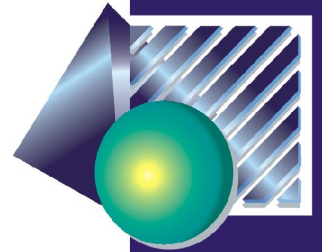


# ClassiX®

# ClassiX®

## Tutorial 4

## Content


1. INTRODUCTION ....................................................................................................3
   1.1 Conditions for this tutorial ....................................................................................3

2. UPGRADING THE APPLICATION WITH APPSWAREHOUSE® MODULES.......4
   2.1 Selecting matching modules (not necessary for the tutorial)...........................................4
   2.2 The object model.........................................................................................................5
      2.2.1 Class CX_USER.....................................................................................................5
      2.2.2 Class CX_CYBER_ENTERPRISE...........................................................................7
   2.3 Database with layer 0 and 1.........................................................................................8
   2.4 The upgraded application ............................................................................................8
      2.4.1 New batch main program with login- and user administration routine.....................9
      2.4.2 Using further modules .........................................................................................13

3. SECURITY.................................................................................................................14
   3.1 Integrating a module for access rights...........................................................................14

4. REVIEW ....................................................................................................................19

5. PROGRAM LISTINGS.................................................................................................20
   5.1 Project file person4.cxpc.............................................................................................20
   5.2 Module person.mod.....................................................................................................21
   5.3 File tutorial4.ext..........................................................................................................22
   5.4 File classix.ini.............................................................................................................22

# 1. Introduction

Tutorial 1 gave an introduction to the InstantView® basics. By the end of the tutorial, we had built a small application. In the following tutorial, we have split this application into modules and upgraded them with the inheritance concept.
Every now and then there was the promise, that this shown way was only an exception. Normally, an application develops from already existing AppsWarehouse® modules.
In tutorial 4 you will see that we keep our promise: We will upgrade the exemplary application with selected AppsWarehouse® modules.
The way this task has been chosen, makes some of the things you've learned about logical database structure in tutorial 3 become important again; now from a practical point of view. At the end, there will be an overview of information about access rights in the ClassiX® 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 or the provided zip archive *cx_Freeware.zip* (see tutorial 1).
The file *person4.cxp.end* shows the final state of the exemplary application in this tutorial.
Simply removing the ending ".end" we can already run the final state with `tutorial4.bat` (for demonstration purposes).

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

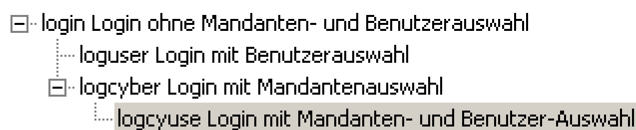
# 2. Upgrading the application with AppsWarehouse® modules

In this tutorial you will get to know and implement some prefabricated AppsWarehouse® modules. The exemplary application shall be upgraded with a login routine to make the user login at the start. To make this possible, potential users of the application have to be registered in the database.
Let's now imagine our application administers (person) data for a particular client and later eventually for further clients.

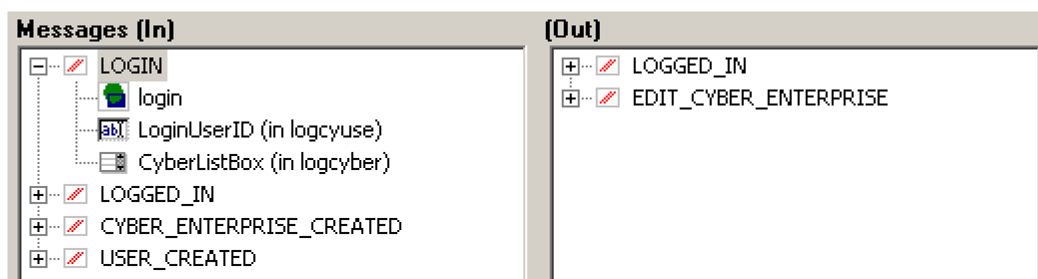
## 2.1 Selecting matching modules (not necessary for the tutorial)

So what are we looking for? Modules for **system login** and for **user** administration in the database.
InstantView® modules can be described with semantic classifiers (see Basics – (tutorial 0) – paragraph 2.3.3.4). All AppsWarehouse® modules are identified like this and therefore they can also be searched. The browser is basically just a module from the AppsWH directory, too. It can be called via object inspector in the tool menu or per *SendMsg(BROWSE_APPSWH).*

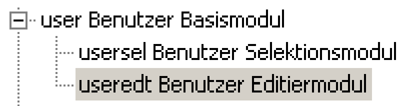
We search for "system login" and find



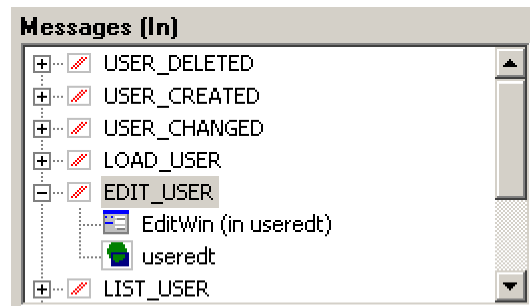

the module *logcyuse* as a suitable module for our task (client selection and user login). The browser also shows the message interface:




whereas the messages accepted by the module (left page) are relevant for the call.
Searching for the keyword "user" provides a whole lot of modules; the short description (synopsis – 2.3.3.4 in Basics (tutorial 0)) makes the module *useredt* come up as a suitable result:




with the input messages:




