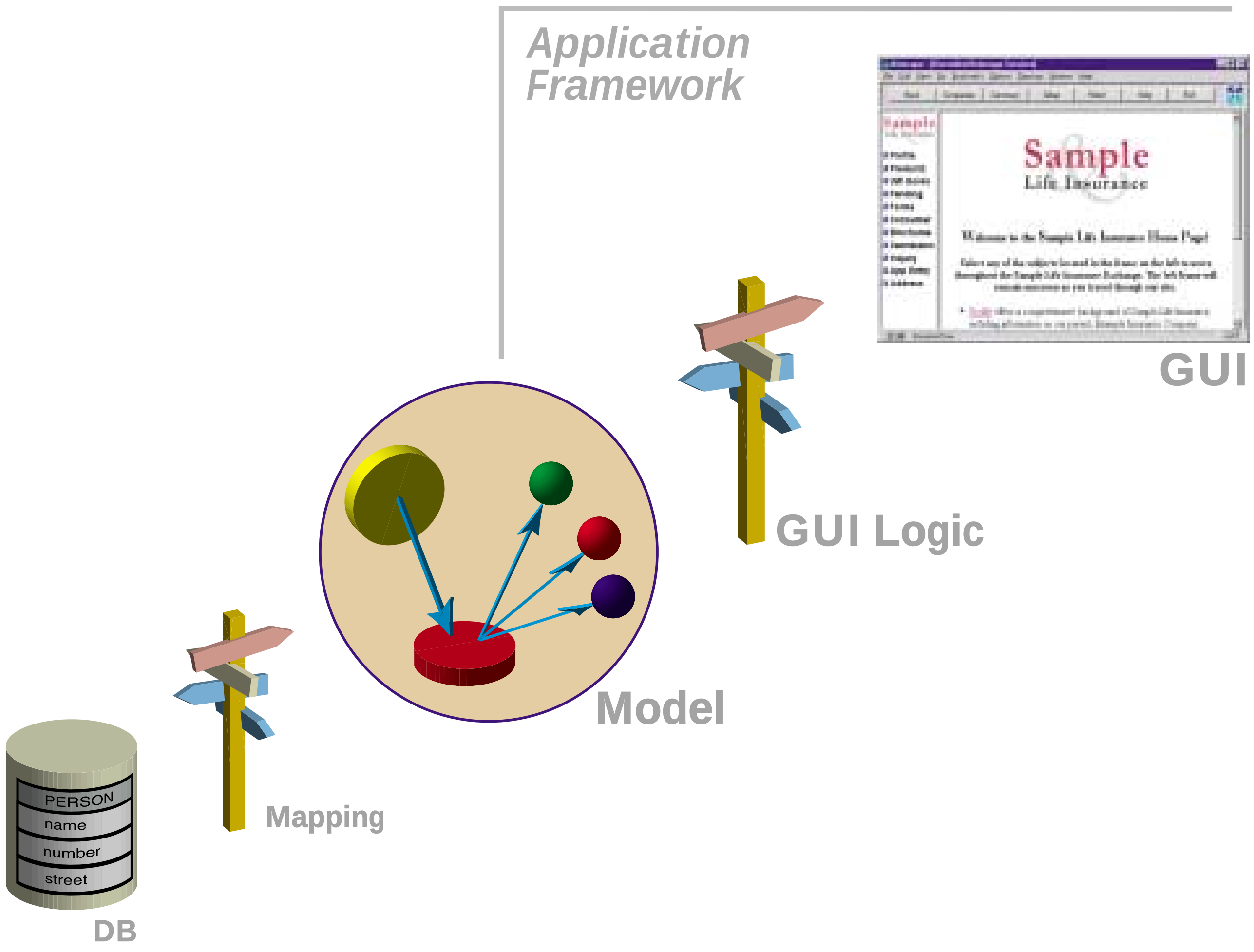
Application Framework

User’s Guide

For Frameworks Release V5.0



Copyright and trademarks

Table of Contents

Documentation overview

Who Should Read this Manual / What You Should Already Know

The Purpose of this manual is to show the concepts, development tools, and programming techniques used in the Application Framework and is intended to be used mainly by Application developers. This manual describes the necessary base functions in the framework and enhances the developer's practical abilities and skills. For Managers and Project-leaders, this manual will provide background information needed for use in decision making.

This manual assumes that you have some knowledge of or experience with the Smalltalk development language and its basic concepts and programming techniques. It is also recommended for the reader to have a fundamental understanding of graphical user-interfaces (GUIs).

To install the Application Framework, you should also have a working knowledge of IBM VisualAge for Smalltalk, and should be familiar with the Object Behavior Framework (which is documented in its own manual).

will guide you through a program creation process from scratch. It will help you in extending your knowledge about fully fledged applications by adding features step by step.

For training in association with the Framework, Mynd offers a wide variety of ready-to-use classes for Smalltalk programming, object-oriented analysis and design methods, and project management tools.

Recommended Chapters

The organization of this manual follows the concept that there are several ways to read our manual. Some chapters are intended to be read by programming novices, while other chapters are to be used by experienced or senior programmers who want to get most out of the Framework. Therefore we recommend You to decide which chapters could be of interest especially for Your needs and skills.

For the Smalltalk and OO programming novice

- Start with **'Tutorial' (page 45)**.

For the designer or architect of an application

- Start with **'Concepts' (page 23)**.
- Optionally start with **'Tutorial' (page 45)** (to learn the basics of the Framework).
- Continue **'Programming Reference' (page 93)**.
- Continue with **'Customizing' (page 217)**.
- Finish **'Creating Runtime Executables for Distribution' (page 261)**.

For the implementer of a smaller application

- Start with **'Concepts' (page 23)**.
- Optionally read **'Concepts' (page 23)**.
- Continue with **'Programming Reference' (page 93)**.
- Finish with **'Creating Runtime Executables for Distribution' (page 261)**.

For the implementer of a complex application

- Start with **'Concepts' (page 23)**.
- Optionally read **'Concepts' (page 23)**.
- Continue with **'Programming Reference' (page 93)**.
- Continue with **'Customizing' (page 217)**.
- Finish with **'Creating Runtime Executables for Distribution' (page 261)**.

Mynd on the Web

Visit the Mynd web-site at HTTP:\\WWW.MYND.DE

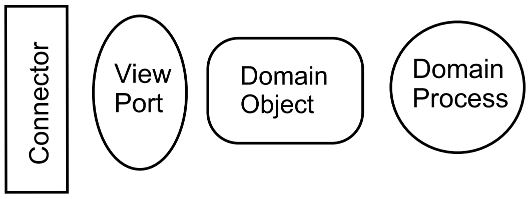
Terms and Symbols of the Manual

To help you interpret the information in our printed manuals:

- Italic indicates a
- Bold **emphasizes** a word or phrase

- Italic indicates and
- Monospace indicates `passages or Smalltalk code`

The following symbols are used for the most important Application Framework entities:



Release Notes

Version 5.0

1. Mnemonic bug fixed

Assume that a child view is opened from the parent view. In the previous version, if you clicked on the parent view surface, nothing happened (it appeared as if the child retained the focus). However, the parent actually regained the focus. Thus, you would assume that the child would process any commands, when actually the parent would process any input.

This was caused by the fact that mnemonics for certain controls were transferred to the parent view. This has been fixed.

2. Help file call fix

In the previous version, the event generated when the F1 key was pressed (to show help) was always trapped by the Application framework help. If you did not define a <HelpText> subaspect, then the Application framework returned the F1 event to VA.

3. Update performance

In this version, when an immediate update <updateItemPresentation> for an AbtContainterDetailsView is made, only the columns with aspect names for the changed aspect are updated. Thus, the update optimization is not executed for a DeferredUpdate. An immediate update is executed when, for example, the ViewPort method <wantsImmediateViewUpdateOf:> returns true.

4. Required maps changed

The required maps for

- micApplication Service - Multi Language Persistence Interface Development ;
- micApplication Service - Multi Language Persistence Interface Runtime

were changed. They now reference parts of the Persistence Framework.

5. Web Interaction Feature no longer supported

The Web Interaction feature is no longer a part of the Application Framework.

6. VisualAge 5.01 compatibility

Frameworks (including Application Framework) now run under VisualAge 5.01.

Version 4.3

1. Inserting MicFwUndefinedObject in a DropDownList under OS/2 causes no error

Under OS/2 a MicFwUndefinedObject can be inserted in a DropDownList with no errors.

In the previous version this caused update errors.

2. Web interaction not supported

The Application Framework connection to the IBM WebConnection is not supported in this version.

This connection was provided as a preview in the previous version; however, performance with the WebConnection architecture in realistic web implementations was less than expected.

Version 4.2

1. View can be opened as a Prompter

A View can be opened as a Prompter.

Processing is halted while the view is being opened.

```
MicFwDomainProcess >> openViewAndWaitUntilRemoved: aViewClass
MicFwDomainProcess >> openViewAndWaitUntilRemoved
```

2. MLS automatic conversion of entries disabled

MLS no longer supports automatic conversion of entries. One reason for this change is that the complex application rules required for the feature are eliminated.

In the previous version, the following entries would be converted as shown:

```
'1.00.000,00'  to '1.000.000,00'
```

```
'1.24.567,00' to '1.240.567,00'
'1..000,00' to '1.000.000,00'
```

The above entries now cause an MLS-Converter-Error.

3. ForegroundColor / BackgroundColor can be set for GroupBox

The ForegroundColor and BackgroundColor can be set for a GroupBox. This was not possible in the previous version.

4. DragAndDropControlImplementation and DragAndDropContrl can be subclassed

DragAndDropControlImplementation and DragAndDropControl can now be subclassed. This supports custom Drag&Drop implementations. This may be required when using custom controls with Drag&Drop.

Hooks:

```
MicFwDomainProcess >> defaultDragAndDropControlImplementation
MicFwDragAndDropControlImplementation >> getModelViewPortFromPlatformItem:
anItem
```

5. Fix for VA CascadedSubmenu enabled status error

If a dynamic Menu that contained CascadedSubmenu's was switched from disabled to enabled, VA would set the Enabledstatus for the CascadedSubmenu and any of its submenus to false. The MenuItem's were also not refreshed correctly. The Framework now provides a fix for this error.

6. Deleting via a dynamic Menu does not cause an error

A dynamic context menu can be opened for an item (such as IconTreeItem) and the item deleted via the menu.

This avoids an error which occurred in the previous version.

7. Column accessor is MicFwBaseModelAccessor

The Column accessor is now MicFwBaseModelAccessor.

In the previous version a Column (static or dynamic) was connected via a MicFwGroupModelAccessor, which caused problems in certain situations.

This change should have no effect on existing applications.

The change in accessor is possible because the DomainProcess's for a Column are not dynamically changed.

8. InformOfChanges executed after localize complete

AppFw causes an <informOfChanges> message to be sent for each new LocaleEntity. The <informOfChanges> messages are now executed only after localize is complete.

In the previous version an <informOfChanges> message was sent and executed during the construction (localize) of the MLS-Context. This led to updates in the AppFw which could cause a renewed access of MLS. During MLS access localize of the of the Context is not complete and thus the Context was inconsistent.

Version 4.0

1. New Domain Processes Browser

This release contains a new version of the DPB. The new capabilities of the browser are described in **'Domain Processes Browser' (page 198)**.

2. TimeArea and DateArea displayed correctly in status bar

The StatusBar adjusts the TimeArea and the DateArea to the size of the string to be displayed. This is independent of the type of font.

3. Obsolete methods

The following methods are never called:

- MicFwDomainProcess>>defaultValidationBlock
- MicFwDomainObject>>validatesWriteAccessTo: aspect value: value

4. New lineups for OS/2

Loading the configmap **micApplication Interaction - Platform GUI Enhancements Runtime** under OS/2 does not cause an OS Point-Load error.

#MicAbtLabelViewHelp is now commented out for the lineups under OS/2.

5. Leaving a RadioButtonSetEmbeddedPart with ESC does not cause a walkback

Using the ESC key to leave a RadioButtonSetEmbeddedPart does not cause a walkback.

```
#MicFwColumnBeginEditEvent>>triggerWithCallData: aWidget clientData: nothing
callData: aCallbackDataRec
```

6. Leaving a ComboBoxEmbeddedPart with ESC without a selection does not write the old value to the model

Leaving a ComboBoxEmbeddedPart with the ESC key without having made a selection does not write the old value to the model.

```
#MicFwColumnEndEditEvent >> triggerWithCallData: aWidget clientData: nothing
callData: data
```

7. MicFwObjectCollectionControl>>isListUpdateChanging optimized

MicFwObjectCollectionControl>>isListUpdateChanging has been optimized: Filling a model list with indexed collections will not cause the list to be reformatted with asOrderedCollection.

8. Empty popup menus supported

Displaying the contents of an empty MetaPartCollection in a popup menu causes a problem in VA: The empty menu is not visible and is never closed. Thus no other popup menus can be displayed. Using MicA-btCwMenuView as the popup menu avoids this problem.

9. MicFwDomainProcess>>startTransaction V3.5 modifications removed

Modifications to MicFwDomainProcess>>startTransaction in V3.5 caused compatibility problems. The modifications have been removed. <startTransaction> now allows using the context and level of the parent when starting a process transaction

10. New method: MicFrDomainProcess>>forceStartTransaction

<forceStartTransaction> starts or activates the process transaction. If the process uses the parent context, then a separate level is started for the process.

11. Performance optimization for committing a non-isolated TransactionContext

Committing a non-isolated TransactionContext will not send <aspectChange>. This is not necessary, since changes are already globally visible due to ReadTraces.

- <commitVersion: aMicFwObjectVersion inContext: aTrContext>
- <commitVersion: aMicFwObjectVersion inContext: aTrContext inRelationship: aRelationship>

12. Application framework exception handling

Application Framework can handle ExError exceptions. The new hooks <micException:...> specify the type of asynchronous handling of an exception. The default implementation specifies that the process that caused the exception is closed. The default implementation of <doPerformAsynchBlock:...> resignals the exception or calls the hook <micEx-ception:...> when the systemIsRuntime-Flag is true.

<doPerformAsynchBlock:...> specifies the action to be performed immediately after the exception occurs (while the stack is not yet flushed).

If the Exception was generated during a CommandRequest, then the hook <defaultExceptionReturnValue-ForCommandType: aCommandClass> is called. In <defaultExceptionReturnValueForCommandType: aCommandClass> can the return value of the command be defined. For example, changing to a notebook page in response to an exception can be enabled or disabled.

New methods in micAPI-override optional

- MicFwDomainProcess >> micException: anException occurredDuringReadOf: anAbstractValue
- MicFwDomainProcess >> micException: anException
- MicFwDomainProcess >> doPerformAsynchBlock: aBlock inException: anException
- MicFwDomainProcess >> defaultExceptionReturnValueForCommandType: aCommandClass.

New methods in micAPI-services

- MicFwAbstractDomainProcessContext class >> systemIsRuntime: aBoolean
- MicFwAbstractDomainProcessContext class >> systemIsRuntime

Methods moved to micInternal - Expert override

- MicFwDomainProcess >> domainProcessContextClass
- MicFwDomainProcess class >> defaultDomainProcessContextClass

13. resetMappings and load/remove code for ToolCenter modified

#MicFwViewPorts>>resetMappings and the load/remove code for the ToolCenter has been modified.

If a registered Broker is defined under #customViewPort:

- The Broker remains if the ToolCenter is loaded
- The Broker will be registered under #viewPort if the Tools are unloaded.

Loading of application #MicFwViewPorts no longer causes the mapping #viewPort->MicFwViewPortRequestBrokerDispatcher to be deleted.

Version 3.5

1. Dynamic child processes: New methods for declaring, instantiating, controlling

The Framework offers a new way of declaring, instantiating and controlling dynamic child processes. The developer can declare a multiple process connection within the domain processes browser. This connection is used as template to define the parameters for the new dynamic child process. For each multiple process connection a MicFwMultipleProcessController is instantiated.

Parameters for the multiple process connection

- **Name of the connection**. This name is the name of the multiple process controller instance which is used to start and managed the new dynamic child processes. The multiple process controller instance can be accessed via the domain process using the method #multipleProcessControllerNamed: aSymbol.
- **Prefix**. This prefix string is used for the connection name of the new dynamic child processes. If no prefix is defined the system creates a default name.
- **Class of the new dynamic child processes**
- **Viewport class used for the child process connection of the new dynamic child process**
- **Base class**. An instance of this class is used to initialize the new child process. This instance can be used to identify and access the new dynamic child process. This class is also used to identify the multiple process controller when using the #startDynamicProcessUsingObject: anObject method of the domain process.
- **MlsContext**. The mls context class to use for the new dynamic child processes.
- **MlsSuperContext**. The super mls context class to use for the new dynamic child processes.
- **Default view**. The default view class to open when opening a view for the new dynamic child processes.
- **Max instances**. The maximum number of instances of the new child processes which should be allowed to be started.

Usage examples

Most convenient API

```
aMicFwDomainProcess startDynamicProcessUsingObject: anObject
```

The system will look for a declared multiple process connection which has a base class which is the class of anObject.

If such a multiple process connection is declared, use anObject to initialize the new process instance and open the declared view.

If a process has already been started for anObject, open a new view or bring the current view to front.

Lower level APIs

```
(aMicFwDomainProcess multipleProcessControllerNamed: #multi)
  openProcessUsingMainObject: anObject
```

Use anObject to initialize the new process instance and open the declared view.

If a process has already been started for anObject, open a new view or bring the current view to front.

```
(aMicFwDomainProcess multipleProcessControllerNamed: #multi)
  prepareNewProcessUsingObject: anObject;
```

Use anObject to initialize an new dynamic process instance

```
  openProcessUsingMainObject: anObject
```

When used first, a new view is opened for the process. If used when the view is already open, bring the view to front.

```
| newInstance newConnectionName controller|
controller := (aMicFwDomainProcess multipleProcessControllerNamed: #multi).
newInstance := controller prepareNewProcess.
```

Create a new dynamic child process.

```
newConnectionName := controller connectionNameForProcess: newInstance.
```
Get the new connection name.

```
controller openProcess: newInstance
```
or
```
controller openProcessNamed: newConnectionName
```
When used first, a new view is opened for the process. If used when the view is already open, bring the view to front.

Other APIs

```
(aMicFwDomainProcess multipleProcessControllerNamed: #multi)
  isProcessStartedForMainObject: anObject
```
Answer if a process instance is registered for anObject.

```
(aMicFwDomainProcess multipleProcessControllerNamed: #multi)
  allProcessesDo: aBlock
```
Iterate through all process instances controlled by the multiple process controller

```
(aMicFwDomainProcess multipleProcessControllerNamed: #multi)
  connectionNameForProcess: aDomainProcess
```
Answer the connection name which was given to the new child process connection.

2. Dynamic column: Position hint, horizontal alignment

The dynamic column now also supports a position hint. This hint is the position of the static column this column should be inserted after.

The dynamic column now also supports a horizontalAlignment. This can be set via one atom out of ##CENTER , ##BEGINNING and ##END .

Example

```
MicFwMetaAspect new
  accessorString: 'defaultProcess aspect';
  title: 'Aspect';
  positionHint: 1;
  horizontalAlignment: ##CENTER;
  columnWidth: 200;
  yourself
```

3. Container details column: 2 default methods for highly dynamic view

The container details column now supports two special default methods which can be used to implement a highly dynamic container details view.

```
#defaultColumnAttributeNameValueOf: anAspect
```
Get the attribute to be displayed depending on the aspect name of the column.

```
#defaultLabelValueOf: anAspect
```
Get the column label to be used depending on the aspect name of the column.

4. Connectors Browser: partName no longer dispayed

In the Connectors Browser in the column ’paneAccessor’: The widget partName is no longer displayed (the ’paneAccessor’ will still be displayed). partName is no longer displayed because the partName does not always correspond to the Framework accessor.

5. Connectors Browser: New menu item ’Clean up Connectors’

The Connectors Browser has a new menu item ’Clean up Connectors’. Selecting this menu item will cause non-existent connectors to be deleted from the Connectors Browser dialog (connectors can become non-existent, for example, after a view is closed, while using the debugger, in the development image, etc.).

6. MicFwModelObject class methods in category ’overwrite-optional’

The following methods are now in category 'overwrite-optional:':

```
MicFwModelObject class >> authorize
MicFwModelObject class >> isAuthorizationActive
MicFwModelObject class >> isGlobalAuthorizationActive
MicFwModelObject class >> isValidationActive
MicFwModelObject class >> validate
MicFwModelObject class >> validateExec
MicFwModelObject class >> authorizeExec
MicFwModelObject class >> isExecAuthorizationActive
MicFwModelObject class >> isExecValidationActive
```

7. Hierarchy list can display a Collection of String’s

A hierarchy list can now display a Collection of String’s. This is very useful, for example, when displaying the last few levels of a tree:

```
Person
- Address
- - #(street,city)
```

8. Visual part Container Details: Columns can be initialized based on type

Columns can now be initialized based on type for the visual part Container Details (#AbtContainerDetails-View). Initializing colums based on type is defined by selecting the context menu for the Container Details part 'Initialize Columns Based On A Type'.

