

ClassiX[®]

ClassiX[®] **Tutorial 5**

Content

- 1. INTRODUCTION3**
 - 1.1 CONDITIONS FOR THIS TUTORIAL..... 3
- 2. TRANSACTIONS4**
 - 2.1 CAPTURING TRANSACTIONS.....4
 - 2.2 EDITING TRANSACTION DESCRIPTIONS6
 - 2.2.1 *Transaction descriptions* 7
 - 2.3 TRANSACTION MANAGER10
 - 2.4 COMPLETE APPLICATION.....11
 - 2.5 FORMULAS AND PLUGGING.....12
 - 2.5.1 *Specifications about the topic formulas and plugging*..... 15
 - 2.6 AN UPGRADING – AGGREGATION PER CALENDAR MONTH.....15
 - 2.7 MORE DIMENSIONS.....17
- 3. REVIEW.....19**
- 4. PROGRAM LISTINGS.....20**
 - 4.1 PROJECT FILE PERSON5.CXP.....20
 - 4.2 MODULE CONSULTING.MOD.....21
 - 4.3 FILE TUTORIAL5.EXT.....22

1. Introduction

This tutorial deals with transaction processing. What is a transaction? All company processes and activities that are also relevant for billing are called transactions. Usually they come with accordant **records**¹. Classical computer applications display them via dynamic data. For this, we have objects of the class **CX_TRANSACTION** or a derivated class in the ClassiX[®] system. Another kind of objects, **CX_MONITOR**, collect information about specific transactions. A typical example for a monitor is an account.

The requirements, how transaction data shall be grouped and **observed** in a certain way changes constantly. This is why the interplay between transaction objects and monitors must be controllable in a flexible way. It is possible, that further monitors shall be activated operationally to improve transparency of business operations in selected areas or in a certain time period.

In the ClassiX[®] system transaction processing is controlled via objects, which belong to company data just like any other CyberEnterprise[®] object.

A transaction manager processes this transaction *booking* based on these controlling objects. The following pages will show how this happens. With this, we will also talk about basic concepts of the ClassiX[®] system that we don't know from previous tutorials, yet. At first, a simple example provides a general view. **This creates holes**, but that's on purpose. The missing elements, such as the concept "plugging" will be covered later, so they are easier to understand as expedient features integrated in a complete system.

1.1 Conditions for this tutorial

To work through this tutorial without any problems, you will need the created files from all previous tutorials and the provided zip archive [cx_Freeware.zip](#) (see tutorial 1 and 3).

The file *person5.cxp.end* shows the final state of the exemplary application. Simply removing the ending ".end" we can already run the final state (for demonstration purposes).

The topics introduced here only work within the **complete** ClassiX[®] environment. Therefore you will need an ObjectStore database in this case. This we can't provide as a download for legal reasons. So please follow these topics in theory.

¹ order confirmation, delivery notes, material issue slips, posting documents, ...

2. Transactions

Again, the exemplary application is our starting point, and we imagine, the “person administration” was no end in itself but we have saved such persons in the database, who want to access a specific consulting service. The consulting is settled on an hourly rate. First we need an overview of hours performed per person. Since our consulting imparts particularly profound and helpful knowledge, we have many loyal customers. Therefore, an overview per customer and calendar month becomes important. We will introduce this in a second step. You might have guessed it: the consulting services are the transactions.

2.1 Capturing transactions

To introduce the topic transaction processing, we need a particularly simple example. That’s why we don’t want to use AppsWarehouse® modules. We will start with a small, new development:

```
Module(Consulting)
[
  Msg(START_CONSULTING)

  Var(person, txn)

  Define(Book)
    CreatePersObject(CX_TRANSACTION) -> txn
    txn DrainWindow
    /* transaction processing ? */ ;

  START_CONSULTING: OpenWindow(win)
]

Window(win, 35, 35, 1000, 120, T("Beratungen", "Consulting"))
{
  Group(g1, 10, 10, 500, 100, "")
  {
    Prompt(CX_PERSON::this, BLUE, 11, 11, "          ")

    Prompt(11, 22, "Datum")
    Date(CX_TRANSACTION::date, 111, 22, 300)

    Prompt(11, 33, "Dauer")
    String(CX_TRANSACTION::value, NF_DIMENSIONED, NF_AUTOMATIC_DECIMALS,
          111, 33, 300)

    Prompt(11, 44, "Thema")
    String(CX_TRANSACTION::description, 111, 44, 300)

    Button(book, NON_SELECTABLE, 23, 55, 120, 8, T("Verbuchen", "Book"))
    [
      SELECT: Book, Lock
    ]

    Attach(date, RIGHT, STRETCH, 15)
    Attach(value, RIGHT, STRETCH, 15)
    Attach(description, RIGHT, STRETCH, 15)
  }
}
```

```

    Attach(book, BOTTOM, 7)
}

Group(g2, 512, 10, 450, 100, "")
{
    ObjectTree(tree, 5, 5, 1, 1)
    [
        INITIALIZE: "CX_PERSON::this" SetFormat
                    FindAll(CX_PERSON) FillObox
        SELECT:      GetObject -> person, person FillWindow, Unlock(, book)
    ]
    Attach(tree, RIGHT, STRETCH, 5)
    Attach(tree, BOTTOM, STRETCH, 5)
}

Attach(g2, RIGHT, STRETCH, 10)
Attach(g1, g2, RIGHT, OPPOSITE, STRETCH, 10)
Attach(g1, BOTTOM, STRETCH, 10)
Attach(g2, BOTTOM, STRETCH, 10)
}

```

In the right part of the window, we see all persons². We can enter accordant data of a consulting service in the left part for a selected person. The consulting time is saved in the dynamic data field *value*, a summary of the counselling interview is entered in the data field *description*. We chose the name “**booking**” for the entering process, although now there is only an object of the class **CX_TRANSACTION** being created. The actually interesting part of the processing is still missing!