We will now integrate the modules *logcyuse* and *useredt.*

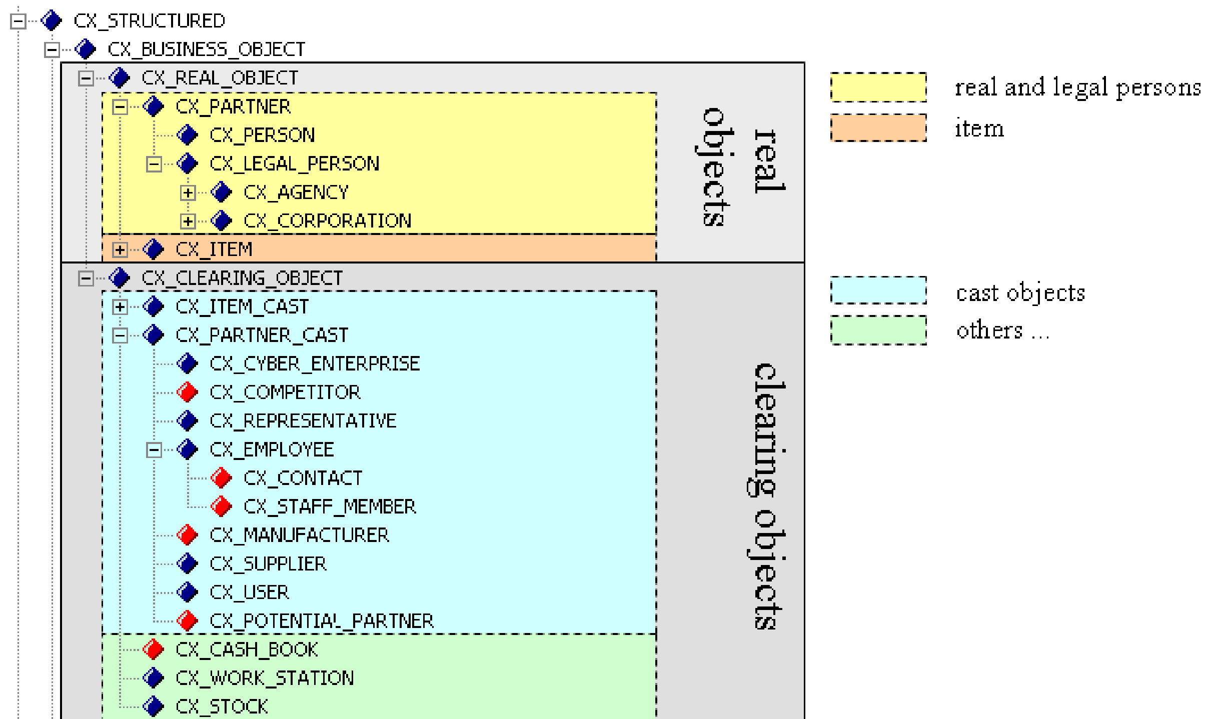
## 2.2 The object model

Although we don't really care about the internal functionality of the modules *logcyuse* and *useredt*, it makes sense to no a bit about what modules actually do. With which CyberEnterprise® objects do we work in the "login with client- and user selection" and in the "user editing module"?