9. #MicFwHelpRequestEvent implemented for all Widgets (that support the event)

#MicFwHelpRequestEvent (generated using F1) is now implemented for all Widgets that support the event.

10. Properties dialog for Domain Processes (Domain Manager) is now available

To display the properties dialog for a Domain Manager part:

- In the **Visual Age Organizer**: Select **Options / Preferences**. The **Visual Age Preferences** dialog appears.
- In tab **General** under **Preferred Settings View**: Select **Properties Table (recommended)**.
- Click **OK**.
- Double-click on a **Domain Manager part** in a view. The **Properties** dialog appears.

11. Disabling a widget with the focus transfers the focus to the next widget

Visual Age will sometimes not transfer the focus to the next widget when a widget is disabled. Frameworks now ensures that the focus is always transferred when a widget with the focus is disabled.

12. MicFwDomainProcess >> childProcesses always returns the defined child process (not nil)

MicFwDomainProcess >> childProcesses always returns the defined Child Process of the Domain Process.

In the previous Frameworks version, MicFwDomainProcess >> childProcesses could return nil if the ModelViewConnector was not initialized (the MVC used lazy initialization). A nil value caused problems, for example, with the hook #MicFwDomainProcess>>transactionStarted 'self childProcesses'.

13. New methods for setting the color of a widget

The following methods for setting the color of a widget have been added:

```
MicFwViewPort >> defaultBackgroundColorValueOf: dispatcherAspect
MicFwViewPort >> defaultForegroundColorValueOf: dispatcherAspect
UserViewPort  >> <aspect>ForegroundColor
UserViewPort  >> <aspect>ForegroundColor
```

14. New methods for limiting the maximum number of searches for a tree selection

The following new methods can be used to limit the maximum number of searches when a selection is made in a tree:

```
MicFwViewPort >>  defaultMaxSearchStepsValueOf: dispatcherAspect
UserViewPort    >>  <aspect>MaxSearchSteps
```

In the previous Framework version, making a selection in a recursive net could result in a very large or infinite search.

Release 3.4 Version 1.1

1. Application Framework General

- New Debugging Tool.
- New Subaspect called Manipulator. The Manipulator is an object intended for communicating informa-

tion from within an application's model to a view. Information contained in the Manipulator will be used by the Adapter of the view.

- Support of Windowing policy classes in order to meet special needs (e.g. creation of complex views in background, removal of all Cached Views when the application is closed)
- Support of Cached Views as standard application of Windowing policies.
- Drag & Drop support is now available for Visual Age for Smalltalk V4.x.
- Dynamic controls are supported.
- No write access to models or Connections while a traced read is performed.
- Informing processes of select / deselect events on Notebook Pages.
- Some internal fixes for the hierarchy list, dynamic forms and the mnemonic extension.
- Domain Processes Browser: Internal fixes, Connection accessor methods now use symbols, obsolete accessor methods are deleted automatically.

2. MLS

- The file which contains to the path to the MLS directory has changed its name. It's now called mls.mls instead of micado.mls.
- A new storage system based on the Persistence Framework was created, which allows the storage of MLS Data in a relational data base. A POM for SQL anywhere is included.
- Old MLS Data can't be used anymore and must be converted from csv files in the new format once again. The old format can't be read by the new system.
- It's now possible to load only the required keys of a Locale Entity into memory, instead of loading a Locale with its entire data, which was the only possibility in the previous version.
- The Domain Processes Browser now offers to declare a context and its supercontext for a Model Connection .
- There's a new group of contexts called Master Contexts which are held as global by the Locale Manager.
- The new group of Locales allows the user to use delegated keys, which are actually resolved in a Master Context.
- Each csv file now contains the codepage in which it was created. The Excel template files now contain the new format. There is no restriction about the number of possible Locales in one csv file anymore.
- New Locale Categories named and in order to display strings which are assembled from static string parts, non-localized object parts and localized object parts.

3. Migration Notes

In release R 3.3 V2.0 the internal structure of the Application Frameworks was revised and optimized. Despite the objective not to change the API, Application Framework based applications are to be migrated at some places, so that they are executable with the new version. With API - conformal applications this expenditure is supported by a Tool quite easily. This Tool is launched through the **micFrameworks** Menu in the Transcript selecting **micFrameworks>Domain Processes Tools>Migrate Accessor Methods**.

The migrations points in detail:

- Basically the Framework operates internally only with symbol / atoms (no strings any longer). Therefore Connections are referenced now by symbols in instead of strings (etc...).
- The closing procedure of a Domain Process has changed and extended. Please refer to the manual.
- The former API concerning closing a View that is controlled by a Transaction Context
- has to be replaced by: commitAndCloseProcess, abortAndCloseProcess, closeProcess respectively.
- The private method >> was removed. If this access is necessary, the view can be asked by means of the method call MicFwDomainProcess >> modelViewConnector abstractView viewImplementation view. However do not consider this a Framework conformal solution. Try to avoid a navigation from a Process to its view.
- All controls that may be used dynamically (Menu, Notebooks, etc.) now need an instance of instead of a Collection. This MetapartCollection offers a Collection interface. You can create and use a MetapartCollection either similar to a Collection with :, etc or use the new Viewport APIs : and leave the administration of the MetapartCollection to the Application Framework which is recommended.
- Internal List Connections cannot be identified through their Connection Name any more. If you used such Connections in your application you must do one of the following migration steps:
    - Ask the Connection instead of to distinguish them in methods like

:, etc.

- If you access the Connection via :, you can preset the internal List Connection with the new Domain Process APIs:
  - MicFwDomainProcess >> addPresetChoiceConnectionNamed: aSymbol withContext: aContext withSuperContext: aSuperContext
  - MicFwDomainProcess >> addPresetHierarchyListConnectionNamed: aSymbol withContext: aContext withSuperContext: aSuperContext
  - MicFwDomainProcess >> addPresetListConnectionNamed: aSymbol withContext: aContext withSuperContext: aSuperContextntext
- Generally no write access should take place during a traced read via the Framework, since this usually leads to further and uncontrolled updates. The new Framework version therefore generates an Exception when such an operation occur. If no better solution for the problem can be found, then the Exception can be suppressed by executing MicFwDomainObject >> permitWriteDuringReadWhile: [code which cases the exception] instead of the direct read access.
- Primitive Objects like Strings and Symbols need no wrapping into generic domain objects any longer when they are used in lists etc. Therefore the method is obsolete.

4. Tools for the Migration

With the **Transcript** Menu item

`micFrameworks>Domain Processes Tools>Migrate Accessor Methods`

you can launch a Tool that supports you with the migration of Connection Names from strings to symbols. All Framework generated read and write accessors in your Domain Processes are recompiled automatically.

In a further step the Tool pops up a MethodsBrowser with all senders of or and displays the corresponding methods. Modifications of these methods must take place then however manually, i.e. all strings must be replaced by symbols.

Installing the Application Framework

What You Need

The Mynd Application Framework runs with IBM VisualAge for Smalltalk 4.x under Microsoft Windows (32bit), IBM OS/2 and MVS . Before you can use Application Framework, you must have VisualAge installed on your system.

Additionally it requires Mynd Object Behavior Framework. It cooperates optionally with Mynd Persistence Framework. Please refer to the related documentation on installing and using these parts of the Mynd Frameworks family.

Application Framework includes the complete support for developing multi tier applications. According to your needs you may load only dedicated parts of it into your development image.

Application Framework provides you with a set of tools sufficient to design and implement all aspects of your application except the User Interface. It has a complete VisualAge integration and you may use Visual Programming techniques. Thus VisualAge Composition Editor is only required to create User Interfaces for the client tier.

Installing the Release Files

IBM Visual Age Smalltalk (version 4.5 recommended) must be already installed on your computer before installing Frameworks.

Installing Frameworks consists of 2 main steps:

- Importing the configuration maps from the library file on the CD ROM (\manager**V50.dat**) to the library file on your computer (typically \vast\mgr50.dat).
- Loading the applications in the imported configuration maps (along with any other required maps) to your image file (typically \vast\abt.icx).

Importing configuration maps

Frameworks is installed in your Smalltalk environment by importing the required Frameworks applications using the Visual Age Configuration Maps Browser. The applications are imported by importing the configuration maps that contain the Frameworks applications (applications contain the classes and methods).

In the System Transcript: Select ***Tools / Browse Configuration Maps***. The Configuration Maps Browser opens. In the Configuration Maps Browser: Select ***Names / Import*** ("names" refers to the names of configurationmaps).

In the dialog "Enter the full path name of the library" ("library" refers to the .dat file that contain the configuration maps): Double-click on the Frameworks library file **V50.dat** (on the CD ROM).

Selecting the configuration maps for importing and click ***"OK"***.

The configuration maps and their applications are imported to your library.The message "Finished importing from \vast**V50.dat**" appears in the System Transcript.

Loading configuration maps

When a configuration map is loaded, all of its required configuration maps are also loaded. Therefore loading the configuration maps in the order described below will minimize the required effort to load all configurationmaps.

- Load with required maps the .
- Applications that support Object Behavior functionality: .
- Load with required maps the .
- Required for support of visual programming in the Composition Editor: .
- Required for Drag and Drop support: .
- Required for authorization support: .
- Required for object validation: .
- Required for multi-language sup-port: *micApplicationService - Multi Language Development*
- Select ***File / Save Image***.

Importing the code with Load Features

To import the code into your library and to load it into your image:

- Select **Tools -> Load/Unload Features** from your Transcript window.
- Select Mynd Application Framework and its Examples.

1

Concepts

1.1. Introduction

1.1.1. Object oriented software and Frameworks

For the last few years, the increasing computing power on desktop computers has revealed the advantages of Smalltalk's purely object-oriented architecture to a broadening domain of users

Object-oriented programming appeals at multiple levels. For managers, it promises faster and cheaper development and maintenance. For analysts and designers, the modeling process becomes simpler and produces a clear, manageable design. For programmers, the elegance and clarity of the object model and the power of object-oriented tools and libraries makes programming a much more pleasant task, and programmers experience an increase in productivity.

The Application Framework defines an architectural concept that will be an advantage to almost any development project, as a lot of time at the start of projects usually goes into defining an architecture. The Framework’s concept is based on the idea of isolating the actual business logic from technical details and encourages a development style that is based on declaration instead of programming. The declarative style greatly reduces the amount of work required to implement a given functionality.

The acceptance of an application does not only rely on the implemented functionality but also on the Ease-Of-Use and ”Human Interface”. A considerable portion of a development project will be spent designing and implementing a system to work in conjunction with a particular user interface.

Therefore the Mynd Application Framework is designed to run independently of any GUI-dependent logic by abstracting the underlying windowing-system’s functionality to an object-oriented approach where the abstraction will be embedded into the business logic and be an equal part of the solution for the customer’s problem domain.

The powerful Smalltalk environments offered by this framework will enable you to develop your software projects faster and more efficiently than ever before with an increased productivity and quality.

In addition, the long-term experience of Mynd professionals in the areas of banking, insurance, and industry has an invaluable influence on the Application Framework, providing you with a valuable and dependable tool.

1.1.2. Objectives of Application Framework

1.1.2.1. Saving through Use

All complex software requires Frameworks, and in particular Application Frameworks, in order to meet the high demands imposed on object-oriented development. On the one hand, new technologies are constantly increasing the performance of applications, on the other hand, the volume of research and development work is also increasing.

The architecture and components of Mynd Application Frameworks enable the stability of the fundamental structures of your applications to be assured.

1.1.2.2. Saving through Reuse

After the initial phase of familiarisation, use of Mynd Application Frameworks guarantees efficient, systematic and thus inexpensive and rapid software development, from the initial draft right through to the productive application. The effect increases with subsequent projects, due to reuse of the Framework itself and the components which are created in a standard manner. By deciding to use Mynd Application Framework, companies create the basis for meeting a major objective of their development operations: “Fit for re-use”.

1.1.2.3. Saving through Adaptation

Much lower costs are incurred for porting to other platforms and connecting to new technologies. This is achieved by promoting a uniform model-based procedure as well as with technical details which are isolated by means of Adapters. It is now no longer necessary to redefine the entire application in order to ensure that new technologies can be used by the company.

1.1.3. Role in Applications

Many business applications developed today provide a Graphical User Interface (GUI) and normally are connected to some other "external" resources like other applications, communication channels, databases and so on. The event-driven nature of such an application leads to a high degree of complexity, since it must be prepared to handle all possible events which occur, of course, in an unpredictable sequence.

The Application Framework play an important role in the layer model of a complex application between the technically oriented database Frameworks (persistence) and the middleware which depends on the field of application (e.g. Insurance Frameworks). They enable application development to avoid various dependencies and also to reuse generally valid patterns and procedures.

Thus one of the most important ideas behind all the Mynd Frameworks is the isolation of the actual business logic from technical details. The frameworks enforce strict separation of:

- Database
- Database Logic (mapping of objects to tables)
- Model (Domain Objects and Domain Processes, the Business logic)
- GUI Logic (and Interaction logic in general)
- GUI (and other external Interfaces)

On the other hand, the "business objects" should carefully be designed to be independent from all technical aspects of any application in order to be reusable in different applications. A business object's responsibility is purely domain-oriented; a GUI, for example, is only one way see it and to work with it.

The main purpose of the Mynd Application Framework is to offer an architecture which separates business objects from technical ones. In particular, the whole GUI subsystem is strictly separated from the business object classes. As a consequence, the framework provides a way to "connect" business objects to GUIs or other external "views". Moreover, it introduces a powerful view controller which fits most of the needs, but which is very flexible and extensible.

The Application Framework is a class library which allows creating portable applications with a robust and clear architecture. It contains sophisticated updating automatisms and lots of default behavior which make application development deterministic and easy from the start. When the application grows, its complexity stays almost the same, because many design patterns can be reused.

1.1.4. Fields of Operation

Application Framework enable an application to be modeled and implemented in a portable manner and independently of the system and software infrastructure. These include:

- Operating system (APIs, characteristics, release change)
- Front-end system (GUI of the operating system, HTML pages, Java applets)
- Style guide (dialogs, notebook, window in window, Drag&Drop)
- Externalizing (texts, mnemonics and one-touch keys, presentation formats, user guidance)
- Scaling (Batch, network computer, network PC, standalone laptops, servers, mainframes)
- Distribution (Internet / Intranet configuration, communication infrastructure)
- Integration of other Frameworks (persistence, replication, communication)
- Integration of external systems (process control, authorization system, legacy applications)

The boundary to middleware Frameworks is not precisely defined. However, the major consideration is that a high level of integration is required, expandable architecture is provided and that various services can be replaced or downloaded from scratch via a Broker concept. Such services include user transactions, internationalization and much more besides. For these services too there is a demand for solutions which are available irrespective of the field of application and which can be gradually implemented in the development process.

1.2. Frameworks Advantages and Risks

Three questions have to be answered before deciding whether a new application component is to be developed with the assistance of available Frameworks or to be developed from scratch.

1.2.1. Programming or Declaring

Every type of declaration which can replace user-defined code increases the systematic nature and manageability of the application development process. Frameworks enable numerous components to be made available in a declarable manner:

- Relations between objects
- Database access
- Dialog control
- Event processing
- Authorisation rules
- Further...

Advantages:

- Less effort
- Fewer faults
- Simplified maintenance

Risks:

- Higher learning requirement

1.2.2. Use Technical Interfaces or Uniform Abstract Protocols

If a company is to be able to use a new technology, it is frequently necessary to redevelop the application from scratch. An architecture which provides uniform and abstract interfaces so that the interfaces can be adapted via Adapters enables such migrations to become manageable.

- Databases
- Software distribution, configuration
- Communication infrastructure
- GUI elements, user guidance
- Platform, implementation language
- Further...

Advantages:

- Lower learning requirement
- Greater independence
- Simplified migration

Risks:

- Dependence on Adapter delivery or development

Risk minimising:

- Customer / manufacturer partnership
- Early planning

1.2.3. Develop or Use and Expand Existing Architecture

It is very difficult to develop a new application architecture. If an architecture which is future-oriented and open for further expansion is available, in-house development will not be necessary in many cases:

- Technical design of the object model
- Persistence of objects
- Software distribution
- Event control
- Dialog control
- Interfaces to services such as authorisation, multilanguage capability
- Further...

Advantages:

- Higher degree of reutilization

- Modular tasks

Risks:

- Higher learning requirement
- Dependent on manufacturer

Risk minimising:

- Seminars
- Manufacturer is also a user

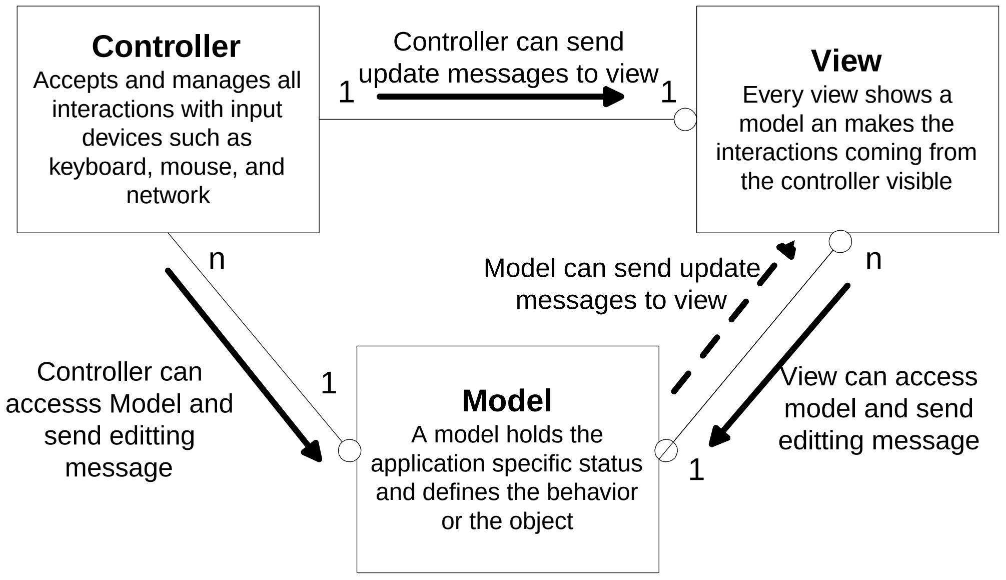
1.3. Mynd Frameworks Concepts

1.3.1. Structure

"Frameworks = (components + patterns) are an object-oriented reuse technique" (Ralph. E. Johnson).

In addition to using isolated components or libraries, Frameworks also enable recurring structures and dynamic correlations to be reused in several applications. Frameworks are available on different levels. Frameworks for segregating Domain Objects from their storage on databases as well as from user interaction have proved to be useful for quite some time.

The use of partially complete timeless architectures which provide patterns for business transactions and objects with their related tasks, from analysis right through to implementation, is becoming increasingly important. Application-specific Frameworks which for instance simulate fundamental concepts of the insurance or finance industry are either already being used or are being developed. Initial solutions are also available in the field of replication at the level of user transactions.


Controller
Accepts and manages all interactions with input devices such as keyboard, mouse, and network
Controller can send update messages to view
1
1
View
Every view shows a model an makes the interactions coming from the controller visible
n
Model can send update messages to view
n
Controller can accesss Model and send editting message
1
Model
A model holds the application specific status and defines the behavior or the object
View can access model and send editting message
1

In the Object Oriented community, the concept of a Model View Controller (MVC) is well-known and broadly accepted. The MVC architecture is designed to provide:

- strict decoupling of the user interface from the model domain Aspects
- control over the flow of messages between the Graphical User Interface and the model domain
- multiple views external to the business model

An application being structured this way shows some obvious advantages: the responsibilities of objects are well-defined and separated from each other, resulting in a higher level of object reusability. Domain and business models can be used in different applications due to their independence from any view-specific aspects. Mynd's Application Framework is targeted at providing a powerful solution to application developers who want to implement systems using the MVC strategy.

1.3.2. Program Structure

Systems developed with the Application Framework follow certain architectural concepts by using and refining some given classes to implement domain-specific processing logic. The following different approaches may be taken:

- **Dialog-oriented**: The application is thought of as sequences of windows and dialogs, controlled by an interactive user;
- **Data-driven**: similar to the Entity-Relationship modeling approach, the static structure of Domain Models (classes representing corporate data) and the relationships among them are defined first;
- **Workflow-oriented**: the focus is set on the modeling of large compound or atomic Domain Processes.

Regardless of which approach is chosen in a project, the Application Framework helps from the beginning in separating different types of objects and functionality - intuitively identifying Domain Models and processes as the "core logic" of the domain, delegation models for interfacing models to presentation layers etc. The level of refinement with respect to data management (persistence), transaction logic and exploitation of sophisticated views or Controls can grow continuously with the application according to the requirements given.

1.3.3. Elements of the Domain

The most important result of object-oriented analysis is a collection of object models describing the static structure of fundamental Domain Object and process-specific classes and their inheritance and association relationships. Moreover, dynamic models are created to show the behavioral aspects of a system and its classes, often illustrated by object state transition diagrams or process models.

