camos.

camos Develop
Developer training

Object pointer & Basics of debugger

Prerequisites

- **Carconfigurator on the state of the 3rd day of modeler training**

Contents:

- **Validity check**
  - Object pointer
  - Recursion
- **Debugger**
  - Stepping the validity check

Training targets

- **After this exercise you should…**
  - Create and use object pointers
  - Name application fields for object pointers
  - Program complex methods
  - Control the run of the application in single-step mode
  - Find an error in the application with the aid of the debug studio

Object pointer

- **Data type Object pointer**
  - stands in acting for an object
  - has components, features and methods of the object
  - a change on the pointer always changes the referenced object

- **Frequent use:**
  - Dynamic subforms
  - Methods that run through the object tree, e.g. allocation of a position number, recursive price calculation

Exercise: Validity check

- **Requirements:**
  - A quotation can only be created if no forbidden values were selected
  - Present rule violations should be displayed in a list

- **Solution:**
  - Via the function IsValid() a check can be carried out if the selected value of a Wasele is allowed or forbidden
  - Via the function Why() the explanation text of a ruled value can be determined

Exercise: Validity check

- **Procedure:**
  - IsValid() is called for each object of the configuration
  - If the object is not valid, the name of the forbidden object and the rule explanation are each written into a list
  - Then the lists are displayed on a form

Exercise: Validity check

- **Create method CheckChildren() in „Configuration"**
- **CheckChildren() has 2 parameters:**
  - ObjName[], string, list, call by „Write"
    - Contains the names of the invalid objects
  - RuleExp[], string, list, call by „Write"
    - Contains the rule explanation of each invalid object
- **Additionally three variables are needed:**
  - pModules[], object pointers to „Modules", list
    - List of object pointers to module objects
  - i, numerical, scalar
    - Loop variable
  - imax, jmax, numerical, scalar
    - Index variable

Exercise: Validity check

- **Procedure code of the method CheckChildren():**

```
# Determine module objects
pModules[] := GetObjPtr('Modules');
#
#
jmax := MaxIndex(pModules[]);
FOR i := 1 TO jmax DO
 # Check if object permitted
 IF not IsValid(pModules[i]) THEN
  #
  # if not, determine name and rule explanation
   imax := MaxIndex(ObjName[]) + 1;
   ObjName[imax] := GetNaming(pModules[i]);
   RuleExp[imax] := Why(pModules[i], pModules[i]);
 ENDIF;
 #
 # Does the object still have child objects? Check these too!
 pModules[i].CheckChildren(ObjName[], RuleExp[]);
ENDFOR;
```

Exercise: Validity check

- **Create init method CheckConfig() in class „start"**

```
# Initialization
NamesList[] := WhyList[] := NOVALUE;
# Trigger rule check
CheckAllRules();
# Check starts in object _Auto
_Auto.CheckChildren(NamesList[], WhyList[]);
#
IF MaxIndex(WhyList[]) = 0 THEN
  RETURN 1;
ELSE
  RETURN 0;
ENDIF;
```

- **Create string list features NamesList[] and WhyList[] under the group „Administration features"**

Exercise: Validity check

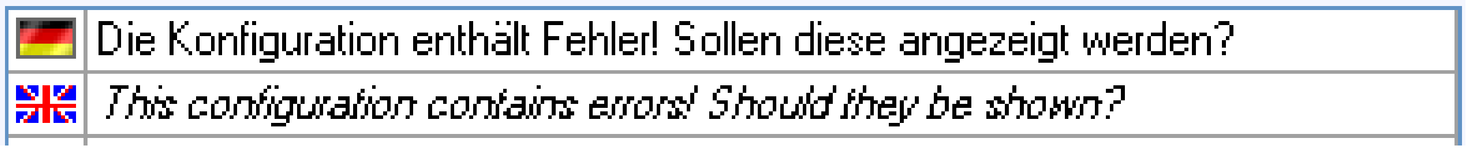
- **Display both lists on a form**
  - New form „ErrWindow“ in start
  - Table with two label columns
    - Assign cause variables, allocate headings, OK-button closes form