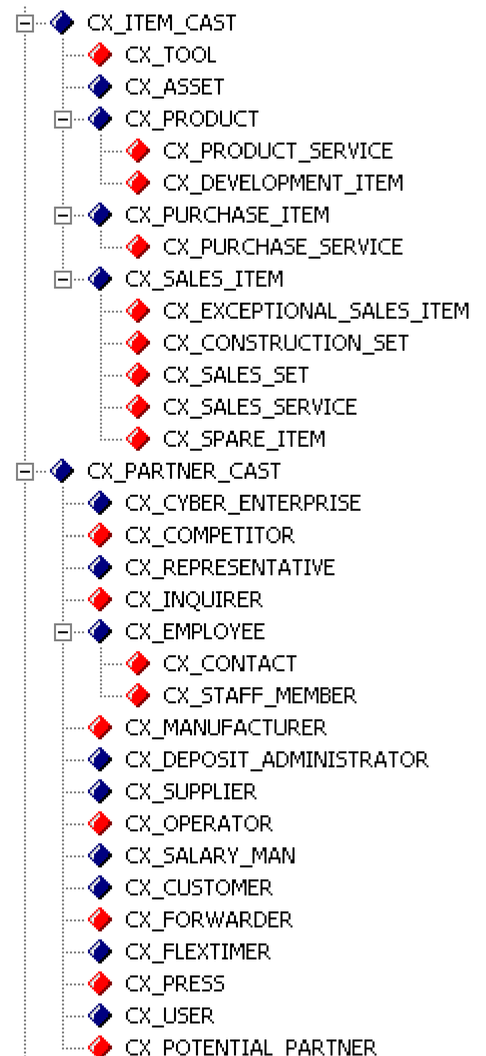
### 2.2.1 Class CX_USER

A user is a person, but not all administered persons in our exemplary application have to be users of the application. "User" is a role which can apply to a person[1].
Users are mapped with objects of the class CX_USER. Other roles to play are customer, supplier, employee – modelled by the classes CX_CUSTOMER, CX_SUPPLIER, CX_EMPLOYEE. The connection between CX_PERSON and CX_USER is worth looking at more closely than it would be necessary just for the integration of the login module. This is about one of the object model principles on which the CyberEnterprise® is based on.

[1] "playing a role" is maybe a bit too much associated with comedies and tragedies. The statement: Object **O** has the role **X** and **Y** means, that **O** is an **X** and a **Y** from a billing point of view.



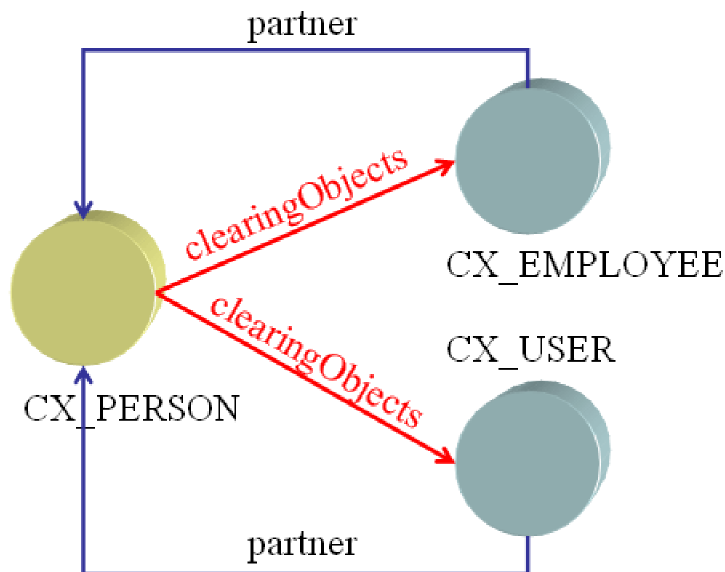
picture: real objects – billing objects[2]




The picture shows the division into objects for the description of real existent things and billing objects.
Relevant for us are natural persons and legal entities (derivated from the class CX_PARTNER) and the real items or services occurring in a company's production (CX_ITEM) as parts of the real objects.
The two first ones refer to the company as customers, suppliers, employees etc.; the latter ones are products or services for sale, but also purchased parts, spare parts, …
On the right, we see the diversity of possible roles for CX_PARTNER and CX_ITEM objects:

[2] to make it easier to follow, some classes are missing in this picture

Real and role objects are connected through a series of $m$-1-relations. The role object refers to exactly one real object. This real object though can appear in different roles:




Module *useredt* creates CX_USER objects and connects them to objects of the class already CX_PERSON which already exist in the database.

## 2.2.2 Class CX_CYBER_ENTERPRISE

The selection of a client at the login refers to an object of the class CX_CYBER_ENTERPRISE, another role for a natural person or a legal entity (see picture on pg.4). The class name means to say, it is a master object integrating all objects that realise a mapping of the company[3] from a business point of view.
The class CX_CYBER_ENTERPRISE is closely connected to the layer concept of the database (see 2.1.2 in tutorial 3). The CX_CYBER_ENTERPRISE object "knows" the layer, which contains the company data. One method of database organisation is to keep layer 0 free and to assign layers 1, 2, ..., $n$ to the clients. Only the CX_CYBER_ENTERPRISE objects are visible in all layers. CLASSIX.INI – the standard initialisation file of the ClassiX®-systems is organised just like this, and from now on we don't want to use `tutorial.ini` anymore. Instead we will use `classix.ini`.

[3] To which the CX_CYBER_ENTERPRISE object refers

## 2.3 Database with layer 0 and 1

What should we do with the test-database?[4] If you look at `tutorial.ini`, you will notice

- that there is only one layer 0 and
- that there are no meta-data for the classes CX_USER and CX_CYBER_ENTEPRISE

Let's now imagine, `tutorial.ini` is upgraded class-, file- and storage directives for the classes mentioned above.
With the command line program

```
cxgntosd -U -Itutorial.ini
```

would update[5] the database; which means that missing segments, REP collections[6] would be generated.
What happens, if we just call

```
cxgntosd -U -Iclassix.ini
```

instead?

We compare the meta-information for the class CX_PERSON:

Tutorial.ini

```
Class(CX_PERSON,  12601, person, CX_PARTNER)
File(person, person)
Storage(person,  DB(1),  "personS",   EP("personL0"), CSeg("cs.person"),
                                              Garbage("geps", "gcs"))
```

Classix.ini

```
Class(CX_PERSON, 12601, person, CX_PARTNER, Docu(14500))
File(person, empty, person)
Storage(person, DB(1), "personS", EP("personL0"), CSeg("cs.person"),
                                              Garbage("geps", "gcs"))
```

and we notice, that segments and REP collections carry the same names. As long as long as it concerns the class CX_PERSON, we can start our application with `tutorial.ini` and see the objects in layer 0. Starting with `classix.ini`, we find the objects in layer 1.
We see, it is possible to update the database with `classix.ini`.