Now we complement the menu item in the batch main program `person5.cxp`:

```

Module (GLOBAL)
[
    Msg(LOGIN,   LOGGED_IN,   EDIT_USER,   PERSONAL_FILE,   . . . .
    Msg(LIST_SECURITY_GROUP, START_CONSULTING)
]

Window(win, BITMAP_SIZE, 1, 1, 1200, 30, . . .
[
    INITIALIZE: SendMsg(LOGIN)
]
{
    Menu
    {
        Item(NON_SELECTABLE, T("Personenverwaltung", "Personal File"))
        [
            SELECT: SendMsg(PERSONAL_FILE)
            LOGGED_IN: Unlock
        ]
        Item(NON_SELECTABLE, T("Beratung", "Consulting"))
        [
            SELECT: SendMsg(START_CONSULTING)
            LOGGED_IN: Unlock
        ]
    }
}

```

and the matching *Extern* statement in `tutorial5.ext`

```

Extern(Consulting, File(consulting.mod), triggeredBy(START_CONSULTING)

```

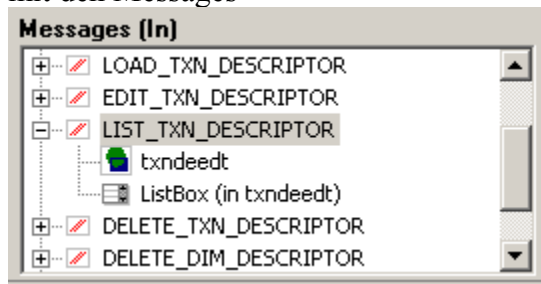
² See picture in 2.4

2.2 Editing transaction descriptions

For now, we leave the module *Consulting* in its semi-finished state, and upgrade the batch main program – `person5.cxp` – once more, so we can enter the transaction descriptions. Here, the AppsWarehouse® helps, and the browser recommends the module *txndeedt* after we have entered the keyword “transaction descriptions”.

```
[-] txndescr Transaktionsbeschreibung Basismodul
    [-] txndeedt Transaktionsbeschreibung Editiermodul
    [-] txndesel Transaktionsbeschreibung Selektionsmodul
```

mit den Messages



We integrate another menu item:

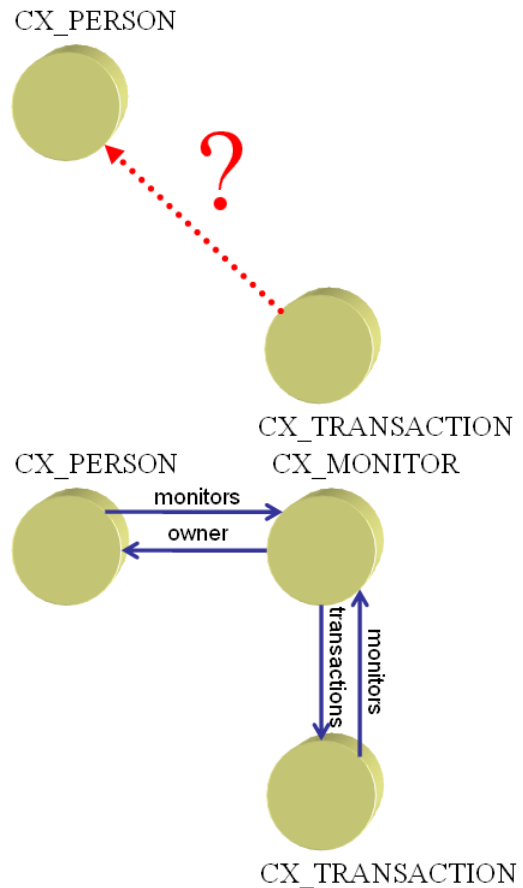
```
Module (GLOBAL)
[
    Msg(LOGIN, LOGGED_IN, EDIT_USER, PERSONAL_FILE, . . . .
    Msg(LIST_SECURITY_GROUP, LIST_TXN_DESCRIPTOR, START_CONSULTING)
]

Window(win, BITMAP_SIZE, 1, 1, 1200, 30, . . .
[
    INITIALIZE: SendMsg(LOGIN)
]
{
    Menu
    {
        Item(NON_SELECTABLE, T("Personenverwaltung", "Personal File"))
        [
            SELECT: SendMsg(PERSONAL_FILE)
            LOGGED_IN: Unlock
        ]
        Item(NON_SELECTABLE, T("Beratung", "Consulting"))
        [
            SELECT: SendMsg(START_CONSULTING)
            LOGGED_IN: Unlock
        ]
        Item(NON_SELECTABLE, T("Transaktionsbeschreibungen", "Transaction
descriptors"))
        [
            SELECT: NULL SendMsg(LIST_TXN_DESCRIPTOR)
            LOGGED_IN: Unlock
        ]
    ]
}
```

The necessary Extern statements already exist in `classix.ext`.

2.2.1 Transaction descriptions

Now we are ready to enter the transaction descriptions. How these descriptions look, depends on what we want to achieve. We have created an object of the class `CX_TRANSACTION`. The data field *value holds/contains* the expenditure of time. The transaction object describes the consulting service for one person, therefore there is a contextual connection to a specific object of the class `CX_PERSON`. The InstantView® variable *person* refers to this object. This is the initial situation:



Additionally, we know, that transactions can be **grouped** (only) by monitor objects.

All classes that have been derivated from `CX_BUSINESS_OBJECT` – `CX_PERSON` is one of them – can refer to monitor objects with the data field *monitors*, the data field *owner* is responsible for the revertive relation.

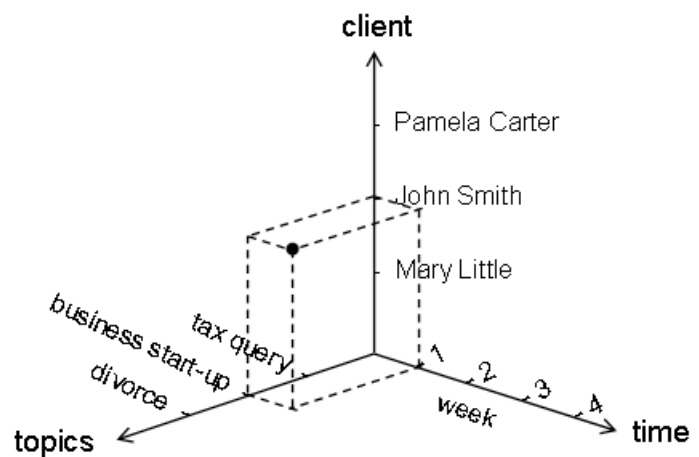
Transactions and monitors are connected amongst each other with an *m-n*-relation (*monitors*, *transactions*).

The result should look like this:

One thing is clear, it accords to the features of the involved classes and it doesn't need to be described explicitly.

It only stays open, where the monitor object comes from.

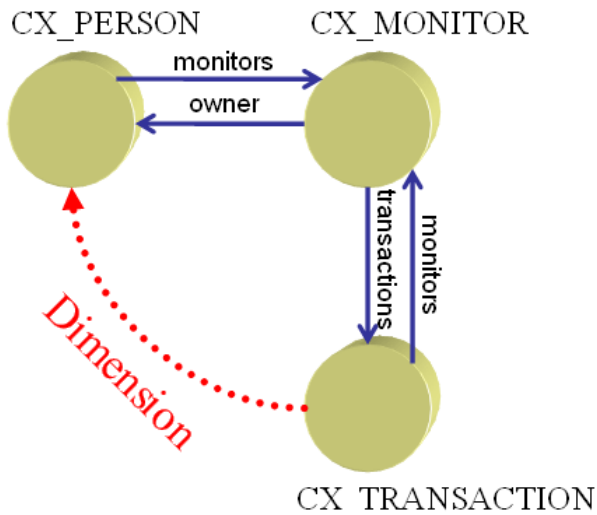
And exactly this is defined in the transaction description. This definition forms the first part of the transaction description. The way to the monitor is stated as an access expression referring to the transaction object. Instead of using "way to the monitor", we say dimension. Why dimension? Our example is not really suited to explain this; it is only one-dimensional.



