running application and searched in the knowledge base. The identification is achieved through the following key combinations.

- the focus is set in the corresponding form element (in the application)
- Ctrl + Alt + J
- change to the development system
- Alt +J in the development system
- the corresponding form is opened and the form element is selected

Additionally/alternatively Alt + W can be pressed, to jump directly to the cause variable of the form element.

33.12. Copy text from a WinMessage

The key combination Ctrl+C can be used for all WinMessages to copy their content into the clipboard.

34. Index of Figures

Figure 1: Navigation and access administration in the frame	19
Figure 2: Creating a knowledge base	20
Figure 3: Surface of the camos Develop development environment.....	21
Figure 4: Dealing with the camos Develop online help.....	22
Figure 5: Adding knowledge base to project.....	25
Figure 6: Active ticket in the ticket system (TicketXS).....	25
Figure 7: Create start-class.....	29
Figure 8: Form Editor – Edit of a static label	32
Figure 9: Syntax check in the procedure editor.....	35
Figure 10: Class structure of the car.....	51
Figure 11: Creating a new component.....	55
Figure 12: Item selector dialog.....	57
Figure 13: Apply values to the structure tree	58
Figure 14: Defining init values	66
Figure 15: Graphic constant editor	72
Figure 16: Creating new dialog language	80
Figure 17: Language-specific fonts.....	81
Figure 18: Extending fontalias by secondary language.....	82
Figure 19: Translating Textelement in the Textmanager	84
Figure 20: Difference between name and naming	86
Figure 21: Definition of the object naming in the class editor	87
Figure 22: Changing the NOVALUE symbol.....	88
Figure 23: Defining component naming	89
Figure 24: Moving Textelements to other sections	94
Figure 25: Menu editor - Create new elements.....	98
Figure 26: Menu editor – Properties of a menu item.....	100
Figure 27: Creating a new currency in the frame	106
Figure 28: Creating a new feature.....	107
Figure 29: Feature editor - Overloading the init value	110
Figure 30: Initialisation of a component	129
Figure 31: Sorting components	140
Figure 32: Search for not set property "Sort"	142
Figure 33: Integrate Textelement in String constant	147
Figure 34: Feature editor - Editing values.....	148
Figure 35: Creating a new UOQ group in the frame	156

Figure 36: Maintenance of units of quantity in the frame.....	157
Figure 37: Allocate a unit of quantity to a feature	158
Figure 38: Groups in the structure tree	161
Figure 39: The rule editor	170
Figure 40: Creating linked rule conditions.....	171
Figure 41: Creating a rule with condition.....	173
Figure 42: Definition of a mixed rule explanation.....	183
Figure 43: Creating a new rule group	191
Figure 44: Assign rule group in rule editor	191
Figure 45: The constraint editor.....	195
Figure 46: Definition of a numerical format.....	203
Figure 47: Assign trigger of a feature.....	206
Figure 48: Creating a new result in the frame	214
Figure 49: Integration of cause variables in RTF-constants.....	220
Figure 50: Definition of a typecast in the item selector.....	226
Figure 51: Procedure editor – Method without side effects	227
Figure 52: Rule “always forbidden”	232
Figure 53: Administrate knowledge bases.....	245
Figure 54: Generating a new release version.....	247
Figure 55: Generating a new working version	248
Figure 56: Typecast in the item selector	253
Figure 57: Structure of a 2D-table	264
Figure 58: Component editor – Specifying a quantity	271
Figure 59: Class editor - Assigning a UOQ for components.....	273
Figure 60: Rule editor of a quantity rule	275
Figure 61: Editor of a decision table.....	284
Figure 62: User maintenance.....	289
Figure 63: Creating a new user.....	290
Figure 64: Securities of the knowledge base.....	292
Figure 65: Securities of the frame.....	293
Figure 66: Securities of a class.....	295
Figure 67: Export knowledge base.....	299
Figure 68: Import knowledge base	302
Figure 69: Package-Export.....	306
Figure 70: Package-Import	307
Figure 71: Integrity check of the complete knowledge base	309
Figure 72: Dialog “Reserved classes”	310

Figure 73: Create KIF	311
Figure 74: Call parameter to start the KIF.....	314
Figure 75: Starting an application without license	316
Figure 76: Textelement Info.....	318
Figure 77: Create a textelement.....	320
Figure 78: How to adjust the context name at the import of Textelements	324
Figure 79: Search dialog – Property “Left border”	336
Figure 80: Show date and dealer in the footer	338
Figure 81: Transparent graphics	343

35. Topic Overview

On the following pages you will find a structured overview of all topics that was dealt with in these training documents. The bold flagged page numbers point to the place in the document where the corresponding element or topic was for the first time introduced and defined.

The other page numbers refer to some of the most important text places on which these terms are used. A complete numbering of all references is not given for reasons of clarity.