> The meta-information describes a logical view of the physical database(s).Different meta-data (= different initialisation files) can refer to the same database(s).

## 2.4 The upgraded application

In this paragraph, our program will be upgraded with expedient safety functions, such as user login, and demonstrate the integration of prefabricated modules.

---

[4] if you have been working with the demo version, the question is certainls only of theoretical interest
[5] -U stands for update, in contradiction to –C (create), which helps create a new database
[6] and if required further physical databases, too

### 2.4.1 New batch main program with login- and user administration routine

We want to “put” the exemplary application – after tutorial 3 – in a batch main program. After starting the computer, this shall enforce a login (with client selection). At first, we turn `person2.cxp` int a module `person.mod`:

```
Module(PersonalFile)
[
  Msg(PERSONAL_FILE, SHOW_ALL_PERSONS, PARENT_CHILD_RELATIONSHIP,
      FAMILY_TREE_1, FAMILY_TREE_2, SHOW_QUERY_RESULT)

  Msg(PERSON_SELECTED, PERSON_CHANGED)       Var(person)

  Define(EnterStateI)  // Button "Save" deactivated
     Lock(,SaveBtn);

  Define(EnterStateII)  // Button "Save" deactivated &
                        // alert to detect user input
     EnterStateI Alert(StartWindow);

  Define(EnterStateIII)  // Button "Save" is active
     Unlock(,SaveBtn);

  PERSONAL_FILE: OpenWindow(StartWindow, 1)
]
// Window to display and edit 1 person object

Window(StartWindow, ACCEPT_DROP, 0, 0, 600, 110,
T("Personenverwaltung","personal file"))
[
   .  .  .
```

Without the line

```
#include "tutorial.ext"
```

and additionally comment out the lines

```
Prompt(310, 6, T("Rufname:","Nickname:"))
String(CX_PERSON::nickName, 410, 6, 150)
[
 ALTERED: EnterStateIII
]
```

since they are not defined in the classix.ini.

As a new batch main program, we introduce `person4.cxp`.
From analysis with the AppsWarehouse® browser or from the documentation about the module *logcyuse,* we know, that the login routine gets started with *SendMsg(LOGIN)*.
This shall happen exactly once, straight after starting the ClassiX® system. This is why *SendMsg(LOGIN)* gets connected with the main module event INITIALIZE.

**1. Important:** Here, the main module **needs** to be `Module(GLOBAL)`.
Reason: After a (successful) login, all modules except for GLOBAL, are reinitialised.

With this, the module windows get closed. InstantView® ends though, if no More window is active.

If there is no `Module(GLOBAL)` with at least one window definition, we get an impression of a simple crash.

**2. Important:** The definitions for the implemented standard libraries Bibliotheken *ivFunctions.mod* and the global variables (*cyberEnterprise, user*) have to be declared in the main module `Module(GLOBAL)`, so they are known to the system.

```
#include "tutorial4.ext"
#include "classix.ext"

Module(GLOBAL)
[
  Msg(LOGIN, EDIT_USER, PERSONAL_FILE)

  GlobalVar(g_defaultLocale, cyberEnterprise, user)
  #include "ivFunctions.mod"
]

Window(win, BITMAP_SIZE, 1, 1, 1200, 30, T("Beispiel Anwendung . . .
[
  INITIALIZE: SendMsg(LOGIN) // to force login
]
{
   Menu
   {
     Item(T("Personenverwaltung", "Personal File"))
     [
       SELECT: SendMsg(PERSONAL_FILE)
     ]
     Item(T("Anwender", "User"))
     [
       SELECT: SendMsg(EDIT_USER)
     ]
   }
 }
```

The file `tutorial4.ext` equals `tutorial.ext` , upgraded with the connection to the new module Modul *PersonalFile* in `person.mod`:

```
Msg(PERSONAL_FILE, SHOW_ALL_PERSONS, PARENT_CHILD_RELATIONSHIP,
    FAMILY_TREE_1, FAMILY_TREE_2, SHOW_QUERY_RESULT)
. . .

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

Furthermore, we integrate `classix.ext`. A file with external references to all modules of the standard AppsWarehouse®.

In tutorial4.bat you should refer to the new `tutorial4.ini`:

```
@echo off
if -%CX_PROJECT_INITIALS%- == -- SET CX_PROJECT_INITIALS=TU
if -%CX_PROJECT_NAME%- == -- SET CX_PROJECT_NAME=TUTORIAL4

SET CX_ROOTDIR=\Classix\Evaluate
SET CX_FREEWARE=\Classix\Freeware
SET CX_BIN=%CX_ROOTDIR%\BIN\4.4
SET OS_SCHEMA_PATH=%CX_BIN%