You could think of the consulting service to be **settled** per person, per time unit (week) and per consulting topic – each transaction takes up on a point in a three-dimensional space³:

Back to our (one-dimensional) example. How could our dimension description look like?

³ Of course, there are no restrictions for the number of dimensions.



The only connection to the `CX_PERSON` object is an `InstantView`[®] variable and we can declare the access expression

```
var (person) .monitors [0]
```

as the dimension description⁴.

An alternative would be a relation from the transaction to the `CX_PERSON` object:

```

Define (Book)
  CreatePersObject(CX_TRANSACTION) -> txn
  person txn Link(masterObject)
  txn DrainWindow
  /* transaction processing ? */ ;

```

And as a dimension description we would simply have: `masterObject.monitors [0]`

But we stay with the first, a bit more indirect variant.

So far, it unfortunately only works, if a monitor object already exists. For the initial consulting service, it has to be created at first, though, and this is exactly, what this function does - *ForceMonitor*.

⁴ *var* is a pseudo function just like *call* (compare tutorial 1 – paragraph 4.3.3). *var(x)* simply provides the value for the `InstantView`[®] variable *x*. In contradiction to *call*, here – with *var* – the connection to the object, for which an access expression is called, gets totally lost. Statement sequences, such as *object Copy(var(x))* seem artificial and unnecessary. The pseudo function is useful, when access expressions appear in formulas (class `CX_FORMULA`) or as a dimension expression for a transaction description.

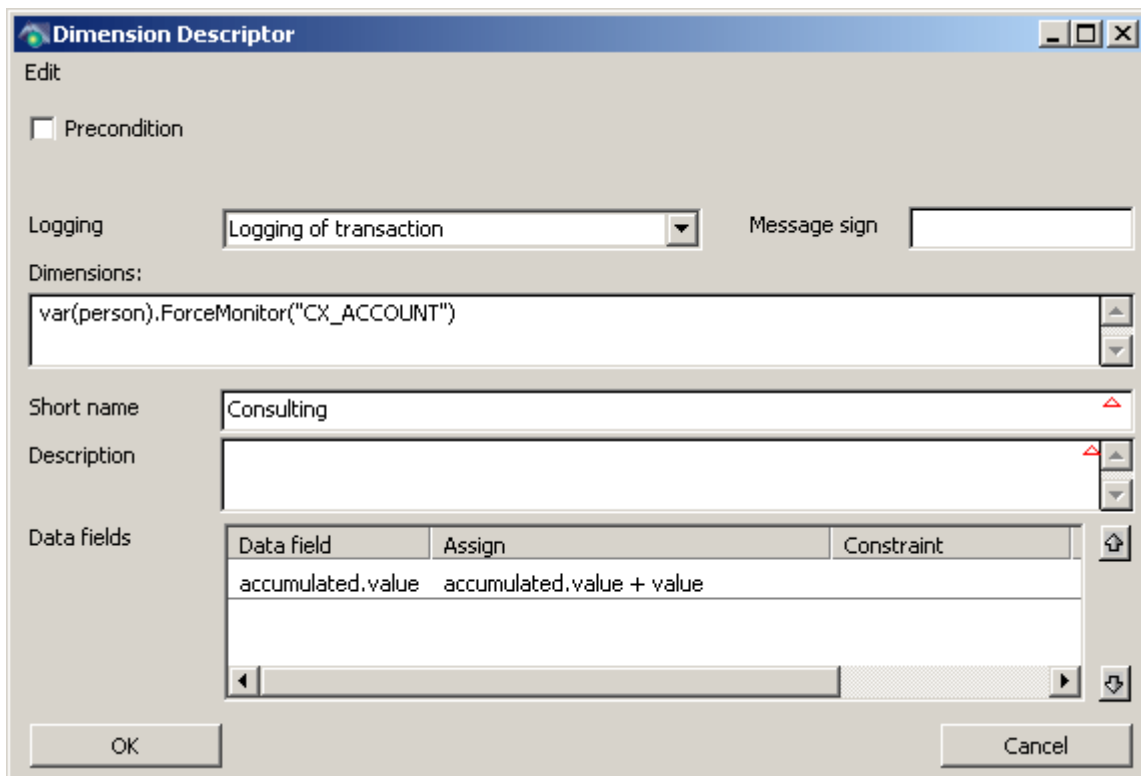
We upgrade the dimension description:

```
var (person) . ForceMonitor ("CX_ACCOUNT")
```

with the parameter, we control the type of the monitor object that is being created. Here, we decide for `CX_ACCOUNT`.

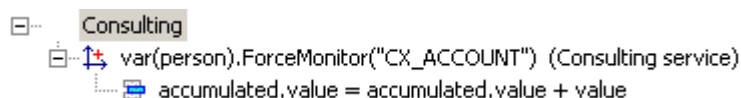
Additionally we wish that the monitor – our `CX_ACCOUNT` object – sums up all performed consulting hours. For this, we assign the dynamic data field `accumulated.value` (`accumulated` is a specifier!).

The formula `accumulated.value + value` describes, what value `accumulated.value` receives in a transaction processing. For a formula to makes sense, `value` needs to refer to the `CX_TRANSACTION` object and `accumulated.value` to the monitor (`CX_ACCOUNT`). At the moment we just take it for granted⁵.



As a registration modus, we require, that the *m-n*-relation (*monitors, transactions*) displayed above gets established between transaction and monitor⁶.

Now the transaction description is complete:



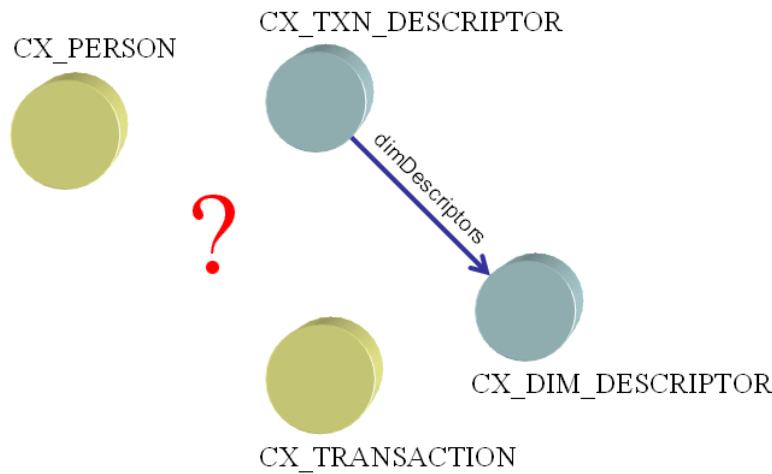
What do we use it for?