2

2D-table	263
----------------	-----

A

Abbreviations	22
Article class	42
Assign trigger	206
Assignment	33

B

Base class.....	28 , 41
Boolean algebra	170

C

Call parameter	313
Cause variable	57 , 219
ChangeQuantity	276
Child class.....	38
Classes	
Securities.....	293
Column class.....	136
Component sorting	138 , 143
Components	44, 53
Check relevant	168
Create object.....	45, 75
Init value.....	66
List components.....	129 , 251, 269
Predecessor	46, 122
Quantity specification.....	271
Result output	224
Rules enabled.....	168
Sort	138 , 143
Translate relevant	88
Values.....	58
Conditions	173
constant	
multilingual string constants	147
Constants.....	71

Graphic constants.....	72
Multilingual string constants.....	208
Multilingual String constants	89
RTF constants.....	219
Constraints.....	195
Constructor	33 , 41, 121
Country code	79

D

Debug run.....	35, 41
Decision table.....	281
Destructor	41, 121

F

Feature	
Units of quantity	157
Features.....	107
Assign trigger	206
Format.....	202
Init value.....	66, 109
Multilingual	151
Reinitialize when object changes	113
Translate relevant	88
Value ranges.....	207
Values.....	108, 147
File extensions.....	300
Filter class.....	341
Filter expression.....	341
Find properties	141, 336
Form editor.....	31
Abstract representation	345
Alignment.....	333
Design mode	116
Form preview	31
Set name.....	32
Form elements	
Component tree	68
Component tree columns	135
Configurationbox columns.....	111, 158
Configurationbox component	56 , 60, 131
Configurationbox feature	149
Configurationbox icon.....	181, 210
Currency	205
Editline	201
Graphic	73
Groupbox	201
Label naming	202
Label static	31
Pushbutton	83
Quantity column	272
Result.....	336
Subform	257

Tab page	329
Form handle	30
Formats	202
Forms	29
Frame	17
Defaults	93
Securities.....	292
settings	18
Functions	
CheckAllRules	192
GetNaming.....	229
GetQuantity	276
GetUOQ	278
IsValid	209
LanguageDialogGet	101
LanguageDialogSet	85, 234
LicenseDemand.....	167
RuleActivate	191
RuleDeactivate	191
SetCurrency.....	106
SetRuleMode.....	192
SetRuleRoot.....	192
SystemSet ('DefaultIcon')	343
WinMessage	209
WinOpen	33
WinOpenDoc	214, 230
WinSetGraphic	343
WinSetTitle	345
WinStartModal	230

G

Groups	
in the structure tree	54
in the wasele list.....	23
List- and tree view	161

H

Hardware options.....	243
Hotkeys	349

I

IF-query	209
Inference machine.....	184
Init value	66, 109
Instructions	
Assignment.....	33
IF-query.....	209
Item selector	57

K

KIF	309 , 311, 312
Knowledge base	17
Administration.....	246, 291
Export.....	299
Import	301
Securities.....	291
Knowledge base options.....	164

L

List operators.....	279
Login.....	295

M

Main currency.....	105
Main language	79
Menus	97
Menu elements.....	98
Menu item options	101
Method	
Procedure editor	33
Methods	
Allow side effects.....	228, 266
Procedure editor	34

N

Namings	85
Component naming.....	88
Feature naming.....	88
NOVALUE naming.....	87
Object naming	86
NOVALUE Symbol.....	59

O

Object class	28 , 42
Object-oriented programming.....	37
Classes and objects.....	37
Inheritance.....	38, 43
Instantiation	38
Multiple inheritance.....	42
Overload.....	43
Online help.....	21
Overload	110

P

Parameter	314
Parent class	38
Password.....	295

Q

Quantities.....	271
How to forbid the quantity 0.....	278
Quantity rules.....	274

R

Readonly	120, 136
Recycler.....	327
Reinitialize when object changes.....	113
Release	239
Release version	245
Reserve.....	239
Reserved classes	241, 310
Restore memory	76
Result output.....	224
Results	214
Local sorting for result values	233
Rule editor	169
Rule explanation	182
Rule groups	191
Rule mechanism	
Constraints	195
Rule mechanisms	
Conditions.....	173
Decision table	281
Rule types	169
Rules.....	167
Always allowed	172
Always forbidden	232
Syntax.....	167
Weighting.....	174

S

Secondary currency	105
Secondary language.....	79
Selection trigger	74
Shortcuts.....	349
Start parameter.....	313
SubtreeOpen	137
Syntax check	34

T

text element.....	322
text elements.....	317
text manager.....	317
Text modules.....	216
Sorting.....	233
Tooltip.....	76
Translate relevant.....	88
Typecast.....	225 , 252, 255

U

Units of quantity	156 , 204, 273
User maintenance	289
User options.....	163
Users and securities.....	289

V

Validity symbols.....	180 , 332
Value ranges	207
Virtual class	42

W

Wasele	23
Working version.....	245