SET CX_BITMAP=%CX_FREEWARE%\bmp;%CX_ROOTDIR%\bmp

SET CX_PATH=%CX_FREEWARE%\AppsWH\Freeware\Tutorial4;%CX_FREEWARE
%\AppsWH\Freeware;%CX_ROOTDIR%\AppsWh
SET CX_PATH=%CX_PATH%;%CX_FREEWARE%\System\Freeware;%CX_FREEWARE%\System
SET CX_PATH=%CX_PATH%;%CX_FREEWARE%\Projects\Tutorial4

SET CX_SYSTEM=%CX_FREEWARE%\system\Freeware;%CX_FREEWARE%\system;
%CX_ROOTDIR%\System
SET CX_DATABASE=%CX_FREEWARE%\projects\tutorial.cxd
SET CX_SYSTEM_DB=%CX_DATABASE%

IF "%CX_LICENSE_FILE%" == "" SET CX_LICENSE_FILE=%CX_ROOTDIR
%\Projects\evaluate.lic

start %CX_BIN%\cx_osr.exe person4.cxp -I tutorial4.ini
```

Starting the application – with `person4.cxp` in the form listed above – the login window appears and asks to login[7]. It is possible to “cheat”, though, ignoring the login request. The login window is not modal, and all other menu items are accessible. This has to change! After a successful login – and in this case only – the login module sends the message LOGGED_IN (see message interface on page 4).
All menu items get blocked at first and they are only enabled with LOGGED_IN:

```
Module(GLOBAL)
[
  GlobalVar(g_defaultLocale, cyberEnterprise, user)
  #include "ivFunctions.mod"

   Msg(LOGIN, LOGGED_IN, EDIT_USER, PERSONAL_FILE)
]
Window(win, BITMAP_SIZE, 1, 1, 1200, 30, T("Beispiel Anwendung . . .
```

[7] You must have started the application once before entering a client.

```
[
   INITIALIZE: SendMsg(LOGIN) // to force login
]
{
   Menu
   {
     Item(NON_SELECTABLE, T("Personenverwaltung", "Personal File"))
     [
       SELECT: SendMsg(PERSONAL_FILE)
       LOGGED_IN: Unlock
     ]
     Item(NON_SELECTABLE, T("Anwender", "User"))
     [
       SELECT: SendMsg(EDIT_USER)
       LOGGED_IN: Unlock
     ]

   }
 }
 . . .
```

At the initial login, there is no user, yet. The login is possible without username and password. We get asked to enter data for a client (so an object of the class CX_CYBER_ENTERPRISE and another object CX_CORPORATION can be generated and connected to each other). For the layer, we state 1. As soon as we want to create one or more users – as soon as there are CX_USER objects – the login routine requires a username to be entered.

### 2.4.2 Using further modules

With `classix.ext`, we have access to the entire standard AppsWarehouse®. We will make use of it now. We also want to call the object inspector in the exemplary application and see all system information that has been discussed in tutorial 3 (meta-classes, database layout logical/physical, query- and index manager, query functions…).

```
Module(GLOBAL)
[
  Msg(LOGIN, LOGGED_IN, EDIT_USER, PERSONAL_FILE,
      INSPECT_OBJECT, START_MINI_WORKBENCH)
  GlobalVar(g_defaultLocale, cyberEnterprise, user)
  #include "ivFunctions.mod"
]

Window(win, BITMAP_SIZE, 1, 1, 1200, 30, T("Beispiel . . .
[
  INITIALIZE: SendMsg(LOGIN)
]
{
  Menu
  {
    Item(NON_SELECTABLE, T("Personenverwaltung", "Personal File"))
    [
      SELECT: SendMsg(PERSONAL_FILE)
      LOGGED_IN: Unlock
    ]
    Item(NON_SELECTABLE, T("Anwender", "User"))
    [
      SELECT: SendMsg(EDIT_USER)
      LOGGED_IN: Unlock
    ]
    Item(NON_SELECTABLE, "System")
    [
      LOGGED_IN: Unlock
    ]
    {
      Item("Object Inspector")  // object inspector
      [
        SELECT: SendMsg(INSPECT_OBJECT)
      ]
      Item("Mini Workbench")  // all other system information
      [
        SELECT: SendMsg(START_MINI_WORKBENCH)
      ]
    }
  }
}
```

# 3. Security

The obligation of a user login gets another meaning, when such objects are connected to the CX_USER object, with which access rights get controlled to data and program functions.
You find a complete description of security objects in the InstantView® documentation.
As long as no security object applies in the ClassiX® system, there are no revisions in the system. Everything is allowed! As soon as a security object applies – and this happens during login, if the CX_USER object refers to such an object – all access rights will be revised.
If the access rights allow for this, another security object can be applied. But it is not possible to return to the initial working state without "security".

The access rights referring to the data can be declared for

- a class,
- a class data field[8],
- a specific (persistent) object and
- for a data field[7] of a specific (persistent) object.

Access to transient objects is not monitored.

Program functions get locked by banning certain messages.

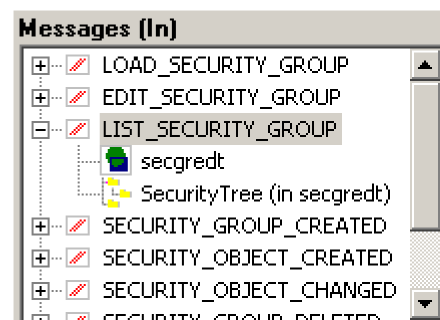
Since only one security object is active at a time, security objects can be merged to a group. We want to create such a group and assign it to a CX_USER object.

## 3.1 Integrating a module for access rights

Using the AppsWarehouse® browser (key word "access groups"), we find the module

secgroup Zugriffsgruppen Basismodul
secgredt Zugriffsgruppen Editiermodul
secgrsel Zugriffsgruppen Selektionsmodul

for editing the security objects, and we start it with the message LIST_SECURITY_GROUP.



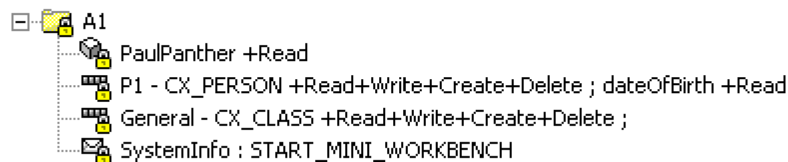