Classes with focus on keeping and maintaining corporate data form a global framework and basis for most applications. Examples of this kind of class are Person, Customer, Contract, Bill, etc.; their Aspects are normally stored in databases. Processing-oriented functionality is also modeled with classes, representing process logic to be carried out in order to perform certain tasks.

In describing the Application Framework, the following terms will be used throughout the manual for the elements described above:

- A **Domain Model** is a class of the domain that defines a certain behavior including the knowledge about state, validation, authorization and data
- **Domain Objects** are instances of a Domain Model
- A **Domain Process** is a class of the domain for processing and workflow-oriented tasks and control flow.
- The **Domain Processes** are instances of the Domain Process. Due to their control-oriented nature, Domain Processes have responsibility for managing transaction, if appropriate.

A Domain Process knows about its involved Domain Objects. This relationship is declared by the programmer, and all necessary transaction, clustering and navigating tasks are done by the framework.

Main and subordinate Domain Processes can be distinguished; conceptually, they are organized in hierarchical parent-child relationships (one Domain Process is a child process of another one). This allows process tasks to be split into smaller pieces represented by child Domain Processes, for example, for editing a detail view.

The property of being a child of another Domain Process is purely dynamic: a child Domain Process can act as an independent process or as a subordinate part of a bigger whole, depending on the context.

1.3.4. Domain Object

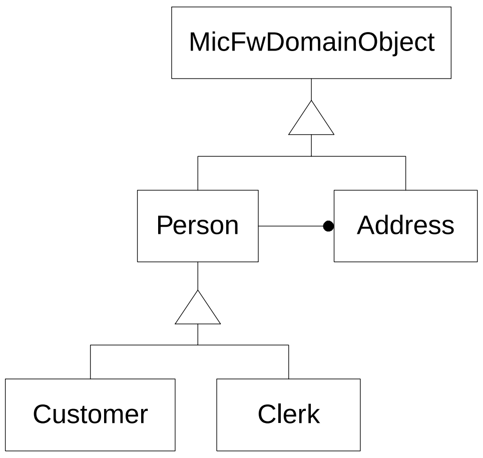
Domain models - with a main purpose of keeping corporate data - are a fundamental part of all applications. All Domain Objects to be connected to views by the Application Framework must be subclassed directly or indirectly from the framework class .

Domain objects implicitly inherit the whole potential of the abstract domain class MicFwPersistentObject which serves as the fundamental domain class for application classes whose instances must be stored persistently by the Persistence Framework (see also [PersFw]). In other words: the concepts of relationship, transactions und persistence can be reused in Domain Objects without restriction.

Example

In the customer service department of a company, customers are referred to clerks for help. Both group members are persons with an identical subset of Aspects like first- and surname, one or more addresses etc.

The classes representing them may be organized according to the following small object model



1.3.5. Attaching Domain Models

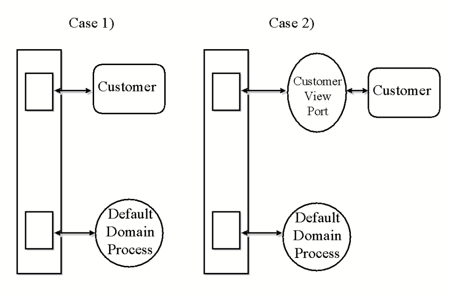
1.3.5.1. Attaching One Domain Model

The simplest - but in most cases not sufficient - configuration is display of only one Domain Model. Two different variations can be distinguished:

- Presentation of Domain Model Aspects without transformations;
- Presentation of Domain Model Aspects with transformations.

The first case doesn't need a special Viewport (a generic one is enough). An example is a person with the attributes firstname and name, where these attributes are presented in a view without transformations.

In the second case a Viewport must be written which provides the necessary transformations. One example may be a presentation of a person's sex (one Model Aspect) in a group of two radio buttons (male / female) on a view.



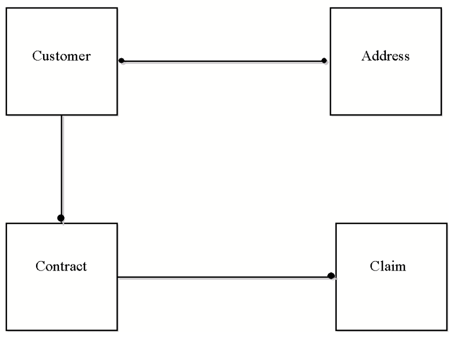
1.3.5.2. Attaching an Object Net

An example from the insurance business will be used throughout the following sections for clarification of concepts. It is a sketch of a typical situation in an application - some views for data acquisition resulting in a creation of one or more Domain Objects.

The application provides the user with a couple of views for entering new claim data or editing an existing claim. The following objects must be attached to the main view:

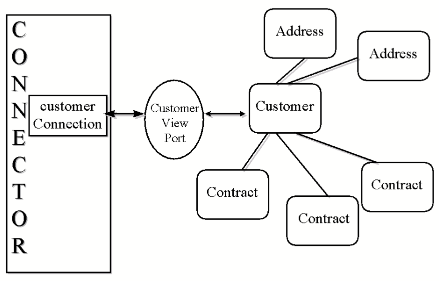
- the edited customer and his main address;
- an overview of all his contracts;
- the data of the newly created claim request.

The relationships between the Domain Models may be as shown in the diagram below:



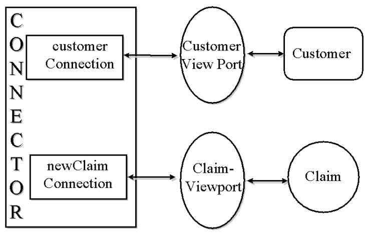
In order to attach objects connected to each other by association or aggregation relationships, it is generally recommended that you implement one or more Viewport classes which provide navigation through the object net. One of the objects is the one primarily attached to the view displaying Aspects of itself and its related objects. The Viewport should be named and designed accordingly.

In the example above, many views would be most interested in the customer's Aspects. The Viewport to be designed for this class must give access to the customer's addresses and contracts. The following picture illustrates the placement and role of the Viewport.



1.3.5.3. Attaching Several Domain Models

Several independent Domain Models may be attached to one Model View Connector by their own - normally named - Connections. Attaching of several Domain Models is necessary, if the Domain Models are executed in one process or under the control of one Main Process.



1.3.5.4. Attaching Collections

The handling of collections is probably one of the most demanding design issues when developing applications. Collections occur under various circumstances, from different types of sources and for different purposes in the flow of an application. Moreover, there is often a need to handle selections in different ways.

The following list shows a few classifications for object collections:

- **Extent collections**: results of arbitrary retrievals, most often retrieved from a database
- **Relationship collections**: full-blown relationships connected in another Domain Object
- **Relationship subsets**: extracted for further processing (delete, remove, edit)
- **Support tables**: all kinds of auxiliary lists providing a choice environment for a single Aspect

The following design rules can be followed when using collections: whether a selection affects the Domain Models or the process (Domain Process).

Example: A view is given showing all contracts of a customer. Starting from this, one contract is selected and edited in another Domain Process (i.e. a child process).

In this context, the selection is irrelevant for the relationship in the model, but serves only as a launcher for new processing. Thus, the selection should be directed to the Domain Process in the first place.

A details view for extended display of the selected element most easily attached with its own Connection, with the selected Domain Model moved in dynamically in the above example using the accessor method of the Base Connection (contract:).

Operating on relationships with the purpose of changing their instance sets (adding a new instance, deleting instances) normally requires transaction processing. Then, all operations can be performed directly on the models, and are discarded if the user cancels operation.

The same holds for moving instances between relationships.

Support tables (code tables, thesauri) need a manager, i.e. some object which is responsible for maintaining the mapping between a single instance and the whole list of available instances. There are several approaches for putting this responsibility somewhere, but most probably there is a real Manager class giving access to the lists. Depending on the implementation, the hook to get into the manager may be a Connection, i.e. the Domain Process is responsible for grabbing and accessing the manager (via a declared

Connection), or a Domain Object provides a path to the manager by means of the corresponding Aspect's value (an object managed by the support table manager).

The List Control has to be named accordingly to select the right path to the available list contents. Since the processing purpose is to display or set a single Aspect in a Domain Object, the selected object is normally sent directly to that Domain Object, possibly converted automatically by an application-dependent type converter for the object's class (see [OBFW]).

1.3.6. Domain Process

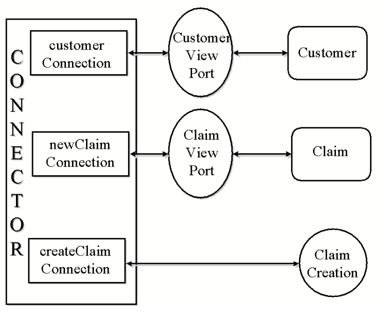
All processing logic and certain aspects of control flow should be placed into Domain Processes. The framework class serves as domain class for its own Domain Processes as well as a simple default class in case where no process logic is explicitly specified (opening a view from a Domain Model for display only). Concerning persistence, the same as described for Domain Models in the previous section also holds for Domain Processes since the class is a descendant of MicFw-PersistentObject. This makes sense if, for example, a Domain Process and its state information must be stored in a database to be saved for later processing (possibly by another user's Smalltalk image).

1.3.7. Attaching Domain Processes

1.3.7.1. Attaching One Domain Process

Especially within the Application Framework, Domain Processes are designed to implement business processing logic or control over existing external modules which provide such functionality (like calculating rates of a certain insurance requirement). Besides this, often some control functionality is involved (e.g. to perform the start or opening of another view controlled by a subordinate Domain Process). The main purpose of a Domain Process is to implement all commands and actions to be performed when the user operates one of the Command Controls like buttons, Menu or toolbar items or doing some special actions like a double-click on a list of objects. The Domain Process decides within the called method when to start or open another view managed by a child process for example.

The following diagram shows a normal configuration with one Domain Process and one or more Domain Models connected to a Model View Connector:



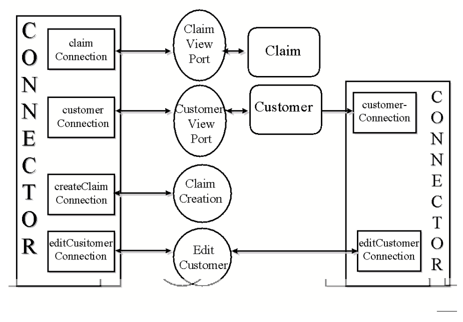
1.3.7.2. Attaching Several Domain Processes

In larger applications, especially when using complex views like Notebooks with some pages offering various functions to the user, it is often advisable to separate the process logic into several Domain Processes. Using this design approach, the resulting models normally are much smaller and probably easier to understand and maintain, and more reusable.

Moreover, there is an additional relationship between Domain Processes, the parent-child relationship supported by the framework in a natural way. This makes it often easier to

- implement dependence between models and correlated behavior, e.g. starting and closing the child process, when the main (parent) process is started or closed respectively,
- to use a Domain Process in conjunction with more than one Model View Connector (thus serving more than one view with some of its functionality).

In the insurance example sketched in the previous sections, it may be possible to implement a customer edit Domain Process which can either operate standing alone, or be started as a child process of the claim request.



1.3.8. Viewport

The previous sections presented a survey of the principal structuring of an application by means of Domain Models and Domain Processes. Within this section, the focus is turned to another aspect of the application to be located somewhere between flow control and presentation. The Application Framework offers certain places to keep all functionality dealing with transformation and presentation of any data stored in an arbitrary object net.

Although this seems to be a very easy concept, it can get rather complicated:

- Data must be accessed and presented from different perspectives and with varying focus on data in an object net (e.g. contract data of a customer, data of a contract covering a claim, main address of a customer, etc.).
- Data must be transformed, condensed and re-combined.Example:
  - the sex of a person as a textual description male / female;
  - the number of covered claims of a policy holder;
  - the kind of event in an insurance case, translated in a readable form by means of a code table;
  - graphical presentation of the tariff data of a property insurance contract calculated on certain conditions.
- Data to be displayed is converted in different ways for various presentation on views.
- The current state of an object or object net may prevent the user from entering certain pieces of data or activating certain operations - the corresponding controls presented on a view must be disabled then.
- Rules of validation can be defined on a field level to allow only correct user input within text entry fields.
- Change operations may or may not be enclosed in transactions which can be committed or aborted.

Summarizing, all kinds of transformations fall into this category, which are designed to retrieve or store data

- navigating through a connected object net

or

- filtering and combining it, or calculating derived values which have no representation in the model or converting data into a (possibly completely different) external representation.

The delegation model of the Application Framework is intended to take over these responsibilities. The class implements the basic default behavior needed by the framework's Model View Connector to handle delegated functionality of connected Domain Objects and Domain Processes as described above in a generic way. Subclasses of the viewport class can be designed to implement the application-specific delegations.

The Framework class does two basic jobs:

- it serves as a generic Viewport class to be used for all connected Domain Models and Domain Processes which don't have their own viewport class assigned,
- it serves as the domain class for application-specific viewport classes - they must be derived from it directly or indirectly.

1.3.8.1. Declaring a Viewport

When opening a Connector, the framework dynamically creates an instance of the proper Viewport class for each connected Domain Object or process - as well as for each element of a List Connection. The "proper" Viewport class is provided by a Broker which detects it by the following default algorithm:

For all Viewport classes (derived from), check the answer of class method portName whether or not it is the class name of the Domain Model or process. If a class can be found, use it; otherwise choose the generic one. See the rules in the Viewport Request Broker capture.

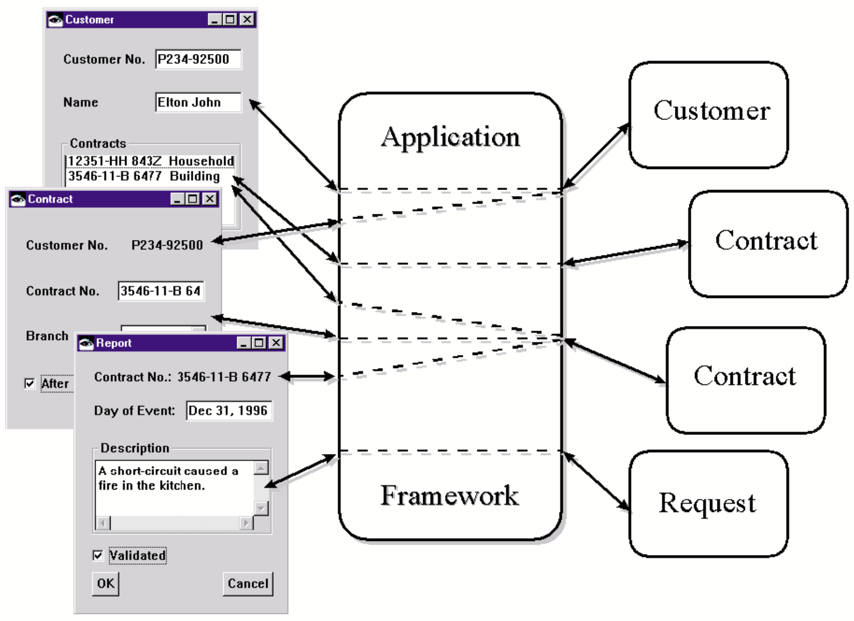
Normally you define in domain processes browser the connection between a domain model and a viewport. But you can register a viewport class (eg, for a list item) in the Framework simply by implementing the portName class method as shown in the following example:

The class should be filtered by a Viewport class 'Create List Viewport' (page 66). The portName method established the relationship between them:

```
MicExampleClubMemberAsListItemViewPort class >> portName
   ^´MicExampleClubMember´.
```

1.3.9. Connecting Objects to Views

One of the main purposes of the Application Framework is interfacing Domain Objects and Domain Processes to the truly event-driven "outside world". This is most commonly represented by Graphical User Interfaces displaying views for end user interaction. Other kinds of interaction may be external or third-party components connected to the application by certain communication protocols. For example, a spreadsheet application may be controlled from a main application with objects connected to it and updated when certain events occur.



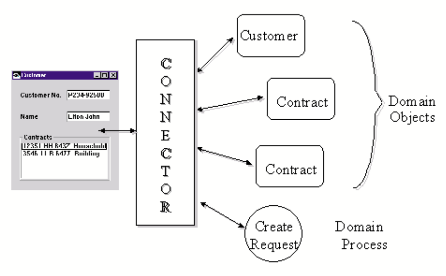
The Application Framework decouples models and views from each other, making the application models independent from views and thus more reusable. On the other hand, decoupling makes it necessary to synchronize both sides of the architecture to provide a consistent state of information both for the user and the internal object space. Connections between the model and views established on Abstract Events and Controls managed, keeping both sides updated:

- if something happens in the view (e.g. the user enters text or presses a Push Button), the connected model gets updated or an operation within the model is started;
- if an Aspect of the Domain Model object changes, all views to which the model is connected are updated.

1.3.10. The Model View Connector

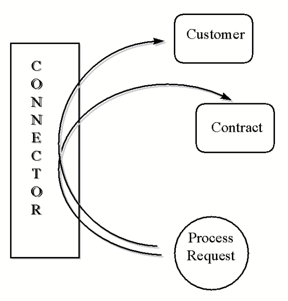
In order to connect objects to an external view, a Model View Connector is established by the Application Framework. Each view is attached to exactly one Connector which, in turn, is connected to the Domain

Objects and Domain Processes on its other side:



Despite the fundamental role the Connector plays within the Application Framework itself, it is almost transparent to the application programmer, since most of the needed application programming interface (API) methods are provided by the abstract Domain Process class itself, or are generated during declaration.

The Connector decouples view and model objects and provides access to connected Domain Objects and Domain Processes on a low level. In this respect, it forms a bridge between the objects, decoupling Domain Objects from Domain Processes and thus making them more combinative and exchangeable:



Using the framework, Domain Objects and processes are connected to a Connector - one Domain Object and one Domain Process in the simplest case. Some trivial situations do not need an explicit Domain Process at all, e.g. for displaying one Domain Object; in such cases no Domain Process needs to be specified and the Domain Object may simply reuse the framework's generic Domain Process.

Generally, more than one Domain Object is connected to a view, and many Domain Processes may be utilized in a complex, editable view. For example

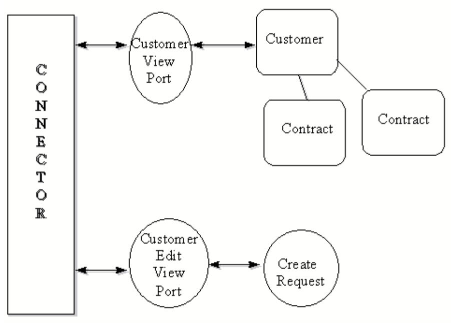
- to display a customer, his main address and a list of all his contracts, or
- to choose some predefined criteria for classification of a newly created contract by selecting elements from Lists whose contents are retrieved from a database.

1.3.11. Viewports

As shown in the previous section, a view will normally consist of more than one Domain Object being displayed on a screen. More often a whole net of Domain Objects will to be processed in one transaction, with one object playing the role of the primary object in the transaction, e.g. editing a Customer or calculating a Policy. Moreover, a lot of current object state information will be transported continuously to the view since each Control's state (enabling state, visibility and other properties) depends on the model's states in a complex way.

In order to separate view-related state handling from Domain Objects and Domain Processes, another kind of class has been introduced. Classes of this type are called Delegation Models because functionality for transportation and the filtering of data and states concerning an object net from a certain object's or view's direction is delegated from the Domain Models to them, leading to a more lightweight kind of

Domain Object.



Delegation models are placed between the Connector and connected Domain Objects and Domain Processes and transform Aspects of the model world to Aspects of the view world and vice versa, not only looking at model contents but also gathering state information concerning a View Aspect. For example:

View Aspect nameModel Aspects firstName, lastName

may be a mapping between Aspects, and a lot of state information can be provided for the View Aspect name to be updated when one Model Aspect changes.

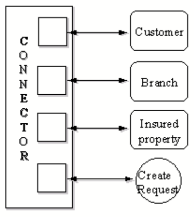
Delegation models are called viewports in the notation of the Application Framework. For each connected Domain Object or Domain Process, one instance of a Viewport is provided by the Connector (see diagram) during the opening of the view:

A simple view with basic accessors and presentation of data do not need the Domain Model to utilize an explicitly defined Viewport the generic one provided by the framework can be relied on in these cases. The default behavior of a Viewport is to propagate simple requests (like accessors or commands) to the connected model. This is often sufficient for Domain Processes, if they do not need special treatment for command states or authorization. This also holds true for Domain Models which implement proper accessor methods for instance variables or even relationships.

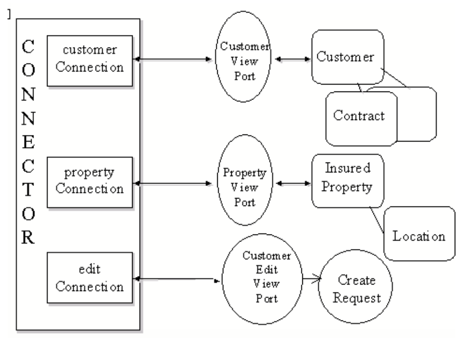
Normally, special Viewport classes are derived from the corresponding framework classes to add application-specific transformations and accessing logic.