In the semi-finished module *Consulting* we have a transaction and a `CX_PERSON` object, and

⁵ bis zur Erklärung in 2.5

⁶ Es ist auch möglich, die Transaktionen nicht zu registrieren. Es kann sinnvoll sein, dass ein der Monitor nur die Werte der Transaktionen akkumuliert.

now, objects of the classes `CX_TXN_DESCRIPTOR` and `CX_DIM_DESCRIPTOR` have been added.



The transaction manager creates the missing connection.

2.3 Transaction manager

The transaction manager – a transient object – gets constructed from a transaction description:

```

Var (txnManager)

0 FindAll (CX_TXN_DESCRIPTOR) GetElement
Call (CreateTxnManager) -> txnManager
  
```

and we achieve the transaction processing (the “**booking**”) with

```

txnManager txn Call (GetProcessed)
  
```

with the variable *txn* **holding** the transaction.

A transaction manager (object of the class `CX_TXN_MANAGER`) always develops from an object of the class `CX_TXN_DESCRIPTOR`. It **contains/holds** the controlling information in a separate, internal, form which is optimised for fast processing.

If changes of in description objects (`CX_TXN_DESCRIPTOR` and `CX_DIM_DESCRIPTOR`) shall take effect straight away, a new transaction manager needs to be created. We didn’t go through that trouble for the exemplary application. Therefore, after editing the transaction description it is necessary to restart.

2.4 Complete application

Now the module *Consulting* can be made “fully functional”. We want to tolerate several transaction descriptions and *Consulting* uses the ones with the identifier “A”. Without a transaction description, the consulting service can’t be “booked” - therefore we implement a problem report for this case. And with the tree (right side) we also want to show the account ([CX_ACCOUNT](#)) with the transactions.

```
Module(Consulting)
[
  Msg(START_CONSULTING)

  Var(person, txnManager, txn)

  Define(Book)
    CreatePersObject(CX_TRANSACTION) -> txn
    txn DrainWindow
    txnManager txn Call(GetProcessed);

  START_CONSULTING: OpenWindow(win)

  INITIALIZE:
    "uniqueID = \"A\"" FindFirst(CX_TXN_DESCRIPTOR)
    Dup ifnot { "Transaktionsbeschreibung ?" Attention }
    Call(CreateTxnManager) -> txnManager
    Oh Call(SetDefaultUnit) // create new CX_VALUES with unit hour
]

Window(win, 35, 35, 1000, 120, T("Beratungen", "Consulting"))
{
  Group(g1, 10, 10, 500, 100, "")
  {
    Prompt(CX_PERSON::this, BLUE, 11, 11, " ")
    Prompt(11, 22, "Datum")
    Date(CX_TRANSACTION::date, 111, 22, 300)

    Prompt(11, 33, "Dauer")
    String(CX_TRANSACTION::value, NF_DIMENSIONED, NF_AUTOMATIC_DECIMALS,
          111, 33, 300)

    Prompt(11, 44, "Thema")
    String(CX_TRANSACTION::description, 111, 44, 300)

    Button(book, NON_SELECTABLE, 23, 55, 120, 8, "Verbuchen")
    [
      SELECT: Book, ClearObox(, tree) FindAll(CX_PERSON) FillObox(,
tree),
      Lock
    ]

    Attach(date, RIGHT, STRETCH, 15)
    Attach(value, RIGHT, STRETCH, 15)
    Attach(description, RIGHT, STRETCH, 15)

    Attach(book, BOTTOM, 7)
  }
}
```

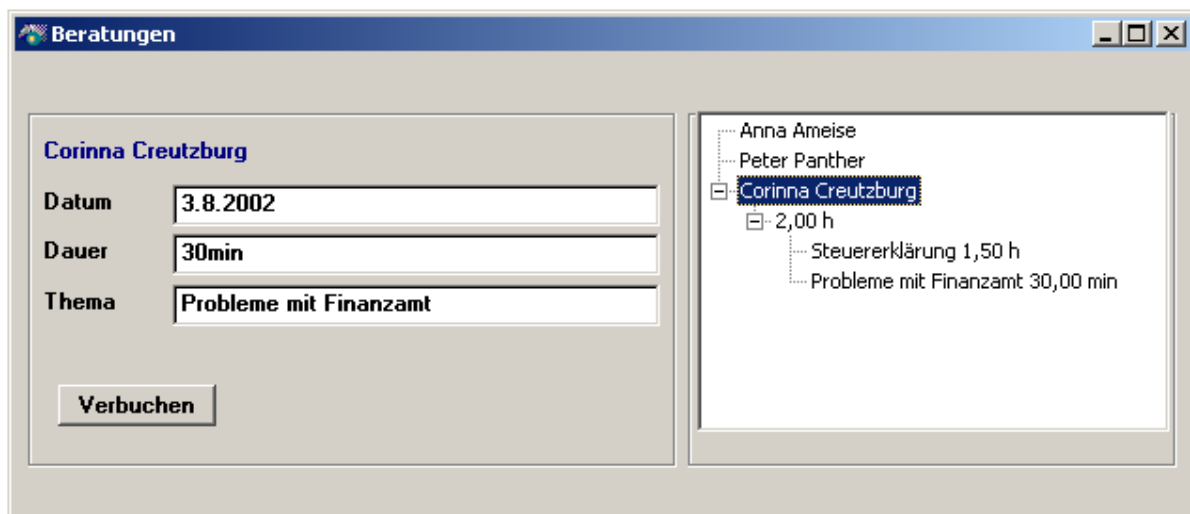
```

Group(g2, 512, 10, 450, 100, "")
{
  ObjectTree(tree, 5, 5, 1, 1)
  [
    INITIALIZE: "CX_PERSON::this" SetFormat
               [ "CX_PERSON::monitors" NODE ] SetFormat
               "CX_ACCOUNT::accumulated.value" SetFormat
               "CX_MONITOR::transactions" NODE ] SetFormat
               [ "CX_TRANSACTION::description" ] SetFormat
               [ "CX_TRANSACTION::value" ] SetFormat
               FindAll(CX_PERSON) FillObox
    SELECT:    GetObject -> person, person FillWindow, Unlock(, book)
  ]
  Attach(tree, RIGHT, STRETCH, 5)
  Attach(tree, BOTTOM, STRETCH, 5)
}

Attach(g2, RIGHT, STRETCH, 10)
Attach(g1, g2, RIGHT, OPPOSITE, STRETCH, 10)
Attach(g1, BOTTOM, STRETCH, 10)
Attach(g2, BOTTOM, STRETCH, 10)
}

```

Now, the entry of a consulting service looks like this:



2.5 Formulas and plugging

Operational instructions, such as

$$\text{totalprice} = \text{priceperunit} * \text{amount}$$

and **condition table** (e.g. price list, **quantity scaling**) shall be maintained by the user. They are no longer part of the program code but there is a formula object and the conditioned bag with their conditions also being formula expressions. The objects use a special mechanism assigning the variables in the formulas to specific object data fields.