```
Module(GLOBAL)
```

[8] more precisely: for an access expression

```
[
  Msg(LOGIN, LOGGED_IN, EDIT_USER, PERSONAL_FILE, INSPECT_OBJECT, ...
  Msg(LIST_SECURITY_GROUP)

  GlobalVar(g_defaultLocale, cyberEnterprise, user)
  #include "ivFunctions.mod"
]

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

. . .
```

Everything else already includes `classix.ext`. We can start the application, edit access rights and assign to a user.

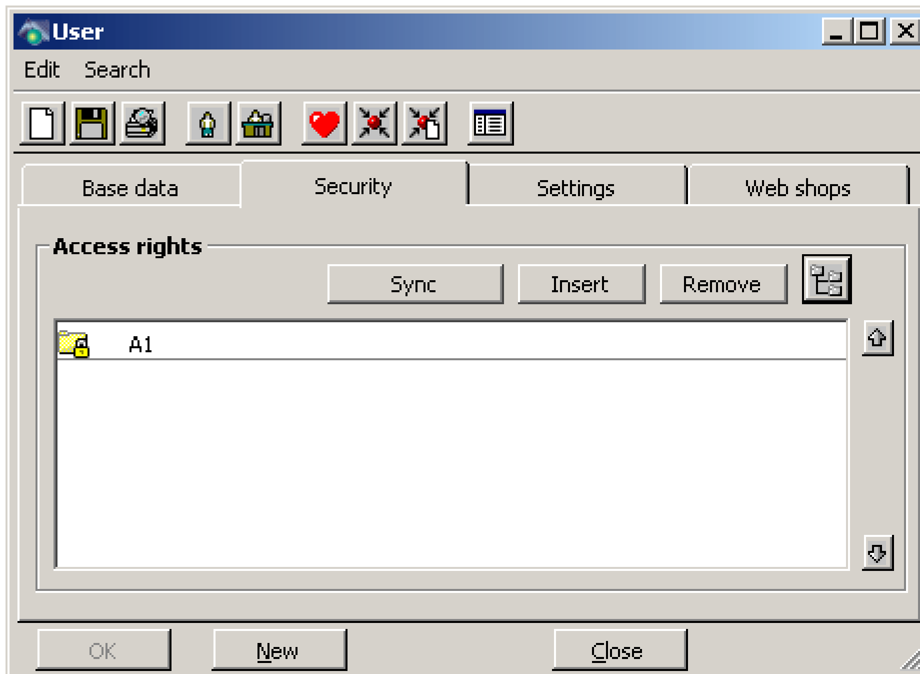
An example:




This group forbids any changes for a specific object. For all other objects of the class CX_PERSON only the writing access to the data field *dateOfBirth* is forbidden.
The following line means that that everything is allowed for all remaining objects (they are all derivated from CX_CLASS). This security object is necessary, and the object order is not optional, either, since the test of access rights follows a best-match-method[9].

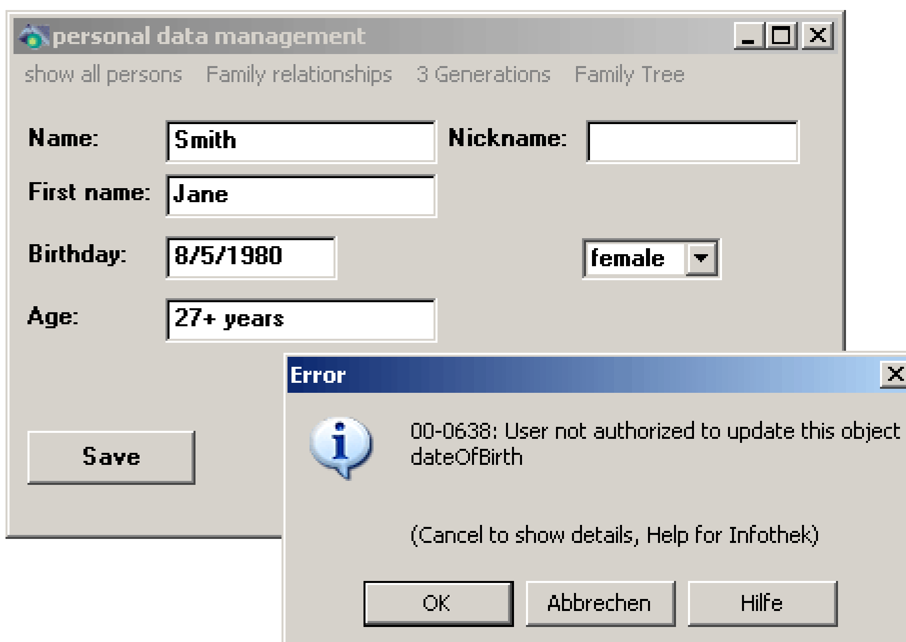
[9] The security objects can also be be listed in a different way, specifying the exceptions to become a security object. The standard AppsWarehouse® modules don't make use of this possibility.

The order of message security objects among each other as well as in relations to the data-oriented security objects is irrelevant.




Here, the message START_MINI_WORKBENCH gets locked.
We assign the group A1 to a user (AA → Anna-Maria Ameise), and the security object A1 applies after a following login as a user AA in the ClassiX® system. The access restriction listed above become active.

What happens if we change a person's date of birth and cause the statement *DrainWindow* with the button "save"? The writing access to the data field *dateOfBirth* was forbidden for all objects of the classes CX_PERSON!
We receive a problem report, and the same thing happens, if we try to change any data field (e.g. the name) for the selected object (Paul Panther).



As useful and reasonable as a control of access rights is – problem reports of this kind are demotivating for the user. It shouldn't be possible to make changes in the entry field *dateOfBirth* to start with! The statement *AdjustView* conducts an accordant adjustment of the entry window.
The same thing is valid for blocked messages. It seems somehow "unfair", if one program offers an action, and as soon as the user selects the accordant menu, it says, he or she has just tried something "illegal"! If a message is blocked can be found out with the statement *TestMsg* and we change the batch main program `person4.cxp` as follows:

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

And in `person.mod` we only have to insert