1.3.12. Model Connections

In some cases, several independent Domain Objects may be connected to a Connector. For Example:



In order to be able to access each of the connected objects in a logical way (e.g. by the role it plays in the context), each connected object respective Viewport is managed by a named Model Connection within the Connector, which gives access to the objects by a declared name.



Like Connectors, the Connections are almost hidden from the application programmer because of generated Model Accessors methods provided automatically by the Domain Processes Browser which traverse the Connector and corresponding Connection to gain access to the connected Domain Object or Domain Process. Nevertheless, the Connection Names must at least be known by the view designers in order to map each Control to a dedicated Connection by name unless the default Connection is to be used.

1.3.13. Using Framework Objects

The most important aspect when using Framework objects are the accessor methods, which are generated automatically by the Framework.

Via these methods the Framework controls the Object Behavior typing and type conversion, the automatic update , reading and writing into Transaction Contexts, validation and authorization.

Because of this, its extremly important to make sure, that all instance variables used in Framework objects have these generated accessors.

If you navigate through an Framework object net and you want to makr sure, that all changes in these objects cause the corresponding updates, you have to touch Framework-accessor methods while navigating

1.3.13.1. Use of non Framework Objects: Add and Delete in Collections

If you use collections instead of Framework-relationships in order to return the content of a list, you should know, that changes in these objects are not known to the Framework unless you tell it.

Because of this, each

```
add: anObject
```

or

```
remove: anObject
```

method must inform the Framework, that the content changed.

For an Aspect called <#listContent>, which returns the collection, each changing method should implement:

```
self aspectChanged:#listContent
```

1.3.14. Transactions

The most often used configurations of transaction handling can be defined within the property list in sections ‘Transaction Main’ and ‘Transaction Child’. Where you want to configure your transactional behavior depends on the context in which the transaction behavior is needed.

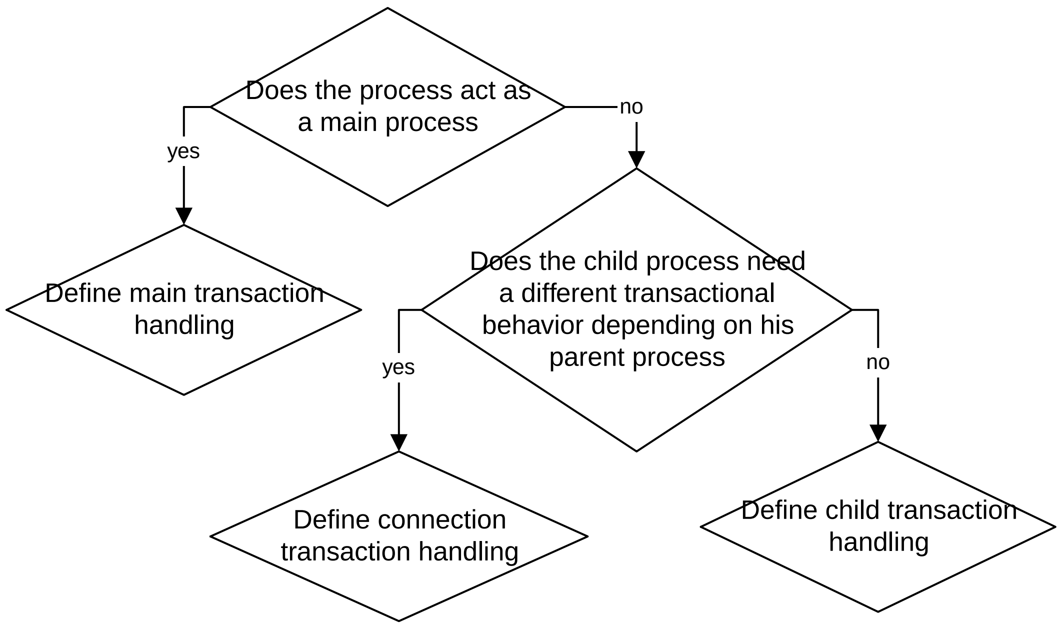
There are two well-defined scenarios where a process needs distinct transactional behavior:

- a process running as Main Process and
- running as a child process.

Besides a Domain Process running as a child may need a different transactional behavior depending on his parent process. This results in three transaction descriptions:

- Main Transaction Behavior
- Child Transaction Behavior
- Connection Transaction Behavior

The following diagram shows which transaction behavior is authorized for special role:


Does the process act as a main process
yes
no
Define main transaction handling
Does the child process need a different transactional behavior depending on his parent process
yes
no
Define connection transaction handling
Define child transaction handling

1.3.14.1. Commonly used patterns

1.3.14.1.1. Main Process Transactional Behavior

For a Main Process there are two often used patterns:

- No transactional behavior for Domain Processes which don't change data or
- Own Persistence Transaction Context when changing data.

1.3.14.1.2. Child Process Transactional Behavior

The most commonly used child process transaction behavior patterns

- No transaction behavior for processes which should write their changes directly into the Transaction Context of their parent.
- **Use parent context** if the parent process changes data and the user should be able to rollback his changes.
- Example: Add a new item to a List.
- Scenario: should make its changes persistent.
- Its child - the - should create an Item. And the add this new Item to the List.
- To avoid to make the new item persistent when it will not be added to the List you have to define a nested transaction for the .
- Own Persistence Transaction Context when the respective process changing data and no process in the parent Process hierarchy does.

Example

Add a new item to a List.

Scenario: should make its changes persistent.

Its child - the - should create an Item. And the first edit the new Item and add this new Item to the list if it is valid and the user does not cancel the dialog.

To avoid to make the new item persistent when it will not be added to the list you have to define a nested transaction for the . The should not get an own transaction level because the abort of this process should abort the creation of the Item as well.

1.3.14.1.2.1. Connection Transaction Behavior

In a third scenario you want to override the child behavior for the case that the respective process runs as a child of a special parent. Then you define a special transaction behavior for the process at its parent process

Hint: If a Connection Transaction Behavior is defined the child transaction behavior won't be used in this connection.

Taking this into account the best way to avoid trouble is **to declare no transactional behavior as children's behavior.** To override the child transactional behavior the definition should be made at the corresponding Connection at the parent process.

1.4. Mynd Frameworks Architecture

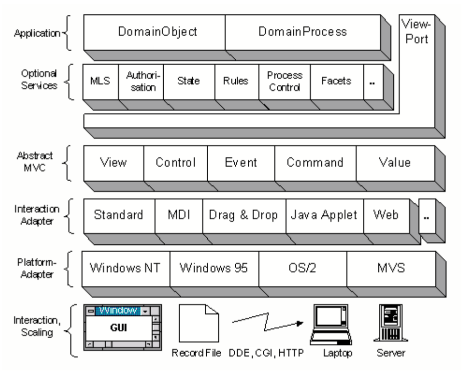
1.4.1. Overview

VisualAge and other object oriented environments usually include a Model View Controller implementation. However, by using it the link to the underlying technology is retained. This refers to details of programming as well as implementation of a GUI style guide.

The fact that for instance

- pressing a button
- selecting a Menu item
- pressing a toolbar icon
- pressing a function key
- mouse double-click
- Drag&Drop operations

are merely different visual representations of the same Aspect of the application, namely the execution of a command is concealed in programming details of the application. To explicitly decouple these details from the domain, the following layered architecture is defined:



1.4.2. Abstractions and Adapters

The Mynd implementation of the MVC abstracts all its core elements and inserts appropriate Adapters for operation on all supported platforms. All listed elements and Adapters are an integral component of Application Framework and are automatically allocated and used by it. The application programmer does not need to have detailed technical knowledge. The system is broken down into an abstract and an adapted part for the following:

1.4.2.1. Abstract View

An Abstract View simplifies the structuring of external accesses for the Framework, as such access can only be routed through real windows. It is irrelevant whether actual access is routed via an Adapter for standard windows, cascaded windows, notebook pages, modal dialogs, HTML pages or even protocol-based data interchange of a windowless application.

1.4.2.2. Abstract Control

An Abstract Control simplifies external access to the Framework, as such access can only take place via real Control. There are only a small number of genuinely different Abstract Controls (see command example in the MVC chapter). The actual access attempts are handled with real Control via an appropriate Adapter.

1.4.2.3. Abstract Event

The entire event control between model and view is carried out with Abstract Events for both directions irrespective of the technology and the special characteristics of the platforms event handling. The genuinely possible events are linked with the corresponding Abstract Events via the Adapter for every Control contained in the view. Processing of events from the model is completely abstract, as any changes are forwarded exclusively via the Abstract Controls, and only these result in an update on the view via their Adapter.

1.4.2.4. Abstract Command

Abstraction for every type of command means that various generally valid functionalities can be implemented in a uniform and reusable manner. These include macros, forward queue, recover, transfer to other processes, undo / redo (rollback / rollforward).

1.4.3. Refined Model View Controller

Mynd Application Framework refined the traditional MVC mechanism to include standard declarable structures. The application developer uses the tools supplied in order to make the declarations.

1.4.3.1. Domain Object

Domain objects describe that part of the MVC model which corresponds to business terms. The behavior and structure are primarily defined in this respect. The appropriate tool (object net browser) from the Object Behavior Framework is used for this purpose, and mapping to the database is described with STOPF from Persistence Framework .

The Domain Object also has open interfaces for connecting authorization and validation. The application developer merely needs to define access to these subsystems; this does not have to be taken into consideration when the application developer programs the business solutions. Application Framework handles all the steps necessary for processing the defined authorizations and validities.

1.4.3.2. Domain Process

The Domain Process is a further aspect of the model part of the MVC and implements business logic that combines the Domain Objects behavior with a well-defined context. Additionally it is used for declaring the user transaction behavior in the application; it can be stored in hierarchies and can also be stored persistently in its current status. Business transactions can thus be represented. This representation process incorporates the business flow logic with or without a connected user dialog and in a manner which can be replicated and resumed. Treatment of authorization and validation is the same as the treatment applicable for Domain Objects.

1.4.3.3. Viewport

The Viewport handles the Control function of the MVC. The developer controls the interaction behavior of all connected models, whether they are Domain Objects or Domain Processes. Inversion of control is a major mechanism in this respect. As the dynamic behavior of the model in terms of user guidance is predefined in Application Framework, the developer does not have to make any event-oriented Connections. Visual components are connected to the model only by completing skeletally defined code parts.

The automatic update mechanism in Viewport also results in a major simplification for the developer. All programmed accesses to the model are recorded and evaluated. In this way, it is not necessary to separately administer the forwarding of changes in the model to the view.

1.4.4. Tools

The Domain Processes Browser is the main tool of Application Framework. It can be used for declaring Domain Processes, their hierarchical relationships and the corresponding user transaction behavior. This is also where the accessed Domain Objects and their Viewports are assigned. A default for visualizing the Domain Process can also be declared here.

The View Connectors Browser offers a visual representation of all active Domain Processes and the connected views together with their contents and permits efficient trouble shooting.

Platform-specific integrations in the development environment can also be used for the declaration. The visual programming Adapter for instance supports visual programming in the composition editor of VisualAge. The Connection between a visual component and the according representation of it in the framework can be defined here.

All tools in other Mynd Frameworks are integrated where applicable. Some have a direct interaction with the operation of Application Frameworks: The Transactions Browser for monitoring the user transactions during the ongoing application, the Object Net Browser or Object Net Explorer for declaring the object network and the Framework Logger for evaluating the past actions triggered by the Framework.

1.4.5. Interfaces

In addition to the seamless link to Mynd Persistence Framework and the open interfaces to subsystems such as authorization and validation, separate Adapters can also be created to interface external products. Brokers are inserted at all central management points within Application Framework for special cases or company-specific extensions; these can be used for replacing entire subsystems of the Framework by customized solutions.

1.4.6. Services

Services are designed in a plug-in manner. They only have to be present if they are needed. You may add the appropriate services step by step during the progress of your project. Refer to the appropriate chapters in this manual for a detailed description of available services like Authorization, Validation or Multi Language Support.

1.4.7. Interaction Adapter

Further interaction Adapters in addition to the standard user interface of OS/2 and Windows NT / 95 are implemented, and can easily be used by the developer without any detailed knowledge. These include the following:

- Drag & Drop protocol
- Creating Menus, Notebook Pages, Forms, etc. on the fly

2

Tutorial

2.1. Introduction

In this chapter we will present a sample application demonstrating some of the most important features of the Application Framework. We will develop this application step-by-step, beginning with a simple Domain Object and a view to represent it. In the course of the tutorial, the Domain Object will get more complex. We will also add Domain Processes, transaction handling, authorization and validation logic, a simple help system etc. Each of these steps will be accompanied by diagrams showing the interaction between the pieces involved. Where appropriate, there will be demonstrations of the most common mistakes and their effects.

After having worked through this tutorial, you should be able to use the basic patterns involved in creating application using the Application Framework.

The model we chose is a fitness club. The application will manage a list of members of the club and allow for editing members and their addresses.

We included a similar example within the frameworks examples. The code for this example can be found in the ENVY-Application There you will find some of the implementations below, but there are extensions for MLS and a different process and view structure implemented.

The code for ObjectModelExplorer can be found in the ENVY-Configuration-Map m
m

2.2. Getting Started

2.2.1. Overview

$$\begin{tikzcd}\boxed{\text{Define a Domain Model}} \arrow[r] & \boxed{\text{Define a Domain Process}} \arrow[r] & \boxed{\text{Define a View}}\end{tikzcd}$$

2.2.2. Preparations for Envy

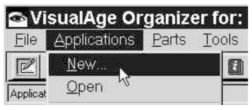
Before you can begin with the examples, you have to do the following steps in the team programming environment ENVY:

- You have to load the development environment. So we load the ENVY- Configuration Map named micObjectBehaviorFramework Development first and then the Configuration Map named micApplication Framework Development. The code for ObjectModelExplorer can be found in the ENVY- Configuration-Map micObjectBeaviorFramework-Tools Object Browser Development and micObjectBeaviorFramework-Tools Object Browser RMI Runtime, Please load them too.
- You need an Application where you will put in all the stuff you will develop for topic fitness club
- This Application requires the ApplicationFrameworks, so you have to register a prerequisite for your new Application. It suffices for the pending examples to indicate the Application "MicFwViewPorts" as "Prerequisite”.

2.2.2.1. Creating a new Application

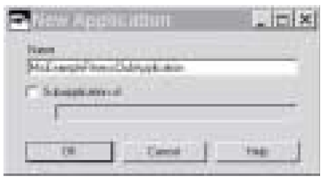
It suffices for most actions inVisualAge to use the VisualAge Organizer. It is one of the two windows which are always actual opened.

As soon as you have brought the VisualAge Organizer into the foreground, you choose the menu "Applications" and there the menu choice "New".



You are now requested to enter the name of your new Application. "MicExampleFitnessClubApplication" was entered in the illustration to the right. However, it is to be recommended to choose another perfix instead of "Mic". This recommendation also applies to class names. You can indeed also load an Application of your colleague only if the application name is not equal to a loaded application.

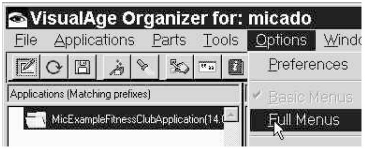
Now you can finish the dialog by clicking ”ok”.



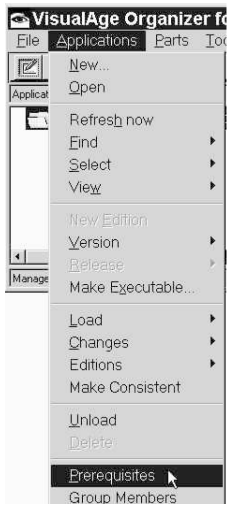
2.2.2.2. Entering the prerequisites

The Prerequisite is "MicFwViewPorts".

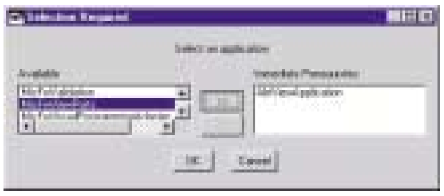
Before you are able to set this prerequisit you have to choose full menus from the options menu.


VisualAge Organizer for: micado
File
Applications
Parts
Tools
Options
Windo
Preferences
Applications (Matching prefixes)
MicExampleFitnessClubApplication(14.
Basic Menus
Full Menus

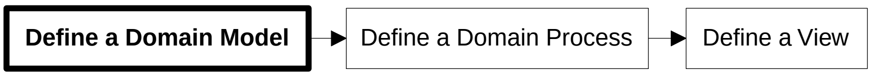
For entering the Prerquistes, you remain in the VisualAge Organizer, select your Application and "Applications" choose the menu choice "Prerequisites".



A dialog box opens itself. Click on "Change..."
Then choose the entry "MicFwViewPorts" in the new dialog box in the left list and displace by clicking the command button ">>" into the right list and confirm you through may click the "ok" command button.



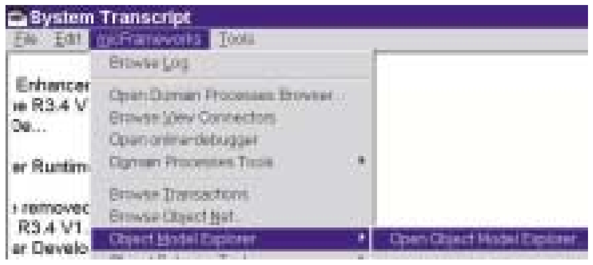
2.2.3. Defining a Domain Model



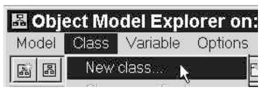
To get started, we'll define a simple Domain Model. At the center of our fitness club application there is the club member, so it makes sense to model it first. In the object model in figure 6.1 the member-class, for our first shot, will have four direct attributes:

- name
- first name
- date of birth
- initial weight when joining the club

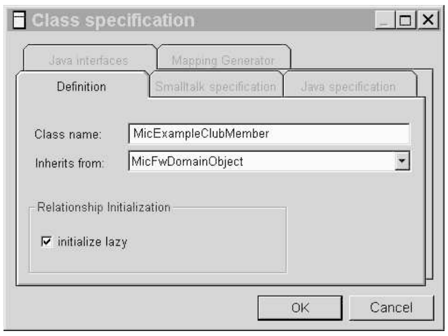
Let's start the Object Model Explorer to create the member class as a subclass of
and to define the four attributes as well as additional information about the class. The Object Model Explorer has to be opened by the SystemTranscript Menu (**micFrameworks -> Object Model Explorer -> open Object Model Explorer**).



To create the member class using menu **Class** → **New Class...**



Here you define the name and the super class of your member class. To define the super class please choose   from the ”inherrits from” list.

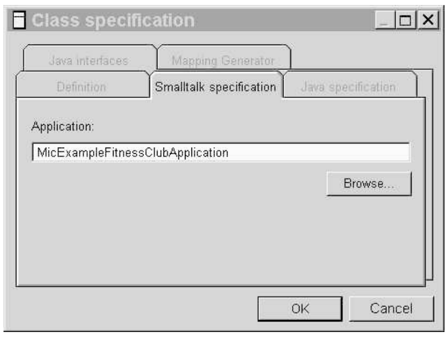


Class name: MicExampleClubMember

Inherits from: MicFwDomainObject

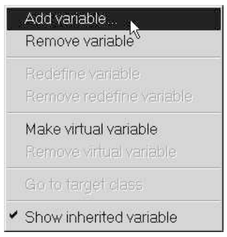
☑ initialize lazy

Now you have to specify the Application. It is the fitness club application you have made a short time before.

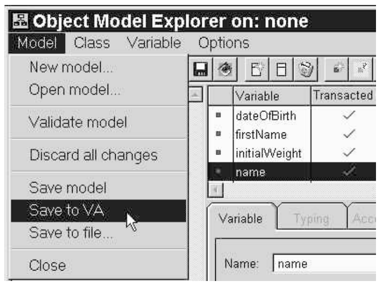


Application: MicExampleFitnessClubApplication

To add attributes click the right mouse button when pointing at the right pane. A context menu appears.



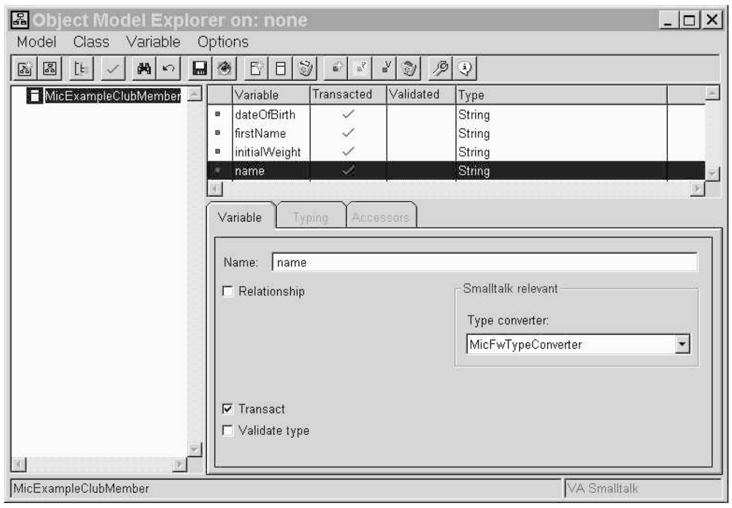
we can add the attributes (instance variables). As each instance variable should be transacted, select the instance variables one by one or all in one and check the box named **Transact**. To save the changes you made (**Model** → **Save to VA**) .