- **Check has to be carried out before the call of the result**
  - Menu „Administration“, menu item „Create offer“

```
IF CheckConfig() THEN
  WinStartModal(WinOpenDoc('Offer', 0, 0, 800, 600));
ELSE
  IF WinMessage('QUESTION', !MsgRuleConflict) THEN
     WinStartModal(WinOpen('ErrWindow'));
  ENDIF;
ENDIF;
```

Exercise: Validity check

- **Create the string constant !MsgRuleConflict**
  - Deposit a text element with the following contents



Die Konfiguration enthält Fehler! Sollen diese angezeigt werden?

This configuration contains errors! Should they be shown?

- **Test the application**

  -> Only faultily selected components are found

  -> Faulty features and components that have no value without permission are not yet found

Exercise: Validity check

- **Method CheckFeatures(ObjName[], RuleExp[])**
  - Variables: i + imax + jmax (numerical), Features[] (string list)
  - Parameter: ObjName[], RuleExp[], (string lists, call by „Write")

```
Features[] := GetAllFeature();
#
jmax := MaxIndex(Features[]);
FOR i := 1 TO jmax DO
   # check all features for validity
  IF not IsValid(<<Features[i]>>) THEN
  #
  # if not permitted, determine names and rule explanation
     imax := MaxIndex(ObjName[]) + 1;
     ObjName[imax] := GetNaming(Self, Features[i]);
     RuleExp[imax] := Why(<<Features[i]>>, <<Features[i]>>);
  ENDIF;
ENDFOR;
```

- **Add to CheckChildren():**

```
    CheckFeatures(ObjName[], RuleExp[]);
```

Exercise: Validity check

- **Method CheckNovalueComps(ObjName[], RuleExp[])**
  - Variables: i + imax + jmax (numerical), Comps[] (string list)
  - Parameter: ObjName[], RuleExp[], (string lists, call by „Write“)

```
Comps[] := GetAllComp();
# check all components for validity
jmax := MaxIndex(Comps[]);
FOR i := 1 TO jmax DO
  #
  IF <<Comps[i]>> = NOVALUE THEN
    IF not IsValid(<<Comps[i]>>, NOVALUE) THEN
    # if not permitted, determine names and rule explanation
      imax := MaxIndex(ObjName[]) + 1;
       ObjName[imax] := GetNaming(Self, Comps[i]);
       RuleExp[imax] := Why(<<Comps[i]>>, <<Comps[i]>>);
    ENDIF;
  ENDIF;
ENDFOR;
```

- **Add to CheckChildren():**

```
CheckNovalueComps(ObjName[], RuleExp[]);
```

Basics of the debug studio

Start or Continue after a stop <F5>

- Debugger is running
- Debugger stopped
- Debugger is running in another knowledge base

Restart <Ctrl> + <F5>

- The already active interpreter run is restarted, i.e. it is reset to the beginning of the program

Pause

- Via the menu item *Pause* the interpreter is switched to the single-step mode

Stop

- The interpreter run is cancelled at the current positon

Debugger operation

- **How do I stop the interpreter?**
  - Press Pause
  - Set Breakpoint

Step In <F11>

- Only in single-step mode
- The debugger executes exactly one program step and stops in the next command line
- So the run of the configuration can be traced step-by-step
- Method calls are further traced, i.e. the method is opened and processed step-by-step

Debugger operation

Step Over <F10>

- Only in single-step mode
- The debugger executes exactly one program step
- Calls of methods however are not further traced, but only considered as further program step

Step Out <Ctrl> + <F11>

- Only in single-step mode
- The debugger executes the current method uninterruptedly until the end (incl. all called methods, forms, etc.)
- After the method call the interpreter is stopped in the higher-order method

Debugger tools & features

- **The most important functions of the debug studio**
  - Stack
  - Object tree
  - Form tree
  - Evaluations: Profiler, report etc.

- **Debug features in the development system**
  - Watchlist in procedures
  - Global watchlist
  - Value display and value change during runtime
  - Opening the current procedure