```
Window(StartWindow, ACCEPT_DROP, 0, 0, 600, 110,
T("Personenverwaltung","personal file"))
[

  DROP: Drop SendMsg(PERSON_SELECTED)
  PERSON_SELECTED: -> person, person FillWindow, person AdjustView
                   EnterStateII
]
{
    . . .
}
```

# 4. Review

For the first time, we could achieve much more functionality in our exemplary application, just by making small changes in the InstantView® code. Working with the login routines of the AppsWarehouse®, is our motivation to break open the restriction on
CX_PERSON and to take a first look at the ClassiX® object model. At the end it was shown how to administer data access rights and program functionality in the ClassiX® system.

**What have we learned:**

- It has been described, where to find AppsWH standard modules and how to select them best.
- The first approaches of an access control with user administration have been demonstrated.
- We have taken a closer look at the database structure.

# 5. Program listings

## 5.1 Project file person4.cxpc

```
// person4.cxp - Tutorial IV final state
// Copyright (c) 2007 ClassiX Software GmbH, Hamburg (Germany)
// All rights reserved

// $Header: //iv/Freeware/Projects/Tutorial4/person4.cxp#4 $ $DateTime: 2007/09/14 09:53:25 $ $Author: Bohl $


#include "tutorial4.ext"
#include "classix.ext"

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

  // ---------------------------------------------------------------------------
  // - global functions and variables ------------------------------------------

  GlobalVar(g_defaultLocale, cyberEnterprise, user)
  #include "ivFunctions.mod"
]