Variable	Transacted
dateOfBirth	✓
firstName	✓
initialWeight	✓
name	✓

Name: name

Your Object Model Explorer should now look like shown in the picture below.



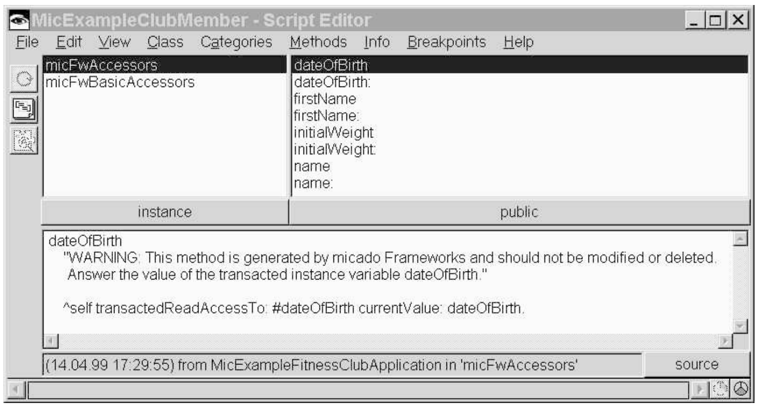
Variable	Transacted	Validated	Type
dateOfBirth	✓		String
firstName	✓		String
initialWeight	✓		String
name	✓		String

Name: name
☐ Relationship
Smalltalk relevant
Type converter: MicFwTypeConverter
☑ Transact
☐ Validate type

MicExampleClubMember | VA Smalltalk

After you have done this, you’ll see that accessor methods for the instance variables have been created. You can examine them in the Script Editor.

Use the VisualAge Organizer to open it. Here you find the new member class in your fitness club application. Double click on it.



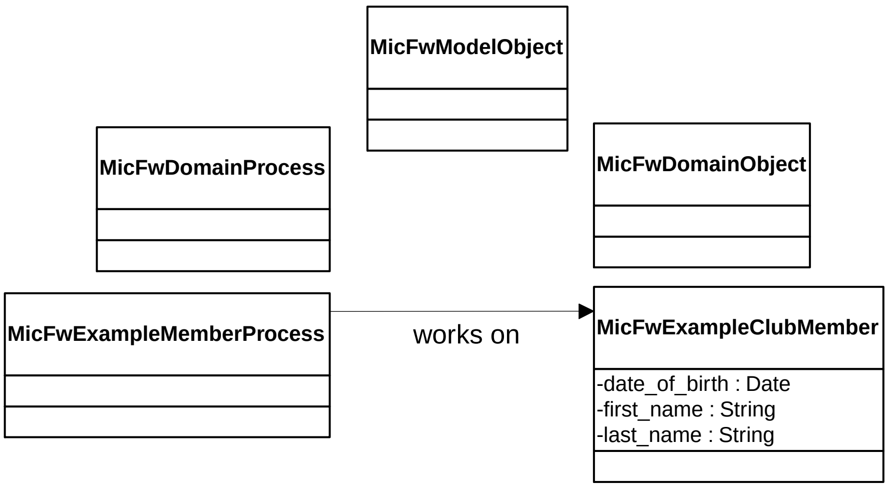
```
dateOfBirth
   "WARNING: This method is generated by micado Frameworks and should not be modified or deleted.
    Answer the value of the transacted instance variable dateOfBirth."

   ^self transactedReadAccessTo: #dateOfBirth currentValue: dateOfBirth.
```

(14.04.99 17:29:55) from MicExampleFitnessClubApplication in 'micFwAccessors'

2.2.4. Defining a Domain Process

Define a Domain Model
Define a Domain Process
Define a View

As you read before the DomainProcess is the view controller and is the place where you can implement domain driven functionality which is only needed in the context of the UseCase in which your view is needed. So it it recommended to implement an own one.



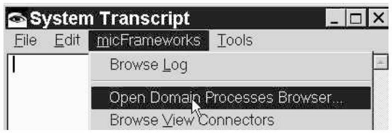
2.2.4.1. Creating the domain process class

You employ the VisualAge Organizer to do this. Create the class for EditMember. To do this you choose the menu "part" in the VisualAge Organizer, there the submenu "New", there the submenu "part...". A dialog box appears. Now type the name of your EditMember class in the first field (e.g. "MicExampleEditMember"). Then you type MicFwDomainProcess in the "inherits from" field (super class) and click ok.

2.2.4.2. Processing the DomainProcess in the DomainProcessBrowser

The DomainProcessBrowser is used to process the DomainProcess in its quality as a UseCase and view controller class.

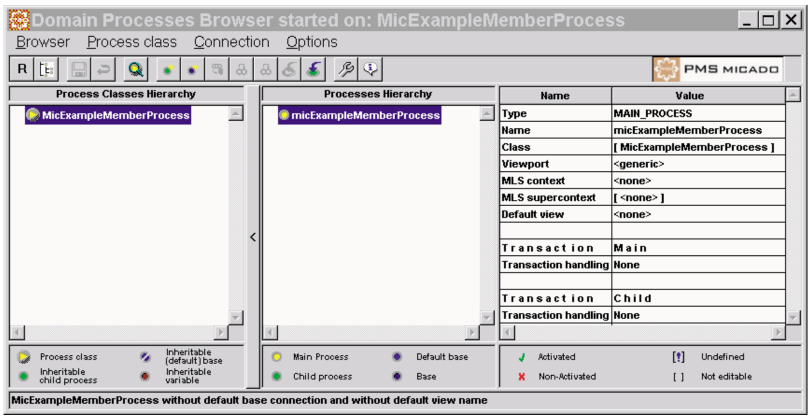
You achieve the DomainProcessBrowser by using the "micFrameworks" menu if you are there then choose the entry "Browse domain Processes...".



A dialog box which resembles that one which appears where you type in the name of your edit member class (e.g. "MicExampleMemberProcess").

A list appears that shows to you only classes which are derived from MicFwDomainProcess. Choose your edit member class from this list and you confirm with "ok".

The actual DomainProcessBrowser appears.



Name	Value
Type	MAIN_PROCESS
Name	micExampleMemberProcess
Class	[MicExampleMemberProcess]
Viewport	<generic>
MLS context	<none>
MLS supercontext	[<none>]
Default view	<none>
Transaction	Main
Transaction handling	None
Transaction	Child
Transaction handling	None

MicExampleMemberProcess without default base connection and without default view name

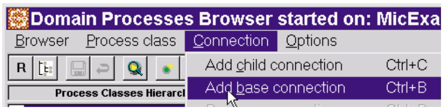
For this example only we are interested in

- the left list, you select the domain processes which you want to define here;
- the list in the center, here once again appears the Process for Process" in its role as a model object for the Relationship to itself; here you will define the relationships of the UseCase (Connections) to the domain objects;
- the right list, is the parameter list; here you can change name and type and a lot more of a selected connection; the transaction handling that you can change here as well is not relevant by now. It is part of a later exercise.
- the status line shows, whether and which default view was declared to the DomainProcess and – not important for us yet – the default base connection (relationship between the process and a domain object, which permits the framework user to identify the model aspect of a data shown visual part only by the aspect accessor see later).

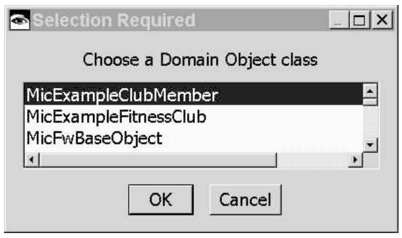
Your task is to define a relationship to the domain object (member class).

2.2.4.2.1. Declaring the connection to the domain object

A connection to the domain object is called base connection. Such a connection can be declared by the selection of menu choice "Add base connection" of the menu "Connection".



A List of domain object classes appears.Here you have to set a type for the connection, meens you have to define which instances are accepted by the system for this connection. Select the entry for your member class from the list.


Selection Required

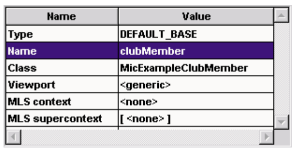
Choose a Domain Object class

MicExampleClubMember
MicExampleFitnessClub
MicFwBaseObject

OK Cancel

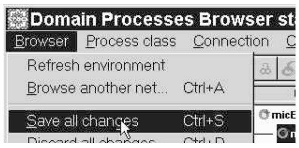
After closing the dialog the center list shows a new connection which is a child node of the connection for our member process.

At the right list you can to change the name of the connection to the domain object club member from ”newDefaultBaseConnection” to "clubMember". The name of a connection is used by the Framework later in order to reach the domain object which is located at this connection from the view. You should ensure to write the name in lower case, because the Framework get into trouble if you use upper case for connection names.



Name	Value
Type	DEFAULT_BASE
Name	clubMember
Class	MicExampleClubMember
Viewport	<generic>
MLS context	<none>
MLS supercontext	[<none>]

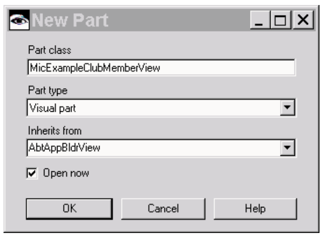
Now you can save your work.


Domain Processes Browser st
Browser
Process class
Connection
Refresh environment
Browse another net... Ctrl+A
Save all changes Ctrl+S
Discard all changes Ctrl+D

2.2.5. Defining a View

Define a Domain Model
Define a Domain Process
Define a View

To define a view for the class we’ll use the VisualAge Composition Editor. Go to the VisualAge Organizer, select your Application and choose (**Parts -> New**...). Now enter the definition for the new part as shown in the figure below:


New Part

Part class: MicExampleClubMemberView

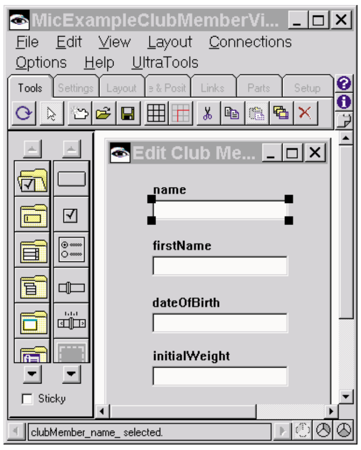
Part type: Visual part

Inherits from: AbtAppBldrView

☑ Open now

OK Cancel Help

Now, draw the Window in the Composition Editor.


MicExampleClubMemberVi...
File Edit View Layout Connections
Options Help UltraTools
Tools Settings Layout Posit Links Parts Setup
Edit Club Me...
name
firstName
dateOfBirth
initialWeight
Sticky
clubMember_name_ selected.

For each attribute of the Domain Model, use one entry field. The name of an entry field is used to store two settings which are needed to know where the entry field gets its content from. To separate this two settings you have to use an underscore.

The first setting is the identifier of the object which provides the content serving attribute. The identifier is the base connection name ”member” that we defined in the domain processes browser.

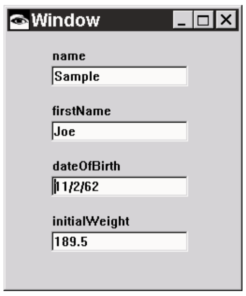
The second name part is the attribute that provides the content.

So the entry field has to be named exactly like this two settings separated by an underscore and followed by an underscore. For instance, the entry field for name must be called 'member_name_'.

When you’re done, save the Window and execute the following code:

```
| theMember |
theMember := MicExampleClubMember new.
theMember
   name: 'Sample';
   firstName: 'Joe';
   dateOfBirth: '11/2/62';
   initialWeight: '189.5'.
MicExampleMemberProcess new
   clubMember: theMember;
   openView: 'MicExampleClubMemberView'
```

You will see the Window you just defined displaying a sample member of our fitness club:



name: Sample

firstName: Joe

dateOfBirth: 11/2/62

initialWeight: 189.5

It’s also very interesting to open two Windows on the same model, for instance by executing the following code:

```
| theMember |
theMember := MicExampleClubMember new.
theMember
   name: 'Sample';
   firstName: 'Joe';
   dateOfBirth: '11/2/62';
   initialWeight: '189.5'.
MicExampleMemberProcess new
   clubMember: theMember;
   openView: 'MicExampleClubMemberView'.
```

```
MicExampleMemberProcess new
   clubMember: theMember;
   openView: 'MicExampleClubMemberView'
```

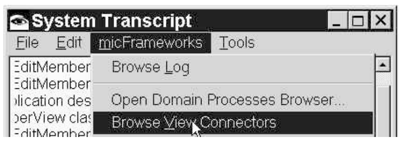
The two Windows will open one on top of the other, so you'll have to drag the top one a little to be able to see the other one. You will see that changes you make to the model in one Window will propagate to the other one as soon as you tab out of the corresponding entry field.

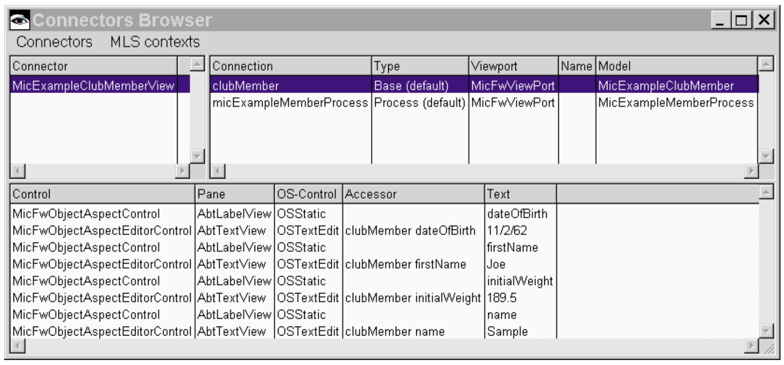
2.2.5.1. A Look Behind the Scenes

At first, the behavior of the   may seem strange to a developer who hasn't used the Application Framework before. There are no methods to populate the entry fields or write their contents back to the model, nor are there any events that the Controls specifically registered to. What's going on ? How does the data get to the Controls and back ?

Fortunately there is a tool in the Application Framework to visualize the Connections between the view and the model. It is called the Connectors Browser.

To reach it use the SystemTranscript Menu (**micFrameworks -> Browse View Connectors**).





Connector
MicExampleClubMemberView

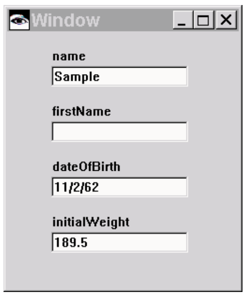
Connection	Type	Viewport	Name	Model
clubMember	Base (default)	MicFwViewPort		MicExampleClubMember
micExampleMemberProcess	Process (default)	MicFwViewPort		MicExampleMemberProcess

Control	Pane	OS-Control	Accessor	Text
MicFwObjectAspectControl	AbtLabelView	OSStatic		dateOfBirth
MicFwObjectAspectEditorControl	AbtTextView	OSTextEdit	clubMember dateOfBirth	11/2/62
MicFwObjectAspectControl	AbtLabelView	OSStatic		firstName
MicFwObjectAspectEditorControl	AbtTextView	OSTextEdit	clubMember firstName	Joe
MicFwObjectAspectControl	AbtLabelView	OSStatic		initialWeight
MicFwObjectAspectEditorControl	AbtTextView	OSTextEdit	clubMember initialWeight	189.5
MicFwObjectAspectControl	AbtLabelView	OSStatic		name
MicFwObjectAspectEditorControl	AbtTextView	OSTextEdit	clubMember name	Sample

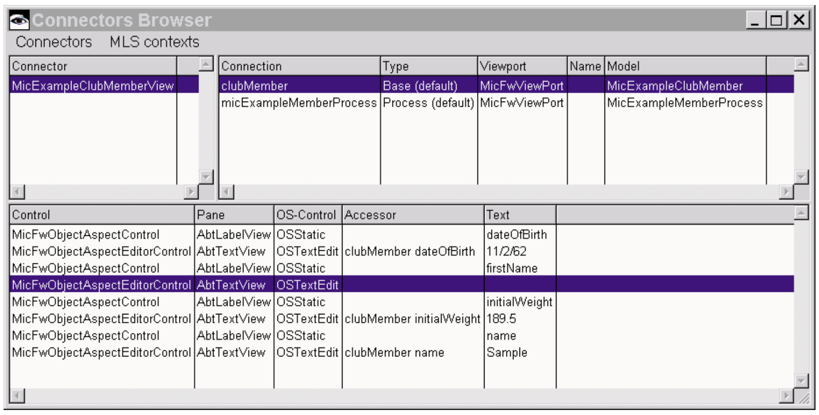
If you open it while the little example is running, it should look like the figure above. You can see that the open   has a Base and a Process Connection. The model of the Process Connection is the Domain Process (   ) as we have. The 'Base' type Connection's model is an instance of   . As there is only one Base Connection, it is automatically the Default Base Connection.

The List at the bottom of the Connectors Browser shows you the Controls in the selected Window. Notice that the name we gave an entry field is listed as an 'accessor' here (with the trailing underscore removed). This accessor will be sent to the default Domain Model as a Smalltalk message to retrieve the value of an attribute, the accessor followed by ':' will be used to set the value of the attribute in the model.

This scheme gives you an indication why correct naming is so important. Imagine you forget the underscore in a Control's name: the Application Framework will not be able to identify the attribute of the Domain Model that the Control is supposed to represent. For example, if you forget the underscore on the 'member_firstName' entry field and open the Window, it will look like this:



name	Sample
firstName	
dateOfBirth	11/2/62
initialWeight	189.5



Connector
MicExampleClubMemberView

Connection	Type	Viewport	Name	Model
clubMember	Base (default)	MicFwViewPort		MicExampleClubMember
micExampleMemberProcess	Process (default)	MicFwViewPort		MicExampleMemberProcess

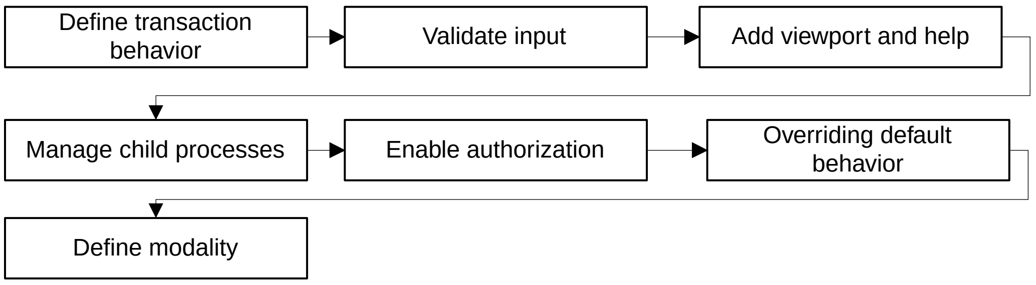
Control	Pane	OS-Control	Accessor	Text
MicFwObjectAspectControl	AbtLabelView	OSStatic		dateOfBirth
MicFwObjectAspectEditorControl	AbtTextView	OSTextEdit	clubMember dateOfBirth	11/2/62
MicFwObjectAspectControl	AbtLabelView	OSStatic		firstName
MicFwObjectAspectEditorControl	AbtTextView	OSTextEdit		
MicFwObjectAspectControl	AbtLabelView	OSStatic		initialWeight
MicFwObjectAspectEditorControl	AbtTextView	OSTextEdit	clubMember initialWeight	189.5
MicFwObjectAspectControl	AbtLabelView	OSStatic		name
MicFwObjectAspectEditorControl	AbtTextView	OSTextEdit	clubMember name	Sample

A quick look at the Connectors Browser will show you what's happening. The one Control is an AbtText-View; however, it doesn't have an accessor associated with it. This immediately tells you that something must be wrong with the name, as the Application Framework interpret everything that is followed by an underscore as an accessor.

The following picture gives you an overview of the application parts you made (gray) and the parts generically added by the Application Framework (white). While you implemented the Domain Process ( ) and the view ( ) the Framework generates a Viewport and a Connection to connect them.

2.3. Beyond Basics

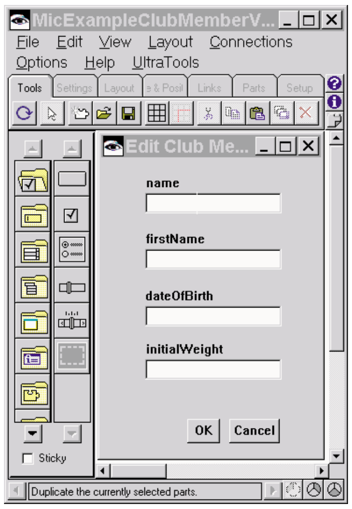
2.3.1. Overview



2.3.2. Transactional Behavior

Of course we want to do something useful with our Domain Process. Let’s add two buttons to our view to allow the user to either commit the changes he made to a member or to cancel them. These buttons will be called 'OK' (commit button) and 'Cancel' (abort button). The resulting Window looks like the figure below.