Operational instructions, selection rules and other **partial algorithms** are often tightly integrated in the company data, and don't belong in the program code. Typical examples are price lists:

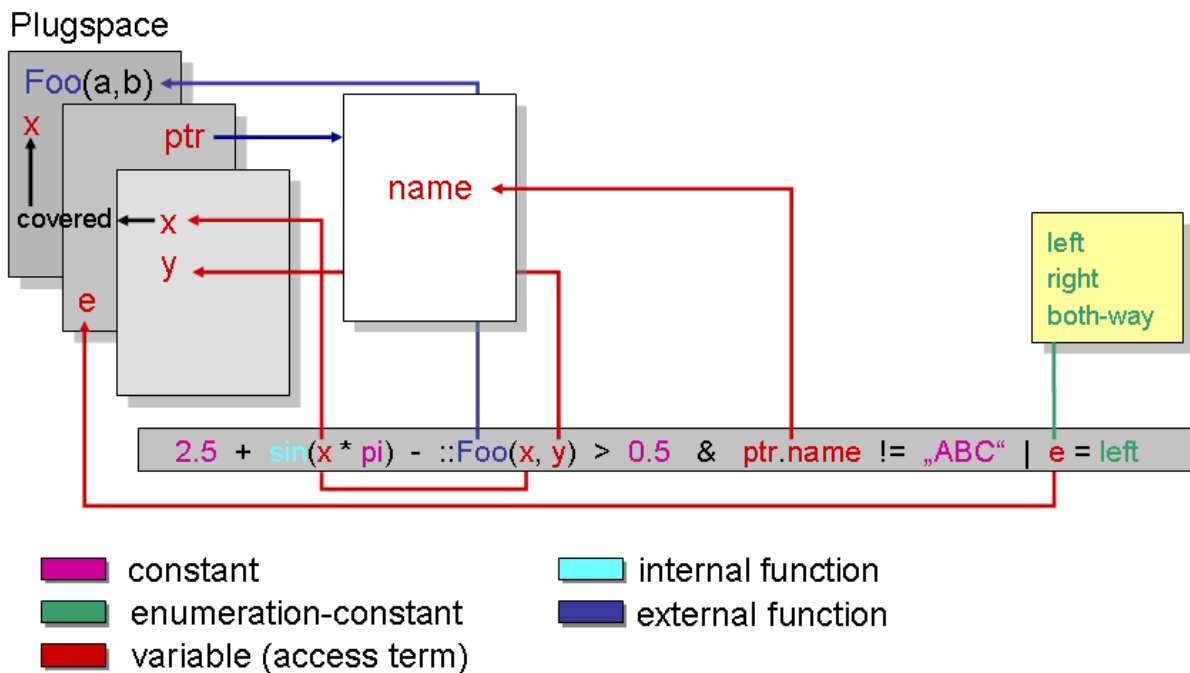
With a **full scope for design**, these are the data the user needs to maintain.

Internal functions are member functions of an object, which develops calculating a term that is included in the formula.

Access expressions and external functions refer to objects that haven't been qualified so far. Evaluating the formula, data fields/methods become initially assigned to the operands data fields/methods. The **bindungsmechanismus / attachment mechanism** is very important.

Registering a vector of objects (PlugSpace), a **new** context develops, through which the access expressions and function calls receive a concrete meaning.

The PlugSpace can be constantly modified; the object order determines, *from where* a data element is preferably is taken⁷. Therefore the formula objects are well-suited to map all operational instructions, occurring in any context in a company.



In this example, `x`, `y`, `ptr` and `e` are variables, that get **attached** to data fields of objects in the PlugSpace before the calculation. This also refers to the function `Foo`.

The objects in PlugSpace are highlighted in blue; the order (declining intensity of colour) determines which variable gets **attached** to which object (the `x` in the last object stays covered for as long as the first object remains in PlugSpace).

`2.5`, `0.5` and `pi` are numeric constants. For strings (constants `„ABC“`) **comparison operations** and chainings are defined. The enumeration constant `left` belongs to the domain, registered in the system for the variable `e`. Function `sin` is a member function of the object developing the operation `x * pi`.

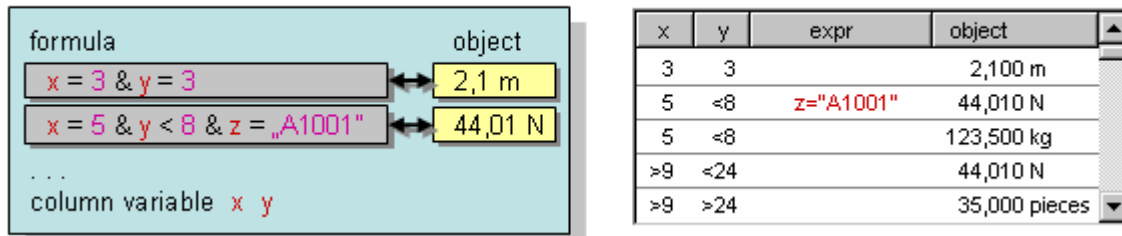
In a formula expression, an object of the class `CX_FORMULA` / the class `CX_CONDITIONED_BAG` described below can appear as an operand.

⁷ Due to the concept of dynamic data fields, here are **larger degrees of freedom**. (also see)

An object of the class `CX_CONDITIONED_BAG` is a container for an **assemblage of objects**, whereas for each object, there is one expression defined, realised with `CX_FORMULA`; if the accordant object logically belongs to it, or not:

To realise a presentation in a table, **column variables** are extracted from the **formula expression**.

CX_CONDITIONED_BAG



`CX_CONDITIONED_BAG` as a data member, can be understood as a conditional pointer – some kind of switch, with the current PlugSpace **occupancy** referring to its current state. Accordingly, the operator '->' on the C++ level / the dot operator used in the access expression **overcharged** for `CX_CONDITIONED_BAG`.

Formula objects are multilingual, too: enumeration constants as well as names of units for dimensionful constant are displayed in the formula expression in the currently activated language. Formula expressions with enumeration constants stay correct, even when the enumeration identifiers get changed.

An object of the class `CX_FORMULA` gets used for the calculation

$$accumulated.value + value$$

A formula object contains an arithmetic expression with constants and variables; the variables are access expressions.

The formula object doesn't contain any information about which objects the variables in the formula expression refer to.