Window(win, BITMAP_SIZE, 1, 1, 1200, 30, T("Beispiel Anwendung (Tutorial IV)", "Sample Application (Tutorial IV)"),
"tutorial4.bmp")
[
  INITIALIZE: SendMsg(LOGIN)
]
{
  Menu
  {
    Item(NON_SELECTABLE, T("Personenverwaltung", "Personal File"))
    [
      SELECT: SendMsg(PERSONAL_FILE)
      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
      ]
    }
  }
}
```

## 5.2 Module person.mod

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

// $Header: //iv/Freeware/AppsWh/Freeware/Tutorial4/person.mod#2 $ $DateTime: 2007/09/18 10:23:04 $ $Author: Bohl $


Module(PersonalFile)
[
  Msg(PERSONAL_FILE, SHOW_ALL_PERSONS, PARENT_CHILD_RELATIONSHIP, FAMILY_TREE_1, FAMILY_TREE_2, SHOW_QUERY_RESULT)

  Msg(PERSON_SELECTED, PERSON_CHANGED)
  Var(person)

  Define(EnterStateI)  // Button "Save" deactivated
    Lock(,SaveBtn)
  ;

  Define(EnterStateII)  // Button "Save" deactivated &
                        // alert to detect user input
    EnterStateI Alert(StartWindow)
  ;

  Define(EnterStateIII)  // Button "Save" is active
    Unlock(,SaveBtn)
  ;

  PERSONAL_FILE: OpenWindow(StartWindow, 1)
]
// Window to display and edit 1 person object

Window(StartWindow, ACCEPT_DROP, 0, 0, 600, 110, T("Personenverwaltung","personal file"))
[
  DROP: Drop SendMsg(PERSON_SELECTED)
  PERSON_SELECTED: -> person, person FillWindow, person AdjustView
                   EnterStateII
]
{
  Menu
  {
    Item(T("alle Personen", "show all persons"))
    [
      SELECT: SendMsg(SHOW_ALL_PERSONS)
    ]
    Item(T("Verwandtschaftsbeziehungen", "Family relationships"))
    [
      SELECT: SendMsg(PARENT_CHILD_RELATIONSHIP)
    ]

    Item(T("3 Generationen", "3 Generations"))
    [
      SELECT: SendMsg(FAMILY_TREE_1)
    ]
    Item(T("Stammbaum", "Family Tree"))
    [
      SELECT: SendMsg(FAMILY_TREE_2)
    ]
  }

  Prompt(10, 6, T("Name:"))
  String(CX_PERSON::name, TOOLTIP("Name der Person", "Name of the Person"), 110, 6, 191)
  [
    Var(partialName, result)

    SELECT:  GetValue( ,name) -> partialName
             partialName "z" + partialName
             "name >= %s & name <= %s" Find(CX_PERSON) -> result
             result Cardinality
             case 0: // nothing found
                    ClearWindow
                  1: // 1 object found
                    0 result GetElement -> person, person FillWindow
                  default: // more than 1 object found
                                 result SendMsg(SHOW_QUERY_RESULT)
             endCase

    ALTERED: EnterStateIII
  ]

  Prompt(10,17,T("Vorname:", "First name:"))
  String(CX_PERSON::firstName, 110, 17, 191)
  [
    ALTERED: EnterStateIII
  ]

  Prompt(10, 30, T("Geburtstag:", "Birthday:"))
  Date(CX_PERSON::dateOfBirth, 110, 30, 120)
  [
    Var(tmp) // Hilfsvariable

    ALTERED: EnterStateIII
    NON_CURRENT:
    SELECT: CreateTransObject(CX_PERSON) -> tmp
                tmp DrainWindow, tmp Call(Age) PutValue(, age)
  ]

  Prompt(10, 43, T("Alter:", "Age:"))
```

```
  String(CX_PERSON::Age()~age, NO_DRAIN, 110, 43, 191)

  Enumeration(CX_PERSON::sexEnum, 405, 30, 100, 35)
  [
    ALTERED: EnterStateIII
  ]

  Button(SaveBtn, NON_SELECTABLE, 10, 70, 120, T("Speichern","Save"))
  [
    SELECT: person ifnot { CreatePersObject(CX_PERSON) -> person }
                   person DrainWindow, EnterStateI
                   person SendMsg(PERSON_CHANGED)
  ]
  Button(220, 70, 120, T("Neu","New"))
  [
    SELECT: ClearWindow, NULL -> person, EnterStateII
  ]
}
```

# 5.3 File tutorial4.ext

```
// tutorial4.ext - link to external modules - Tutorial IV
// Copyright (c) 2007 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)


 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))
```

# 5.4 File classix.ini

```
// ClassiX.ini   ClassiX initialization file
// Copyright (c) 2007 ClassiX Software GmbH, Hamburg (Germany)
// All rights reserved

// $Header: //iv/main/system/classix.ini#3 $ $DateTime: 2002/02/27 11:12:05 $ $Author: Duesterhoeft $

// Definition of specifier, slots and data members
#include "classix.dic"
// if persistent instead of #include "classix.dic"
//   SystemDB(CX_SYSTEM_DB)

MetaInfo
// optional: DLLs with C++ functions to be called by CX_FORMULA
//   FormulaCppDLLs(ClassiX)
// Loaction of schema database
   Schema(CX_SCHEMA)
// optional: location of ClassiX query server process
//   QueryServerDB(CX_SERVER)
// Checking of illegal pointers activated
   CheckIllegalPointers
// Mapping of database
   Database(1, CX_DATABASE)
// Definition of classes, pseudo classes, files and storages
#include "classix.odb"

// Include own classes at this point, e.g.
// PseudoClass(MY_OWN_CLASS,       20000, myOwnClass, CX_CLASS)
// File(myOwnClass,                empty, myOwnClass)
// Storage(myOwnClass,             DB(1), "myOwnClassS", EP("myOwnClassL0"), CSeg("cs.myOwnClass"), Garbage("geps", "gcs"))

// Help files directory
   Help("CX_ROOTDIR\SYSTEM\IV###.hlp")
// Editor
// Please set enviroment variable of your favourite editor:
// SET CX_EDITOR=c:\cw32\cw32 %%s -g%%d
// OS/2 editor
// OS2Editor("start epm %s '%d'")
// Windows editor
   WinEditor(CX_EDITOR)
```