Make sure you’ve named the 'OK' button 'MicExampleMemberProcess_ok_' and the 'Cancel' button 'MicExampleMemberProcess_cancel_'. Now, pressing one of the buttons will result in either 'commit' or 'abort' being sent to the Domain Process as a Smalltalk message.



The corresponding methods in are:

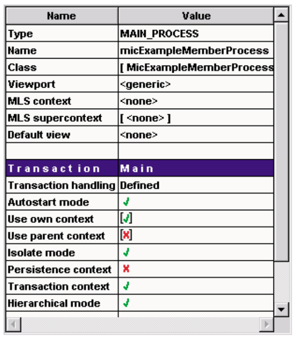
```
commit
"commit the changes made and close the associated window"
self commitAndCloseProcess
```

and

```
abort
"abort the changes made and close the associated window"
self abortAndCloseProcess
```

Now we have to define the transaction behavior of our member process. Therefore we open the domain processes browser on our process an take a closer look on the parameter list. Here are three possibilities to define transaction behavior. For now our member process is the entrance of our application. The transaction behavior of the first started process has to be defined at ”M a i n”.

Click in the field under ”Main” and choose the radio button ”defined”. Here you have to change the behavior in the following way (see next figure):



Name	Value
Type	MAIN_PROCESS
Name	micExampleMemberProcess
Class	[MicExampleMemberProcess
Viewport	<generic>
MLS context	<none>
MLS supercontext	[<none>]
Default view	<none>
Transaction	Main
Transaction handling	Defined
Autostart mode	✓
Use own context	[✓]
Use parent context	[✗]
Isolate mode	✓
Persistence context	✗
Transaction context	✓
Hierarchical mode	✓

Usually it make sense to set isolation to because isolation means that the process reads commited values only and this avoids that the user gets confused by an other user who works on the same object.

In this example we don't use persistence. So we need no persistence transaction context.

If you choose Hierarchical Mode or not isn't important for this example. So choose what ever you want.

Now close the browser and test what we've done so far. Executing

```
| theMember |
theMember := MicExampleClubMember new.
theMember
   name: 'Sample';
   firstName: 'Joe';
   dateOfBirth: (Date today);
   initialWeight: 189.5;
theMember inspect.
MicExampleMemberProcess new
   clubMember: theMember;
   openView: 'MicExampleClubMemberView'
```

Will open an inspector as well as the Window we have defined on a member. Go to the inspector and check the instance variable 'name'. It will show 'Sample' just like it does in the editing Window. Now, go to the editing Window and change the name to 'Smith' and tab out of the field. Go back to the inspector and click on 'name' again. Nothing has changed so far. Click on 'Cancel' in the editing Window and recheck the name in the inspector: again, nothing has changed.

Now, close the inspector and execute the same code again. This time, after changing the name to 'Smith', press 'OK'. Check the name in the inspector and, notice that it has changed to 'Smith'. The transaction handling works.

2.3.3. Validating Input

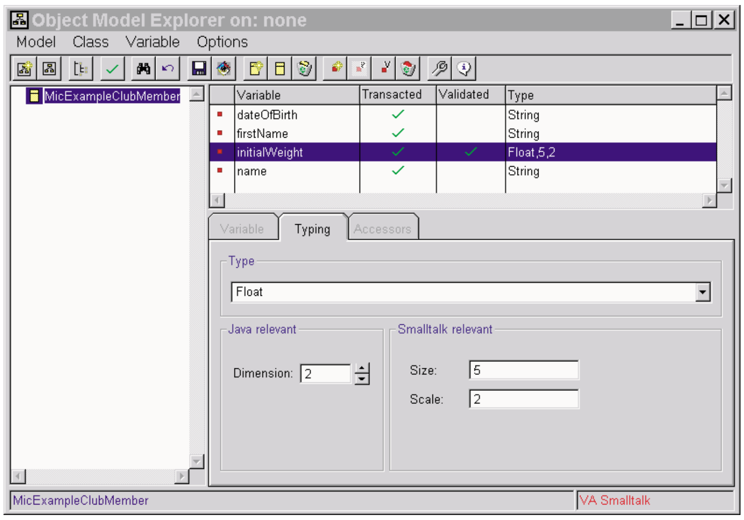
An important aspect of common applications is validation.Often, the user has to enter data of a certain type (like for instance date) in a field. Data that doesn't conform to that type must be rejected. Sometimes the data entered must also be in a specific range.

The Framework provide a mechanism to define validation on the Domain Object level. The advantage of defining it there is that it is independent of the views and consistent across all views and other parts of the system that deal with the attributes in question.

Let's extend our example with some validation logic. The 'initialWeight' attribute of a club member represents a measurement of his weight when he joined the club. Because this is numerical data, we'll assign it the type 'Float'. Doing this is straightforward: open the Object Model Explorer on the class . Now, select the instance variable 'initialWeight'. Choose **Set Type** from the context Menu and select **Float** in the following dialog.

Since we want the type to be validated whenever we assign something to 'initialWeight', also check the box labeled **Validate Type** in the lower part of the Object Model Explorer.

The browser should look like this now:


Object Model Explorer on: none
Model Class Variable Options
MicExampleClubMember

Variable	Transacted	Validated	Type
dateOfBirth	✓		String
firstName	✓		String
initialWeight	✓	✓	Float,5,2
name	✓		String

Variable Typing Accessors

Type: Float

Java relevant — Dimension: 2

Smalltalk relevant — Size: 5, Scale: 2

MicExampleClubMember
VA Smalltalk

Now, save your changes (**Model** → **Save to VA**). You're ready to start the application again. Executing

```
| theMember |
theMember := MicExampleClubMember new.
theMember
   name: 'Sample';
   firstName: 'Joe';
   dateOfBirth: (Date today);
   initialWeight: 189.5.
theMember inspect.
MicExampleMemberProcess new
   clubMember: theMember;
   openView: ‘MicExampleClubMemberView’
```

will open the now familiar Window again. Go to the 'Initial weight' entry field and enter something that is not a Float, like 'xxx'. You will see the entry ”**error**” in the entry field. If you press the OK-button, in the inspect window you will see that the former value will not be changed.

To prevent errors, it's practical to check whether or not the user's entry for the weight is inside a plausible range. This is where another Framework mechanism is very useful. Validation can be declared for each attribute of a Domain class on the basis of the value of one argument block. The argument passed to that block is the new value. The block has to return true when the value is acceptable and false otherwise.

Before you implement the method you have to load the configuration map ”micApplication Service – Validation Runtime”.

Example

Suppose we want to limit the values for 'initialWeight' to a range from 70 to 500 pounds. The method for declaring this is a class method in :

```
initializeValidationTable
   super initializeValidationTable.
   self
      validateWrite: #initialWeight
         using: [:value | (value notNil and: [value > 70])
            and: [value < 500]]
```

First, the block checks to determine that the value is not nil (nil wouldn't understand the '>' and '<' comparison operators, so that an exception would be raised) and then checks to determine whether the value is inside the range.

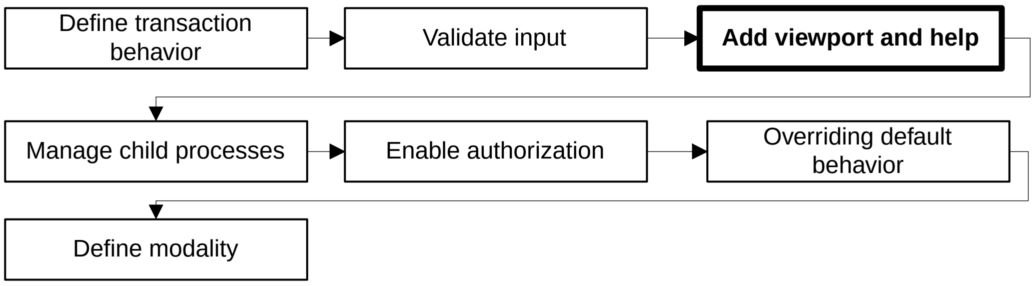
Now, it's time to test what we have done so far. Execute

```
| theMember |
MicExampleClubMember initializeValidationTable.
theMember := MicExampleClubMember new.
theMember
   name: 'Sample';
   firstName: 'Joe';
   dateOfBirth: (Date today);
```

```
  initialWeight: 189.5.
MicExampleMemberProcess new
  clubMember: theMember;
  openView: ‘MicExampleClubMemberView’
```

Please note that validation is initialized in line 2. Now, try to enter a value outside of the range 70 to 500 or even one of those values themselves. As soon as you try to tab out of the field, the value changes back to what it was before.

2.3.4. Adding a Help System



2.3.4.1. Possible Help systems

At this point, we may want to introduce help features for the Window we created so far. On a field level, the Framework supports two different types of help: regular help (the one you get when you press F1) and Hover Help (the tiny Windows that pop up when you leave the cursor on a Control for a couple of seconds without doing anything. Under Windows-Operating System this feature is called ToolTips).

2.3.4.2. The ViewPort – The place for the help system

The Help information have something to do with displaying an attribute of a domain model at the view. If we implement this information at the view we can not reuse it at other views where we display the same model.

If we implement the help system at the domain model completely then we would dicide how to show the help at the view.

The best way is to implement the texts at the model and the way of displaying at a class which is esspecially made for such code – the Viewport.

At the viewport you can decide how to display help information of a model.

2.3.4.3. Name Convention for Viewports

You can have several Viewports and Viewport classes for the same domain object and the domain object class. The reason is that you want to display a domain object at different views in a different way. So it is recommended to name a Viewport class in a way that you can identify where the ViewPort is used. At our member process we have one base connection which is typed to club member. So it is enough that the name of the Viewport contains the related domain object class (here our club member) and the domain process which work on the domain object (here).

So a name which contain any needed information would be

2.3.4.4. Create Viewport

The definition of our Viewport is:

```
MicFwViewPort
  subclass: #MicExampleClubMemberAtMemberProcessViewPort instanceVariableNa-
mes: ''
    classVariableNames: ''
    poolDictionaries: ''
```

2.3.4.5. Connect Viewport to Model at Process

The Viewport must be able to identify its model.

To define a Viewport for the club member at the place where the funtionality of the Viewport should be used. This is done by defining it at the base connection at the domain processes browser on your member process.

So open the domain processes browser on your member process. Expand the connection hierarchy in the central list, select the base connection which is typed to your club member (e.g. clubMember) and choose the viewport at the settingstable (right list) from the dropdown list. If you made the viewport and it is not visible right now click on the refresh button which is located in the left uper corner markted with an ”R”.

Tip: The name of the Viewport is too long to be displayed completely, if you want to have more space to read, click at the collaps button marked with ”<” located at the right side of the class hierarchy.

Now you can close browser and Save the work.

2.3.4.6. F1 Help Protocol

As mentioned before the protocol for help texts is defined in the Viewport. For regular help, a method named has to be implemented (e.g. for the attribute 'name') which returns the text to be displayed when the user presses F1.

```
nameHelpText
^ self model nameDetailedHelp
```

As you see you have to use the method call to navigate to the related domain model. Here we have to implement the method .

```
MicExampleClubMember>>nameDetailedHelp
   ^'An Employee need the name of the member to identify him.'
   , ((self name isNil or:[self name isEmpty])
     ifTrue:[ ' So it is highly recommended to fill out this entry field.']
     ifFalse:['']).
```

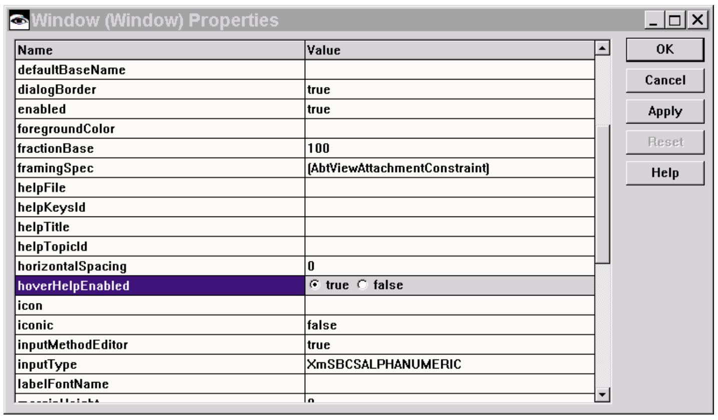
2.3.4.7. Hover Help Protocol

For Hover Help, a method named will be invoked if it is present. Consequently, implementing

```
MicExampleClubMemberAtMemberProcessViewPort>>nameHoverHelpText
   ^self model nameShortHelp
MicExampleClubMember>> nameShortHelp
   ^'The name of the member.', (self name isEmpty
     ifTrue:[ ' Please to fill out this entry field!']
     ifFalse:[''])
```

2.3.4.8. Enable Hover Help at Visual Age

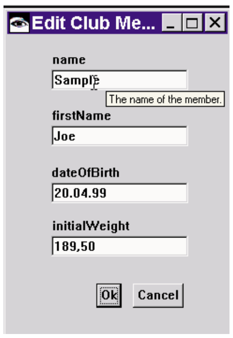
If you’re using the VisualAge Composition Editor, make sure to check the box 'enable Hover Help over all children' in the Main Window attribute dialog (**openSettings -> hoverHelpEnabled -> true**).



Name	Value
defaultBaseName	
dialogBorder	true
enabled	true
foregroundColor	
fractionBase	100
framingSpec	(AbtViewAttachmentConstraint)
helpFile	
helpKeysId	
helpTitle	
helpTopicId	
horizontalSpacing	0
hoverHelpEnabled	[✓ true] false
icon	
iconic	false
inputMethodEditor	true
inputType	XmSBCSALPHANUMERIC
labelFontName	

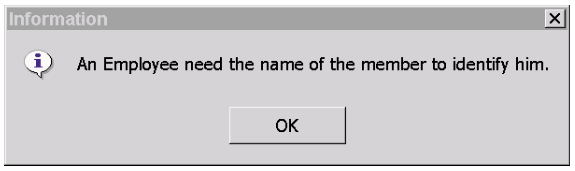
2.3.4.9. Run Application with Help System

Now, run the application again and leave the cursor on the ‘name' entry field. You should see the following picture:



name	Sample
	The name of the member.
firstName	Joe
dateOfBirth	20.04.99
initialWeight	189,50

Pressing F1 on the same field yields


Information

An Employee need the name of the member to identify him.

OK

2.3.5. Managing Child Domain Processes

Any real application will naturally consist of a multitude of Domain Processes which interact with one another. To show how this interaction can take place (and to stick to what is said in the introduction to this chapter), we will now extend the sample application with another Domain Process.

This process will manage the list of members of the fitness club and allow for insertion, deletion and editing of members. What do we need to accomplish this ?

- a Domain Object to represent the club with its list of members
- a view to display the List and the associated controls
- a Domain Process to control these objects

2.3.5.1. Create Club Domain object

Since there will only be a single instance of the club, we'll make use of the Singleton pattern and store it in a class variable named 'Current'. The definition for the club class thus becomes:

```
MicExampleObjectDomain subclass: #MicExampleClub
instanceVariableNames: ''
classVariableNames: 'Current '
poolDictionaries: ''
```

2.3.5.2. Relationship between Club and Member

Use the Object Model Explorer to define the relationship between the club and the member. First you have to add both classes to the model (**Class -> Add class net**).

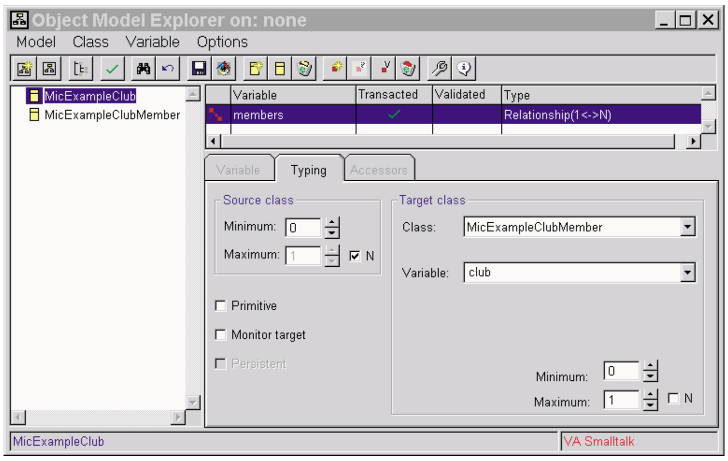
There will be a 1-to-n relationship to MicExampleClubMember (one club has multiple members), so there needs to be one instance variable to hold the list of members, the instance variable '**members'** (**Variable -> AddInstanceVariable**) and then (not needed for this example but for transactions) you set this instance variable to be transacted.

We want to define a both side relationship because we don't want to loose the information that we have a 1-to-n-relationship. So we need an additional attribut at the club member. So we add an instance variable named '***club'*** at the club member (**Variable -> AddInstanceVariable**).

Now we have to define the relationship. Therefore you set the Relationship toggle button to true and change to the typing page.

Here we define the relationship to   .

The Object Model Explorer then should look like:



MicExampleClub
MicExampleClubMember

Variable	Transacted	Validated	Type
members	✓		Relationship(1<->N)

Source class
Minimum: 0
Maximum: 1 [✓ N]
☐ Primitive
☐ Monitor target
☐ Persistent

Target class
Class: MicExampleClubMember
Variable: club
Minimum: 0
Maximum: 1 ☐ N

MicExampleClub | VA Smalltalk

The Singleton pattern for managing the one and only instance of is implemented in the following class methods:

```
current
"answer the sole instance of the receiver"
Current isNil ifTrue: [Current := self new initialize].
^Current

current: aValue
"set the sole instance of the receiver"
Current := aValue
```

We also need some code to initialize a list of members for our testing. It makes sense to write the following class method in the application class of our example ( ):

```
initializeMembers
"MicFwFitnessClubExample initializeMembers"
| p club |
MicExampleClub current: nil.
club := MicExampleClub current.
p := MicExampleClubMember new.
p
  club: club;
  name: 'Sample';
  firstName: 'Joe';
  dateOfBirth: (Date fromDays: 23589);
  initialWeight: 180.4.
p := MicExampleClubMember new.
p
  club: club;
  name: 'Anyone';
  firstName: 'Frank';
  dateOfBirth: (Date fromDays: 23889);
  initialWeight: 192.7.
```

This code must be executed only once to initialize a club with two members.

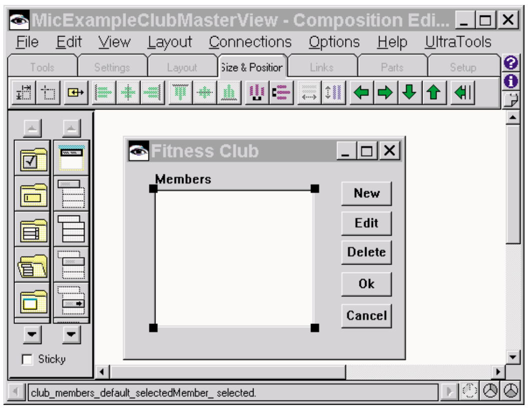
2.3.5.3. Create Club View

Now, we need a view to represent the list of members and to show the Controls necessary for manipulating it. This view can look like the figure below.

Please use the following names for the Controls:

- for the 'New' button:
- this will be implemented in the Domain Process that manages this view
- for the 'Edit' button:
- as you guessed, we will invoke the edit process / window we already have
- for the 'Delete' button:
- this will be implemented in the Domain Process that manages this view
- for the 'Ok' button:
- this is a default message implemented in the Framework. Closes the Window and commits all transactions
- for the 'Cancel' button:
- this is a default message implemented in the Framework. Closes the Window and commits all transactions
- for the List Control:

- used to retrieve the list and get / set the currently selected member



The choice of names will be explained when we get to the Domain Processes involved.

2.3.5.4. Create domain process to handle a club

We will create a Domain Process that manages the view we just defined and that has one child Domain Process for editing the currently selected member.