So a formula can be calculated, there must be a context, which determines the values of the variables. This context is simply an ordered amount of objects. For each variable (= access expression) the objects get tested one after the other, if there is a value for this access expression. This value is the (temporary) value of the variable for the current calculation. For the context (= ordered **object collection/ assemblage**), we introduce the term PlugSpace.

Before calculating a formula, the variables get **connected** to accordant data fields of the objects in PlugSpace. The **connection** is only valid for the current calculation. Die Bindung gilt nur für die aktuelle Berechnung. In case of ambiguousness, the object order in the PlugSpace is **deciding**.

Objects get registered as context for the plugging with the statement `PlugSpace`. Further statements to manipulate the PlugSpace are `PlugSpacePush` and `PlugSpacePop`.

An example:

First, two objects are “put into the PlugSpace”, whereas the variable *person* shall already contain an object `CX_PERSON` (Anna Ameise)

```
Var(formula, tmp, person)

CreateTransObject(CX_TRANSACTION) -> tmp, 150.0cm tmp Put(value)
[ tmp person ] PlugSpace
```

and after, we design and calculate formula expressions

```
CreateTransObject(CX_FORMULA) -> formula
“(value + 0.5m) / 2.0s” formula Put
formula Call(Evaluate)
```

delivers the value 2m/s, while we receive the string “Anna*Ameise” with

```
“firstName + \”*\” + name” formula Put
formula Call(Evaluate)
```

The transaction manager independently puts the transaction and the monitor into the PlugSpace, before a formula like

accumulated.value + value

is evaluated.

2.5.1 Specifications about the topic formulas and plugging

Independently from the transaction processing, the class `CX_FORMULA` plays a central role in the ClassiX[®] system. Many calculations are defined by the user⁸ and are therefore part of his or her data. Formulas are parts of another object – `CX_CONDITIONED_BAG`. Here, formulas are **predicates** for the selection of specific objects.

Check out the opportunities of this class in the CyberEnterprise[®] documentation.

Access to the data in the PlugSpace is also possible on the InstantView[®] level:

Statement *Plug*.

With the PlugSpace⁹ used above

```
CreateTransObject(CX_TRANSACTION) -> tmp, 150.0cm tmp Put(value)
[ tmp person ] PlugSpace
```

we would receive the string “Anna” and a `CX_VALUE` object 150cm with

```
Plug(firstName)
Plug(value)
```

2.6 An upgrading – **aggregation** per calendar month

⁸ therefore the syntax for the formula expressions is very similar to arithmetic expressions in BASIC

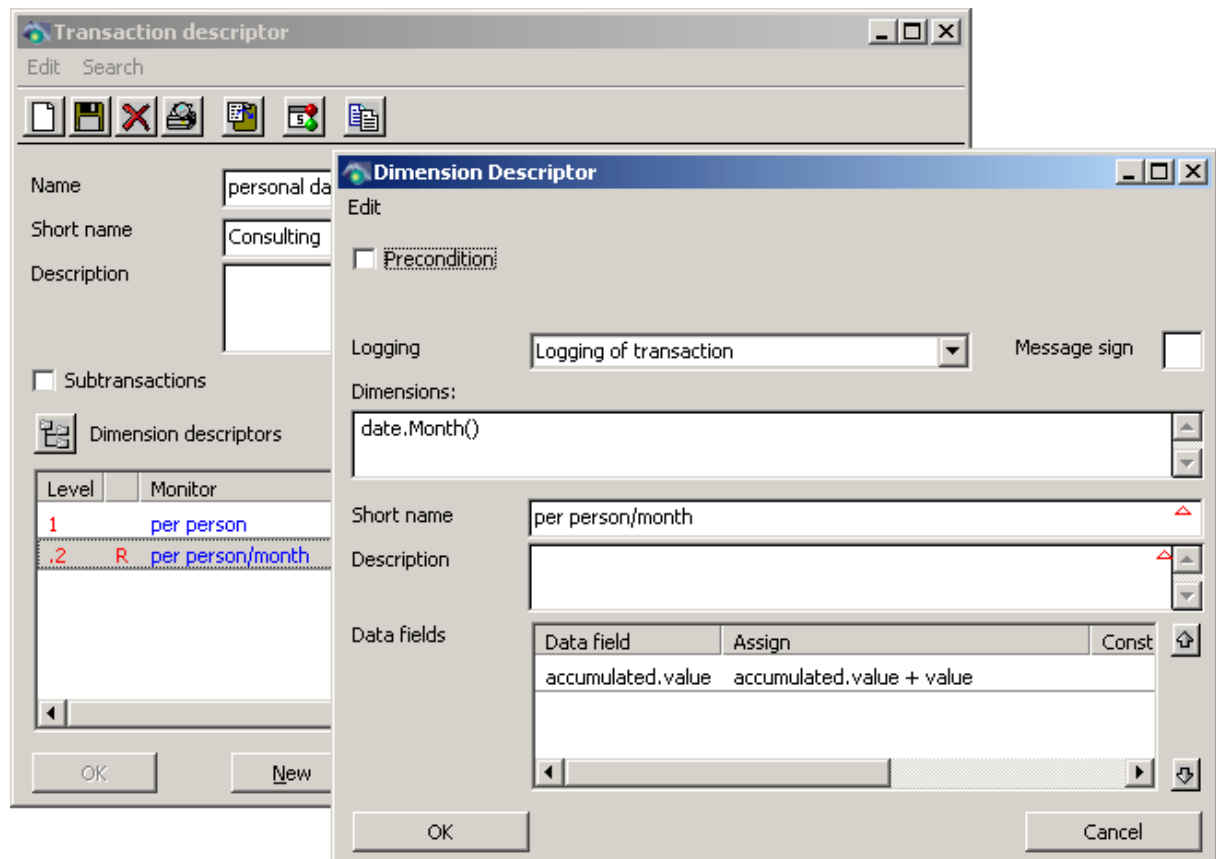
⁹ variable *person* contains the `CX_PERSON` object Anna Ameise again

The task definition originally was the **aggregation** of all consulting services of one client per calendar month. The total amount per client shall be maintained though.

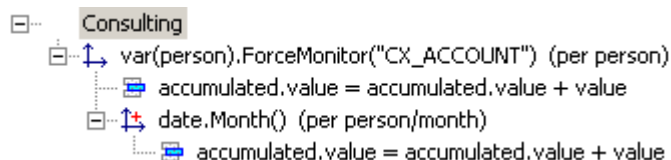
At first, we upgrade the transaction description:

- Step 1, we now select „no transaction registration”, since the transaction shall be assigned to the next step.
- For the subordinate description we select `date.Month()` as the dimension.