2.3.5.4.1. Class definition

The name of the club process should express that the process is the entrance process of the application. So we named it master. So the class definition looks like this:

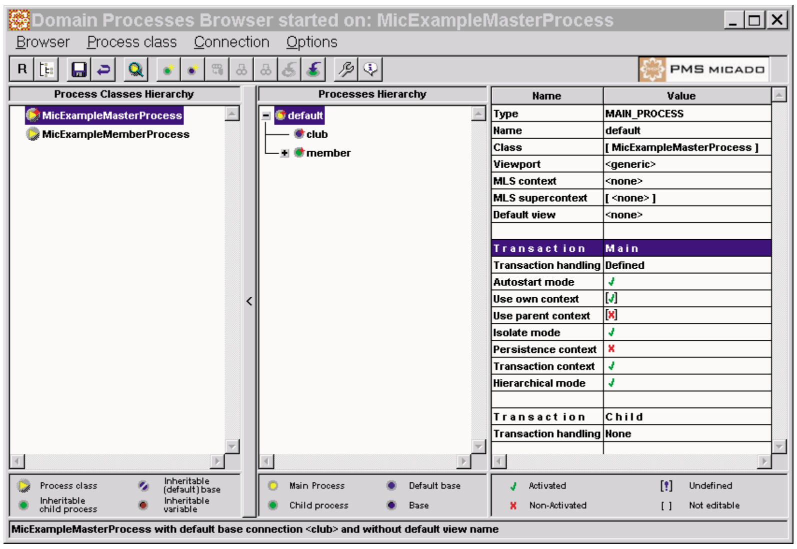
```
MicFwDomainProcess subclass: #MicExampleMasterProcess
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
```

2.3.5.4.2. Connections

This Domain Process will have the following properties:

- a Base Connection named 'club' for an instance of
- a Process Connection named for the member editing Domain Process
- a Process Connection named which references itself

After you saved this definition, go to the Domain Processes Browser and define the connections according the figure below. Change into 'default' by editing the property **Name**. Make sure you set the default view name to and turn on transaction handling.



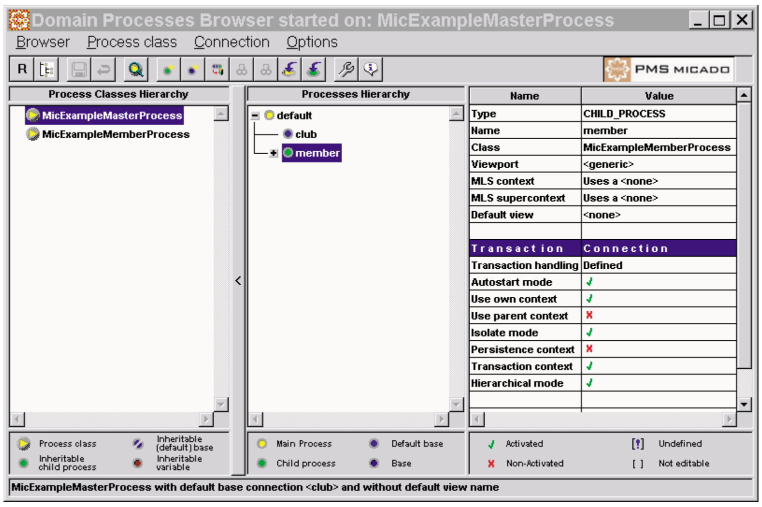
Name	Value
Type	MAIN_PROCESS
Name	default
Class	[MicExampleMasterProcess]
Viewport	<generic>
MLS context	<none>
MLS supercontext	[<none>]
Default view	<none>
Transaction	Main
Transaction handling	Defined
Autostart mode	✓
Use own context	[✓]
Use parent context	[✗]
Isolate mode	✓
Persistence context	✗
Transaction context	✓
Hierarchical mode	✓
Transaction	Child
Transaction handling	None

MicExampleMasterProcess with default base connection <club> and without default view name

2.3.5.4.3. Transaction of child process

Be aware that your main transaction behavior which we defined for the member process as the application starter will not be taken when it is started by an other process. This is the situation here.

So we have to defined a transaction behavior at the child process connection where the member process is defined at. If you closed the Domain Processes Browser please open the Domain Processes Browser on your master proces again, select the child process connection and define the transaction behavior as see below:



Domain Processes Browser started on: MicExampleMasterProcess

Process Classes Hierarchy: MicExampleMasterProcess, MicExampleMemberProcess

Processes Hierarchy: default, club, member

Name	Value
Type	CHILD_PROCESS
Name	member
Class	MicExampleMemberProcess
Viewport	<generic>
MLS context	Uses a <none>
MLS supercontext	Uses a <none>
Default view	<none>
Transaction	Connection
Transaction handling	Defined
Autostart mode	✓
Use own context	✓
Use parent context	✗
Isolate mode	✓
Persistence context	✗
Transaction context	✓
Hierarchical mode	✓

MicExampleMasterProcess with default base connection <club> and without default view name

Finally we save all changes.

2.3.5.4.4. Instance Variable for current selected member

The currently selected club member is a status information of the domain process and should be defined in the same way as we defined instance variables of a usual domain object. So we open the Object Model Explorer by the SystemTranscript Menu (**micFrameworks -> Object Model Explorer -> open Object Model Explorer**).

To be able to work on our club process (e.g.) we have to add it to the model (**Class -> add class net).**

Now we add the instance variable named 'currentMember' to the process. It is not necessary but a good idea to set the transact flag.

Now we can save our work (**Model -> save to VA**) and close the browser.

2.3.5.4.5. Filter the Selection

To make the Connection protocol for storing the selected member compliant with the protocol used by List Controls, we need to implement two methods in a new Viewport for the master process.

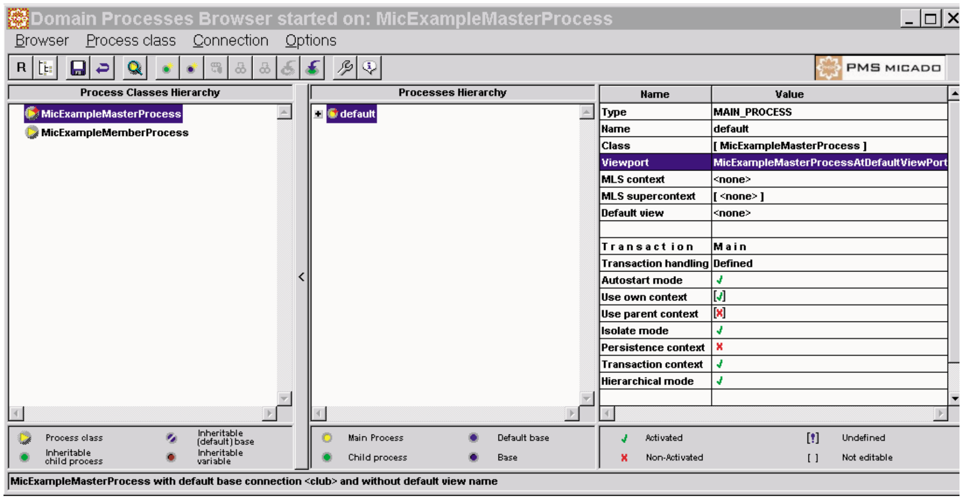
2.3.5.4.5.1. Create Viewport

We need a new Viewport because the methods contains view driven code. The creation method we have to execute is:

```
MicFwViewPort subclass: #MicExampleMasterProcessAtDefaultViewPort
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
```

2.3.5.4.5.2. Bind the Viewport to the Process

The Viewport should be defined at the connection named of the club process. It is the same procedure as we made before for the club member. So Domain Processes Browser should look like the following:



Name	Value
Type	MAIN_PROCESS
Name	default
Class	[MicExampleMasterProcess]
Viewport	MicExampleMasterProcessAtDefaultViewPort
MLS context	<none>
MLS supercontext	[<none>]
Default view	<none>
T r a n s a c t i o n	M a i n
Transaction handling	Defined
Autostart mode	✓
Use own context	[✓]
Use parent context	[✗]
Isolate mode	✓
Persistence context	✗
Transaction context	✓
Hierarchical mode	✓

MicExampleMasterProcess with default base connection <club> and without default view name

Don’t forget to close and save it.

2.3.5.4.5.3. Implement the filter Methods

```
MicExampleMasterProcessAtDefaultViewPort>>selectedMember
   ^Array with: self model currentMember
MicExampleMasterProcessAtDefaultViewPort>>selectedMember: aMemberColl
   (aMemberColl notNil and:[ aMemberColl notEmpty])
      ifTrue:   [ self clubMember: aMemberColl first.
                  self parent member2 clubMember: aMemberColl first.]
```

2.3.5.4.6. ViewPort for list items

Before we can test the view, we have to define a couple of additional methods. First of all, since the List is composed of club members, we have to define how a club member should display itself in a List. This is view driven code. So we have to implement this code in a Viewport.

2.3.5.4.6.1. Create List Viewport

Every Intem in a list is decorated by a Viewport. For our task we need an user defined Viewport for a club member displayed as item in a list.

```
MicFwViewPort
   subclass: #MicExampleClubMemberAsListItemViewPort
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
```

2.3.5.4.6.2. Bind Viewport to Model

A Model find its Viewport by a class method at the relateted Viewport class that returns the class name of its related model. This class method is called   .

So implement the following code:

```
MicExampleClubMemberAsListItemViewPort class >>portName
   ^’MicExampleClubMember’
```

2.3.5.4.6.3. Item presentation

The method for doing this is to implement the message 'asListEntry' in the corresponding class, meaning we will put the following method into   :

```
asListEntry
"answer the list representation of the receiver"
|myModel|
^(myModel := self model) name asString , ', ' , myModel firstName
```

2.3.5.4.7. Open Child Process

We have defined the name of the 'Edit' button in the view as 'default_edit_'. The Button will be sent the message 'edit' when it is pressed. As we want this 'edit' to open the member edit Window on the currently selected member, we have to implement domain and view related code.

2.3.5.4.7.1. Domain Code

The implementation of domain related code in   is:

```
edit
"edit the club member that is currently selected in the parent"
  ^self member
    clubMember: self currentMember;
    yourself
```

2.3.5.4.7.2. View related Code

As you see there is no call of : # . The reason is that this is view related code and has to be implemented at the Viewport of the master process.

The implementation of view related code in is:

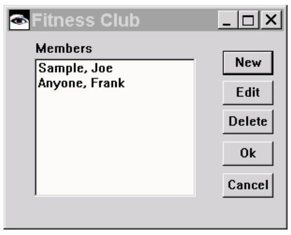
```
edit
  self model edit openView: #MicExampleClubMemberView
```

2.3.5.5. Start Club Application

Now it’s time to test what we have done so far. Execute

```
MicExampleMasterProcess new
  club: MicExampleClub current;
  openView: #MicExampleClubMasterView
```

to see the following Window:



Selecting a member in the List and pressing 'View / Edit' will take you to the edit window we already implemented. Changes made in the edit window will be saved and propagated back to the List window only if you press 'OK'; pressing 'Cancel' will abort the corresponding transaction and discard all changes.

There are still two methods we haven’t implemented so far: deleting and inserting members.

2.3.5.6. Delete a member from the club

The application need no interaction with the user after the user decides to delete a member.

In the we implement:

```
deleteMember
"delete the currently selected member"
self club members remove: self currentMember
```

2.3.5.7. Add a new member to the club

Now we want to create a new member and add it the the list. If we are lazy and don’t care about the validations of a new member we have to implement one method only. The method has to be implemented into

```
newMember
"insert a new member into the list and set the selection"
| theMember |
theMember := MicExampleClubMember new.
theMember club: self club.
self currentMember: theMember
```

To avoid trouble it is a good idea to initialize to attributes of a member with default objects. Therfore a smalltalker user a method called initialize to do this and implement it as an instance method at the class which should be instanciated:

```
MicExampleClubMember>>initialize
  super initialize.
  self
    name: '';
    firstName: '';
    initialWeight: 0;
    dateOfBirth: Date today.
```

2.3.5.8. Test delete and add member

When you test deleting and creating club members, sometimes it is necessary to start from the beginning again. Then execute:

```
MicExampleFitnessClubApplication initializeMembers.
```

Afterwards you can execute:

```
MicExampleMasterProcess new
   club: MicExampleClub current;
   openView: #MicExampleClubMasterView
```

2.3.5.9. Create, edit and add a member at once

The Implementation of adding a member to a club is not a very professional way to do this. In practice you want to edit a member in the same procedure as you create it and if you and if you decide to cancel the transaction every action should be rolled back at one time.

The three actions are:

- Creation of a member
- Adding the member to the club
- Fill the attributes of the new member

So we need a new transaction and this means a new process. The new process will control the transaction and call the edit member process when it is starting. This should happen in one view.

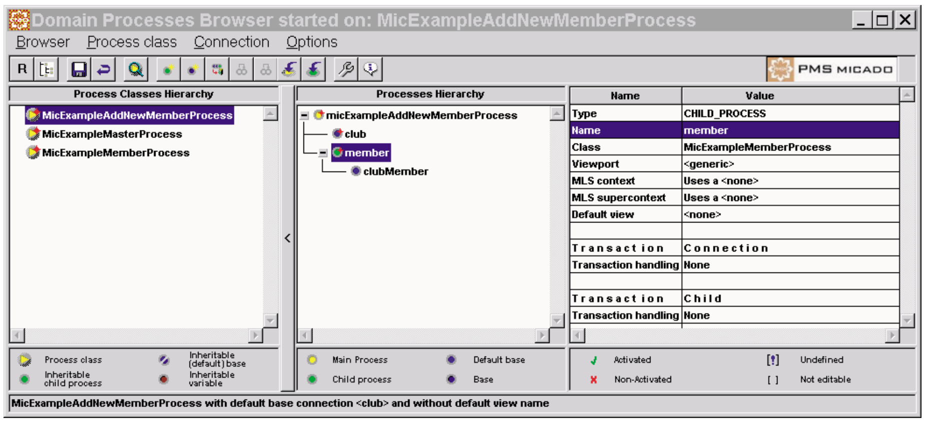
2.3.5.9.1. Create add new member process

The add new member to club process could be created by executing the following code:

```
MicFwDomainProcess subclass: #MicExampleAddNewMemberProcess
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
```

2.3.5.9.2. Definition of the add new member process

The add new member to club process should be defined by the domain processes browser. After you have done all definitions with the exception of the transaction behavior (see next topic) it should look like this:



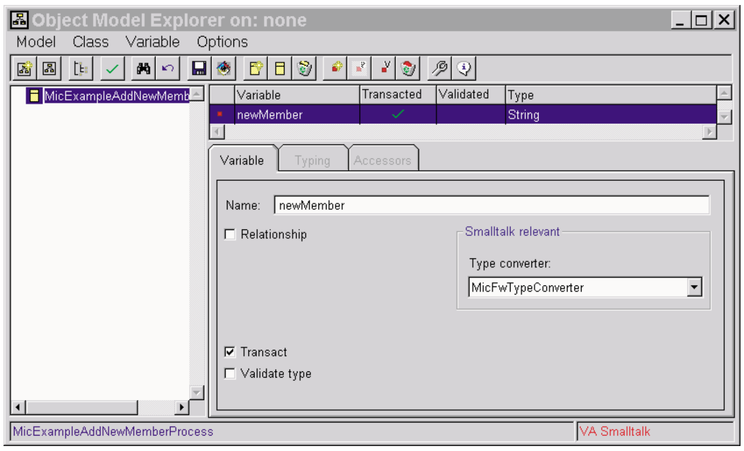
Process Classes Hierarchy:
- MicExampleAddNewMemberProcess
- MicExampleMasterProcess
- MicExampleMemberProcess

Processes Hierarchy:
- micExampleAddNewMemberProcess
  - club
  - member
    - clubMember

Name	Value
Type	CHILD_PROCESS
Name	member
Class	MicExampleMemberProcess
Viewport	<generic>
MLS context	Uses a <none>
MLS supercontext	Uses a <none>
Default view	<none>
Transaction	Connection
Transaction handling	None
Transaction	Child
Transaction handling	None

MicExampleAddNewMemberProcess with default base connection <club> and without default view name

Now we need the instance variable for the new member. So open the Object Model Explorer, add the add new member process to the model and add the instance variable   to it.



	Variable	Transacted	Validated	Type
●	newMember	✓		String

Name: newMember

☐ Relationship

Type converter: MicFwTypeConverter

☑ Transact

☐ Validate type

MicExampleAddNewMemberProcess | VA Smalltalk

Don’t forget to save your work!

2.3.5.9.2.1. Using Transaction hooks

Now we have to implement the creation of a new member and to initialize our child process with the new member at the beginning of the transaction.

So we reimplement the method at :

```
transactionStarted
   self newMember: MicExampleClubMember new.
   self member clubMember: self newMember.
```

This method is called automatically when a process starts.

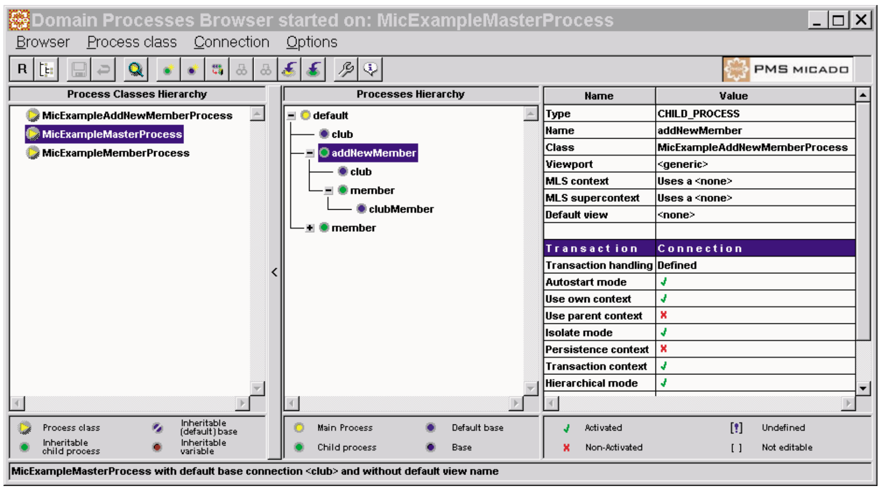
And we have to add the new member to the club after the user commits the transaction:

```
MicExampleAddNewMemberProcess>>aboutToCommitTransaction
   self club members add: self newMember
```

This method is called automatically when a user commits a process successfully.

2.3.5.9.3. Integration of the add new member process

Now we add the add new member process to the master process. Therefore we open the Domain Processes Browser on our master process, add a new child processes typed as add new member process and define a child transaction behavior for it at the connection. The Browser should look like this:



Domain Processes Browser started on: MicExampleMasterProcess

Browser Process class Connection Options

Process Classes Hierarchy
- MicExampleAddNewMemberProcess
- MicExampleMasterProcess
- MicExampleMemberProcess

Processes Hierarchy
```
default
├── club
├── addNewMember
│   ├── club
│   └── member
│       └── clubMember
└── member
```

Name	Value
Type	CHILD_PROCESS
Name	addNewMember
Class	MicExampleAddNewMemberProcess
Viewport	<generic>
MLS context	Uses a <none>
MLS supercontext	Uses a <none>
Default view	<none>
Transaction	**Connection**
Transaction handling	Defined
Autostart mode	✓
Use own context	✓
Use parent context	✗
Isolate mode	✓
Persistence context	✗
Transaction context	✓
Hierarchical mode	✓

MicExampleMasterProcess with default base connection <club> and without default view name

Now we have to change our method at the MasterProcess:

```
newMember
   "insert a new member into the list and set the selection"
   ^self addNewMember
      club: self club;
      yourself
```

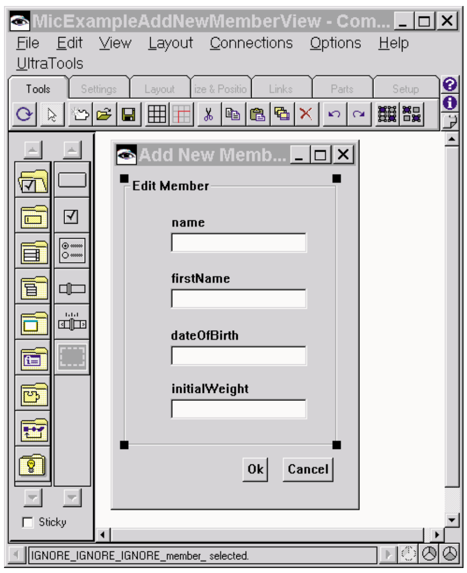
The view related code should be palced at the related Viewport MicExampleMasterProcessAtDefaultViewPort:

```
   newMember
      self model newMember openView: #MicExampleAddNewMemberView
```

2.3.5.9.4. Create a view for add new member process

Now we build the MicExampleAddNewMemberView. The buttons which commit or abort the view are controlled by the AddNewMemberProcess but the rest should be controlled by its child the member process. Therefore we use a GroupBox which can be something like a view for a process. To let a Groupbox react like a View for the member process you have to set the 4th name part of the Groupbox to the child process connection name .