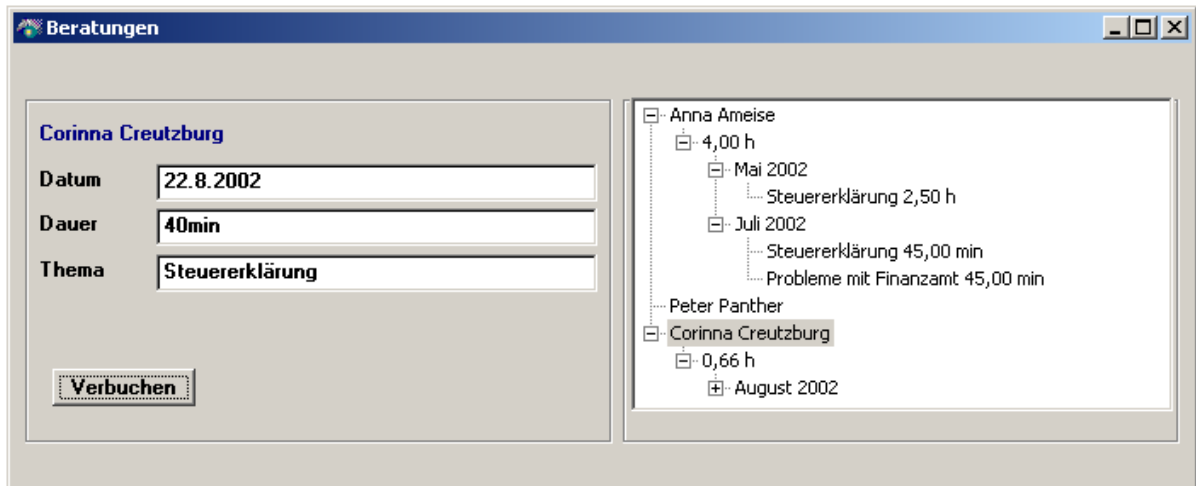
The formula expression for the accumulation of time expenditure is stated for both steps, since we want to see the total hours and the hours per month.



The transaction description in the tree diagram now looks like this:



To present registered transactions the way it is shown here,



we need to upgrade the format specifications:

```
ObjectTree(tree, 5, 5, 1, 1)
[
  INITIALIZE: "CX_PERSON::this" SetFormat
             "CX_MONITOR::accumulated.value" SetFormat
             [ "CX_PERSON::monitors" NODE ] SetFormat
             [ "CX_MONITOR::subMonitors" NODE ] SetFormat
             [ "CX_MONITOR::transactions" NODE ] SetFormat
             "CX_LOG_CUBE::dimensions[1]" SetFormat
                                     // show sub-dimension
             "CX_TRANSACTION::description" SetFormat
             "CX_TRANSACTION::value" SetFormat
             FindAll(CX_PERSON) FillObox

  SELECT:    GetObject -> person, person FillWindow, Unlock(, book)
]

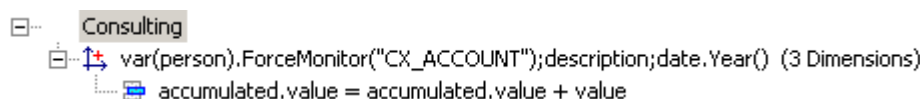
```

An object of the class `CX_LOG_CUBE`¹⁰ gets created for the sub-dimensions. For *accumulated.value*, we change the class in `CX_MONITOR`, to make the format element be used for `CX_ACCOUNT` and `CX_LOG_CUBE`. We make the month become the first dimension, 0th dimension is the `CX_ACCOUNT` object. And we have to make sure, that the sub-monitors appear in the tree diagram at all.

2.7 More dimensions

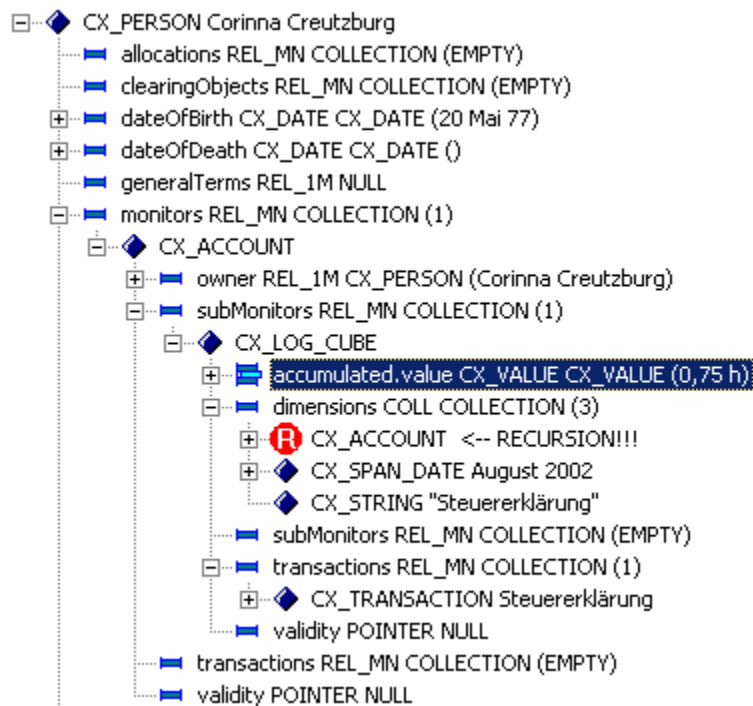
In paragraph 2.2.1, we referred to a hypothetical, 3-dimensional case to justify the term “dimension”. At the end, we want to describe this case and have a look at the monitor objects being created during a transaction booking.

Several dimensions get separated by semicolon. The new access expressions are *description* und *date.Year()*.



¹⁰ also derivated from `CX_MONITOR`

Now, only those transactions are **aggregated**, that stand for the same client and the same topic in one year. After entering a transaction, we look at the affected objects with the object inspector:



We see, that objects of the class **CX_LOG_CUBE** are even being used for the multidimensional case. Just like for the case of the sub-dimension (previous paragraph) the following applies:

The 0th dimension of a **CX_LOG_CUBE** object is always a monitor.

3. Review

You have acquired a general idea of transaction processing, and you know that transaction descriptions are the medium used by the ClassiX[®] system to provide high flexibility.

You know the concept “plugging” as an indirect data medium and you have seen that operational instructions of the user become part of the company data with objects of the class CX_FORMULA.

4. Program listings

4.1 Project file person5.cxp

```
// person5.cxd - Tutorial V
// Copyright (c) 2002 ClassiX Software GmbH, Hamburg (Germany)
// All rights reserved

// $Header: //iv/Freeware/Projects/Tutorial1/tutorial1.cxp#4 $ $DateTime: 2007/09/14 09:53:25 $ $Author: Bohl $

#include "tutorial5.ext"
#include "classix.ext"

Module(GLOBAL)
[
  Msg(LOGIN, LOGGED_IN, EDIT_USER, PERSONAL_FILE, INSPECT_OBJECT, START_MINI_WORKBENCH)
  Msg(LIST_SECURITY_GROUP, LIST_TXN_DESCRIPTOR, START_CONSULTING)
]

Window(win, BITMAP_SIZE, 1, 1, 1200, 30, T("Beispiel Anwendung (Tutorial V)", "Sample Application (Tutorial V)"),
tutorial5.bmp)
[
  INITIALIZE: SendMsg(LOGIN)
]
{
  Menu
  {
    Item(NON_SELECTABLE, T("Personenverwaltung", "Personal File"))
    [
      SELECT: SendMsg(PERSONAL_FILE)
      LOGGED_IN: Unlock
    ]
    Item(NON_SELECTABLE, "Beratung")
    [
      SELECT: SendMsg(START_CONSULTING)
      LOGGED_IN: Unlock
    ]
    Item(NON_SELECTABLE, T("Transaktionsbeschreibungen", "Transaction descriptors"))
    [
      SELECT: NULL SendMsg(LIST_TXN_DESCRIPTOR)
      LOGGED_IN: Unlock
    ]

    Item(NON_SELECTABLE, T("Anwender", "User"))
    [
      LOGGED_IN: Unlock
    ]
    {
      Item(T(Verwalten, Edit))
      [
        SELECT: SendMsg(EDIT_USER)
      ]
      Item(T(Zugriffsrechte, "Access Security"))
      [
        SELECT: NULL SendMsg(LIST_SECURITY_GROUP)
      ]
    ]

    Item(NON_SELECTABLE, "System")
    [
      LOGGED_IN: TestMsg(INSPECT_OBJECT) if Unlock else { TestMsg(INSPECT_OBJECT) if Unlock }
    ]
    {
      Item(NON_SELECTABLE, "Object Inspector")
      [
        SELECT: SendMsg(INSPECT_OBJECT)
        LOGGED_IN: TestMsg(INSPECT_OBJECT) if Unlock
      ]
      Item(NON_SELECTABLE, "Mini Workbench")
      [
        SELECT: SendMsg(START_MINI_WORKBENCH)
        LOGGED_IN: TestMsg(START_MINI_WORKBENCH) if Unlock
      ]
    ]
  }
}
}
```

4.2 Module consulting.mod

```
// consulting.mod - Tutorial V
// Copyright (c) 2007 ClassiX Software GmbH, Hamburg (Germany)
// All rights reserved

// $Header: //iv/Freeware/AppsWh/Freeware/consulting.mod#3 $ $DateTime: 2007/09/18 10:23:04 $ $Author: Bohl $

Module(Consulting)
[
  Msg(START_CONSULTING)

  Var(person, txnManager, txn)

  Define(Book)
  CreatePersObject(CX_TRANSACTION) -> txn
  txn DrainWindow
  txnManager txn Call(GetProcessed)
  ;

  START_CONSULTING: OpenWindow(win)

  INITIALIZE: "uniqueID = \"A\"" FindFirst(CX_TXN_DESCRIPTOR) Dup ifnot { "Transaktionsbeschreibung ?" Attention
                                                                    cancel
                                                                    }
              Call(CreateTxnManager) -> txnManager
              Oh Call(SetDefaultUnit)
]

Window(win, 35, 35, 1000, 120, "Beratungen")
{
  Group(g1, SELECT_MULTIPLE, 10, 10, 500, 100, "")
  {
    Prompt(CX_PERSON::this, BLUE, 11, 11, " ")
    Prompt(11, 22, "Datum")
    Date(CX_TRANSACTION::date, 111, 22, 300)

    Prompt(11, 33, "Dauer")
    String(CX_TRANSACTION::value, NF_DIMENSIONED, NF_AUTOMATIC_DECIMALS, 111, 33, 300)

    Prompt(11, 44, "Thema")
    String(CX_TRANSACTION::description, 111, 44, 300)

    Button(book, NON_SELECTABLE, 23, 55, 120, 8, "Verbuchen")
    [
      SELECT: Book, ClearObox(, tree) FindAll(CX_PERSON) FillObox(, tree), Lock
    ]

    Attach(date, RIGHT, STRETCH, 15)
    Attach(value, RIGHT, STRETCH, 15)
    Attach(description, RIGHT, STRETCH, 15)

    Attach(book, BOTTOM, 7)
  }

  Group(g2, 512, 10, 450, 100, "")
  {
    ObjectTree(tree, 5, 5, 1, 1)
    [
      INITIALIZE: "CX_PERSON::this" SetFormat
                "CX_LOG_CUBE::accumulated.value" SetFormat
                [ "CX_PERSON::monitors" NODE ] SetFormat
                [ "CX_MONITOR::subMonitors" NODE ] SetFormat
                [ "CX_MONITOR::transactions" NODE ] SetFormat
                "CX_LOG_CUBE::dimensions[1]" SetFormat // show sub-dimension
                "CX_TRANSACTION::description" SetFormat
                "CX_TRANSACTION::value" SetFormat
                FindAll(CX_PERSON) FillObox

      SELECT:   GetObject -> person, person FillWindow, Unlock(, book)
    ]
    Attach(tree, RIGHT, STRETCH, 5)
    Attach(tree, BOTTOM, STRETCH, 5)
  }

  Attach(g2, RIGHT, STRETCH, 10)
  Attach(g1, g2, RIGHT, OPPOSITE, STRETCH, 10)
  Attach(g1, BOTTOM, STRETCH, 10)
  Attach(g2, BOTTOM, STRETCH, 10)
}
}
```

4.3 File tutorial5.ext

```
// tutorial5.ext - link to external modules - Tutorial V
// Copyright (c) 2002 ClassiX Software GmbH, Hamburg (Germany)
// All rights reserved

Msg(PERSONAL_FILE, SHOW_ALL_PERSONS, PARENT_CHILD_RELATIONSHIP, FAMILY_TREE_1, FAMILY_TREE_2, SHOW_QUERY_RESULT)
Msg(X)

Extern(AllPersons, File(allpersons.mod))
Extern(AllPersons2, File(allperson2.mod), triggeredBy(SHOW_ALL_PERSONS)) : AllPersons

Extern(QueryResults, File(queryres.mod), triggeredBy(SHOW_QUERY_RESULT))

Extern(Relation, File(relation.mod))
Extern(ControlledRelation, File(controlledrel.mod)) : Relation
Extern(AdvancedRelation, File(advancedrel.mod), triggeredBy(PARENT_CHILD_RELATIONSHIP)) : ControlledRelation

Extern(FamilyTree, File(ftree.mod), triggeredBy(FAMILY_TREE_1, FAMILY_TREE_2))

Extern(PersonalFile, File(person.mod), triggeredBy(PERSONAL_FILE))

Extern(Consulting, File(consulting.mod), triggeredBy(START_CONSULTING))
```