So the view looks like:


MicExampleAddNewMemberView - Com...
File Edit View Layout Connections Options Help
UltraTools
Tools Settings Layout ize & Positio Links Parts Setup
Add New Memb...
Edit Member
name
firstName
dateOfBirth
initialWeight
Ok
Cancel
Sticky
IGNORE_IGNORE_IGNORE_member_ selected

The names of the parts are:

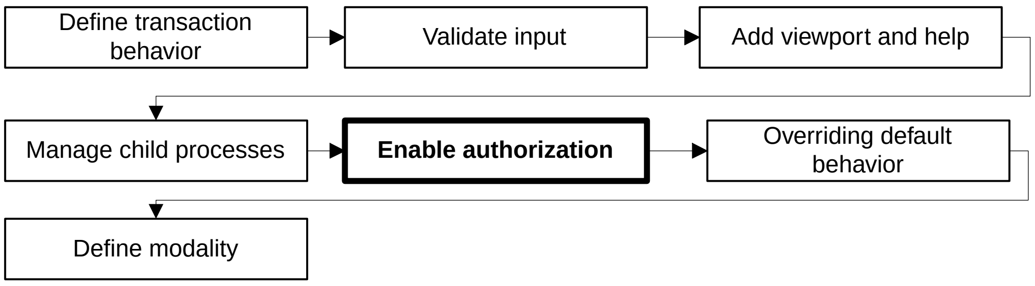
- Cancel-Button:
- Ok-Button:
- EditMember-GroupBox:

The entry fields are named like they are named at the

Hint: When the 4th name part of the Groupbox is set then the process of the groupbox starts when the view come up.

Now we can start our master process again and take a look to the transactions browser to what happens while you add a new member.

2.3.6. Authorization



2.3.6.1. Overview

2.3.6.1.1. Where to implement authorization

Authorization does not belong in an application GUI (as implemented in many 4GL tools). Authorization should be managed as far "inside" an application as possible. Benefits of this approach include the elimination of security loopholes.

2.3.6.1.2. Types of authorization

Most applications distinguish between the following 2 types of authorization mechanisms:

- Manage access to the application’s data (for a DO).
- Manage the right to execute certain functions (for a DP)

This tutorial demonstrates both types of authorization.

2.3.6.1.3. Required applications

Authorization requires that application **micApplication Service – Authorization Runtime** is loaded.

2.3.6.1.4. Types of authorization in this tutorial

In this tutorial, 3 types of access to user information will be implemented:

- **canRead**: The user can read data.

- **canModify**: The user can add or modify data.
- **isCoach**: The user is a coach and can change the 'initial weight' attribute of a member.

To implement this, we need an object that:

- Identifies a user
- Checks the password
- Determines the user's access rights.
- Knows the user name
- Knows the user password
- Knows the user security level (to avoid frequent checking of the user's security status)

2.3.6.2. Create MicFwDomainObject>>MicExampleUser

Create MicFwDomainObject>>MicExampleUser with the following attributes:

- **canModify** (boolean)
- **canRead** (boolean)
- **isCoach** (boolean)
- **name** (String)
- **password** (String)

2.3.6.3. Create MicFwDomainObject>>MicExampleSession

For Authorization implementation, the DO must know at all times who the currentUser is. This is accomplished with a MicFwDomainObject>>MicExampleSession.

- Create **MicFwDomainObject>>MicExampleSession**.
- Create **MicExampleSession>>currentUser**. Type as:
  - **transacted**
  - **primitive ->1 relationship** to a **MicExampleUser**
  - **cardinality 0..1**

2.3.6.4. Create MicExampleClubMember>>currentUser

- Create **MicExampleClubMember>>currentUser**. Type as:
  - **transacted**
  - **primitive ->1 relationship** to a **MicExampleUser**
  - **cardinality 0..1**

2.3.6.5. Create MicFwDomainObject>>MicExampleAuthorizedDomainObject

Instance variable <session> must be added to DO's. Therefore, the parent class for MicExampleClub and MicExampleMember DO's will be reset to MicExampleAuthorizedDomainObject.

- Create **MicFwDomainObject>>MicExampleAuthorizedDomainObject**
- Create **MicExampleAuthorizedDomainObject>>session**. Type as:
  - **transacted**
  - **primitive ->1 relationship** to a **MicExampleSession**
  - **cardinality 0..1**
- Redefine parent class: **MicExampleAuthorizedDomainObject>>MicExampleClub**
- Redefine parent class: **MicExampleAuthorizedDomainObject>>MicExampleClubMember**

2.3.6.6. Create MicFwDomainProcess>>MicExampleAuthorizedProcess

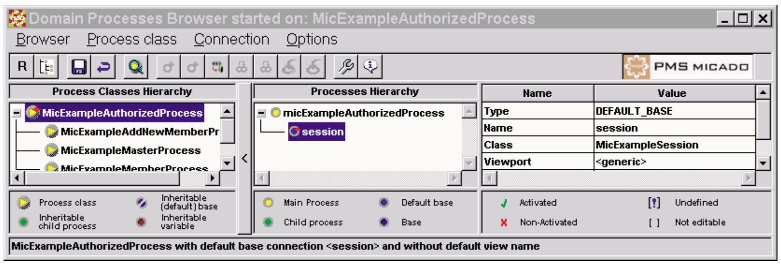
Any process that

- edits a member
- has a child process that edits the member

must initialize the child process or club member with a user. This includes several processes; therfore, a new parent process will be created.

- Create **MicFwDomainProcess>>MicExampleAuthorizedProcess**
- Redefine parent class: **MicExampleAuthorizedProcess>>MicExampleMasterProcess**
- Redefine parent class: **MicExampleAuthorizedProcess>>MicExampleAddNewMemberProcess**
- Redefine parent class: **MicExampleAuthorizedProcess>>MicExampleMemberProcess**
- In **DPB**: Create **MicExampleAuthorizedProcess base connection <session> to MicExampleSes-**

sion as shown in the following diagram:



Name	Value
Type	DEFAULT_BASE
Name	session
Class	MicExampleSession
Viewport	<generic>

MicExampleAuthorizedProcess with default base connection <session> and without default view name

The following methods are required to initialize the DP’s and DO’s with the session:

```
MicExampleAuthorizedProcess >>initChild: aProcess
  aProcess session: self session

MicExampleAddNewMemberProcess >>transactionStarted
  "Create new member and open member process on it"
  |theMemberProcess|
  self initChild: (theMemberProcess := self member).
  self newMember: MicExampleClubMember new.
  theMemberProcess clubMember: self newMember.

MicExampleMasterProcess >>newMember
  |theAddToClub|
  self initChild: (theAddToClub := self addNewMember).
  ^theAddToClub
    club: self club;
    yourself

MicExampleMasterProcess >>edit
  "edit the club member that is currently selected in the parent"
  |theMemberProcess|
  self initChild: (theMemberProcess := self member).
  ^ theMemberProcess
    clubMember: self currentMember;
    yourself

MicExampleMasterProcess>>transactionStarted
  super transactionStarted.
  self club session: self session.
  self club members do:[:eachMember | eachMember session: self session]

MicExampleMemberProcess >>transactionStarted
  "Hook for initializations immediately after starting a transaction."
  self clubMember session: self session
```

The session must be initialized with a user:

```
MicExampleSession>>initialize
  super initialize.
  self currentUser: MicExampleUser new
```

The user must be initialized:

```
MicExampleUser>>initialize
  super initialize.
  self password: '';
  name: ''.
```

The AuthorizedDomainObject must be initialized:

```
MicExampleAuthorizedDomainObject>>initialize
  super initialize.
  self session: MicExampleSession new
```

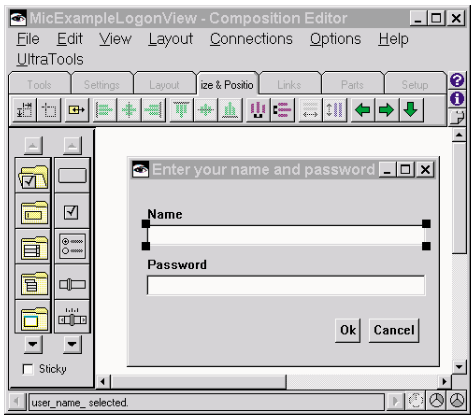
2.3.6.7. Log on

The logon procedure for this tutorial will be kept as simple as possible:

- A user with a userName longer than two characters can read data.
- A user with a password that matches his user ID reversed (e.g. user 'ABC' with password 'CBA') can modify data.
- A user 'JOE' with password 'EOJ' is the coach.

2.3.6.7.1. Create logon View

The view shown in the following diagram will be used for the logon.


MicExampleLogonView - Composition Editor
File Edit View Layout Connections Options Help
UltraTools
Tools Settings Layout ize & Positio Links Parts Setup
Enter your name and password
Name
Password
Ok
Cancel
Sticky
user_name_ selected.

The view includes the following:

- Entry field with name 'user_name_'
- Entry field with name 'user_password_'
- Button with name 'micExampleLogonProcess_commitAndCloseProcess_'
- Button with name 'micExampleLogonProcess_abortAndCloseProcess_'.

2.3.6.7.2. Create logon process

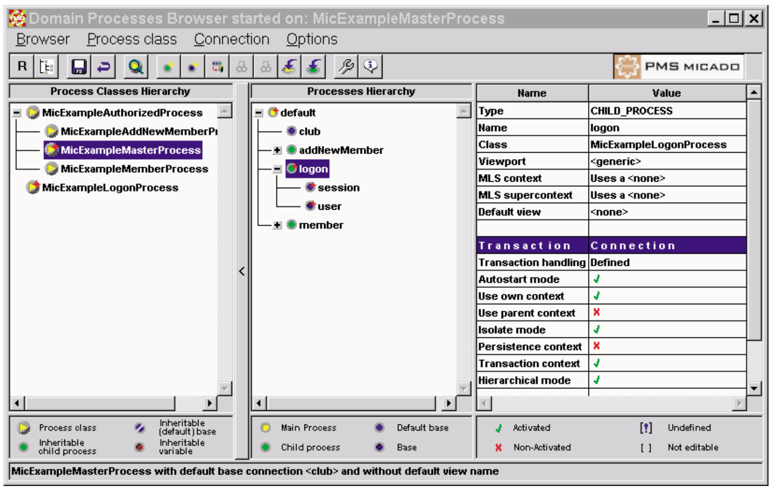
Create **MicFwDomainProcess subclass: #MicExampleLogonProcess**

In the **DPB**:

- Specify **MicExampleLogonProcess child process MicExampleMasterProcess**.
- Specify **MicExampleMasterProcess child process MicExampleSession** (for access to user).
- Specify **MicExampleMasterProcess child process MicExampleUser** (for access to user).

The session is passed by the parent process. The User has to be created by the logon process itself. The default view will of course be the we just defined.

These setting are shown in the following diagram:



Name	Value
Type	CHILD_PROCESS
Name	logon
Class	MicExampleLogonProcess
Viewport	<generic>
MLS context	Uses a <none>
MLS supercontext	Uses a <none>
Default view	<none>
Transaction	Connection
Transaction handling	Defined
Autostart mode	✓
Use own context	✓
Use parent context	✗
Isolate mode	✓
Persistence context	✗
Transaction context	✓
Hierarchical mode	✓

MicExampleMasterProcess with default base connection <club> and without default view name

2.3.6.7.3. Methods to start logon process

The DP needs to be initialized and opened by its parent. This requires methods to:

- Initialize the child process with the parameter (this method is domain code and should be implemented at the master process):

```
MicExampleMasterProcess>>initializeLogon
```

```
   ^self logon
     session: self session;
     yourself
```

- Open the view of the logon process (should be implemented at the viewport of the master process):

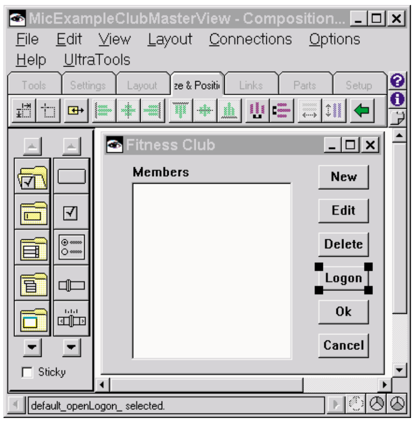
```
MicExampleMasterProcessAtDefaultViewPort>>openLogon
   ^self model initializeLogon openView: #MicExampleLogonView
```

When the view opens, the transactionStarted message is sent by the child process MicExampleLogon-Process. This method can be overridden to complete the initialization:

```
transactionStarted
   "start the receiver"
   |aNewUser|
   self user: (aNewUser := MicExampleUser new).
   self session currentUser: aNewUser.
```

2.3.6.7.4. Modify Club view

Add a button on the Club view for the logon:



default_openLogon_ selected.

The 'Logon' button is named 'default_openLogon_'.

In the DPB: Define the connection to

2.3.6.8. Authorization rules

Now that we have gone through so many preliminary steps, what about the real authorization mechanisms? They are actually very straightforward and far less trouble than our primitive user management scheme.

2.3.6.8.1. Attribute authorization API

Domain objects (actually their classes, as authorization is class based) understand the following basic messages to define access rights:

- **authorizeReadUsing: aRule**. Authorize read access to all the object's Aspects based on the result of evaluating <aRule>. <aRule> is either a block or a message which can have a value of 'true' (to allow access) or 'false' (to deny access)
- **authorizeRead: anAspect using: aRule**. Authorize read access to <anAspect> of a Domain Object based on the result of evaluating <aRule>
- **authorizeRead: anAspect using: aRule ifProhibitedReturn: value**: Authorize read access to <anAspect> of a Domain Object based on the result of evaluating <aRule>. If the access is denied, return <value>
- **authorizeWriteUsing: aRule (same as above for write access)**. AuthorizeWrite: anAspect using: aRule (same as above for write access)

2.3.6.8.2. Implementing attribute authorisation rules

We said that only a user who has a name which is spelled with more than two letters should be able to read data at all.

A User can write if he can read and has a password that is same as his name if you read it reverse.

A User is permitted to change the initialWeight of a member if he is the coach. The attributes are that his name is 'JOE' and that he has the right to modify.

To make it easier to handle later on we implement some method canRead at our User:

```
MicExampleUser >>canRead
```

```
  ^(self name size > 2)

MicExampleUser >>canModify
  ^(self name = self password reverse) & self canRead

MicExampleUser >>isCoach
  ^self canModify & (self name = 'JOE')
```

Because this behavior is the same for all authorized Domain Object classes in the example, we can define it in class:

```
initializeAuthorizationTable
  "allow read access to all of the domain object's aspects
   only for authorized users"
  super initializeAuthorizationTable.
  self authorizeReadUsing: [:aModel| aModel session currentUser canRead].
  self authorizeRead: #session using: [true]
```

We also stated that a club member’s initial weight can only be entered by someone who qualifies as a coach. This is reflected in the class method:

```
initializeAuthorizationTable
  "allow write access to the initial weight aspect only for coaches"
  super initializeAuthorizationTable.
  self
    authorizeWrite: #initialWeight
      using: [:aModel| aModel session currentUser isCoach].
      "only a coach is allowed to enter the initial weight"
```

2.3.6.8.3. Action authorization theory

The protocol for Domain Process authorization is also class based. The messages to define authorization are:

- **authorizePerformUsing: aRule**. Allow execution of all Aspects of the receiver's class based on the result of evaluating the block (or message) <aRule>.
- **authorizePerform: anAspect using: aRule**. Allow execution of < > of the receiver's class depending on <aRule>.

2.3.6.8.4. Implementing action authorization rules

We stated earlier that only users with the right to modify data should be able to add and delete members. The following method in MicExampleMasterProcess sets authorization accordingly:

```
initializeAuthorizationTable
  "allow creating/deleting members only for users who can modify data"
  super initializeAuthorizationTable.
  self authorizePerform: #newMember
    using:[:aModel | aModel session currentUser canModify].
  self authorizePerform: #deleteMember
    using:[:aModel | aModel session currentUser canModify]
```

In authorized Domain Processes, we want to allow committing changes only for authorized (i.e. with the attribute 'canModify') users. The method to initialize authorization is:

```
initializeAuthorizationTable
  "allow perform access to the commit aspect
   only for users who can modify data"
  super initializeAuthorizationTable.
  self authorizePerform: #commit
    using: [:aModel | aModel session currentUser canModify]
```

2.3.6.8.5. Initialize Authorization rules

You have to make sure that all these methods are invoked once when the application is initialized:

```
MicExampleAuthorizedDomainObject allSubclasses
  do:[:aClass| aClass initializeAuthorizationTable].
MicExampleAuthorizedProcess allSubclasses
  do:[:aClass| aClass initializeAuthorizationTable].
```

Now you're ready to test the example.

But it is a good idea to reinitialize our current club. Before we start the application again we execute:

```
MicExampleClub current: nil.
```

Now we can start the application:

```
MicExampleMasterProcess new
  club: MicExampleClub current;
  session: MicExampleSession new;
  openView: #MicExampleClubMasterView
```

2.3.6.9. Validate if edit and delete a member is possible

There are not only authorization rules which hinders the user to execute an action. You can use validation rules as well. Here we need validation rules for the method   for our master process because this method needs the selected member to work on. If no member is selected a call of the method causes a walkback. So we implement rules which check if a member is selected:

```
  MicExampleMasterProcess>>initializeValidationTable
    "allow editing members if a member is selected"
    super initializeValidationTable.
    self validatePerform: #edit
      using: [:aModel | aModel currentMember notNil].
    self validatePerform: #delete
      using: [:aModel | aModel currentMember notNil].
```

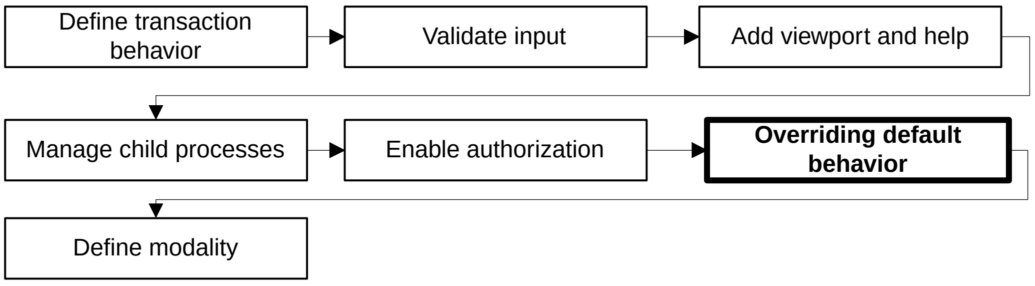
To activate the rule we have to execute:

```
  MicExampleMasterProcess initializeValidationTable
```

Before we can start our application, we have to enhance the initializeMember method at our application class to have the right to edit a member’s data. Now we can start the application again:

```
  MicExampleMasterProcess new
    club: MicExampleClub current;
    session: MicExampleSession new;
    openView: #MicExampleClubMasterView
```

2.3.7. Overriding Default Behavior



One of the goals of the Application Framework development was to provide as much default behavior as possible. However, not every user might like this default behavior, which is why it can be overwritten wherever in many cases.

In this example, we’ll demonstrate how to change the default behavior of Action Controls (buttons, in this case) with respect to authorization. The default behavior, as you have noticed in the last section, is to disappear when the user has no right to perform the associated command.

This may not be desirable for all applications.

2.3.7.1. Change default display behavior for authorized actions

You may decide that you’d rather just disable the Controls in question instead of making them disappear. Let’s do this for the fitness club example.

The place to convert domain behavior into display behavior is the Viewport . The Viewport we need is the

.

Viewports are being sent a message to determine whether or not a Control associated with a specific Aspect of the underlying Domain Process will be visible. This message can be reimplemented to provide application-specific behavior. In our case, we want to stop the Controls from disappearing, so we’ll always have to keep them visible. The implementation for this message consequently is:

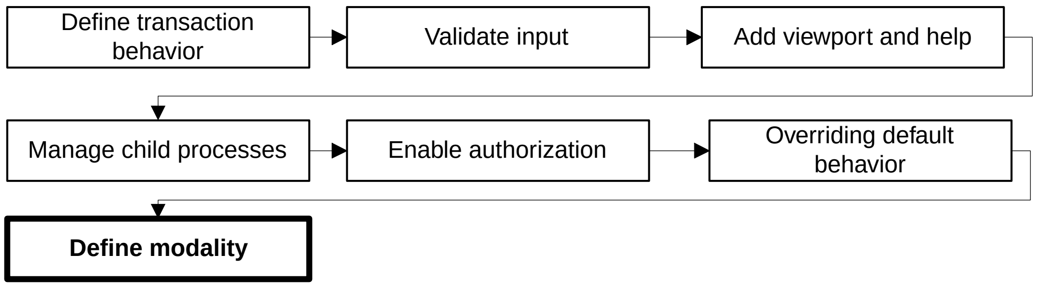
```
defaultVisibleValueOf: dispatcherAspect
  "we want controls to be visible no matter
   if access to them is allowed or not"
  ^true
```

However, we want the Controls to be disabled when access is prohibited. The mechanism for enabling / disabling is similar to that for visibility:

```
defaultEnabledValueOf: dispatcherAspect
   "we want to enable controls only when the
    user is authorized to perform the associated actions"
   ^(super defaultEnabledValueOf: dispatcherAspect)
     and:[self model permitsPerformAccessTo: dispatcherAspect]
```

Now we can start the application again using the old workspace.

2.3.8. Modality



Modality is defined as the (dis)allowance for the user to switch from the current view to another window. A modal view requires input from the user before the application will continue.

A non-modal view will remain on the screen but will allow continuation of work in other windows. Views will be started as non-modal views by default.

The modality-state can be refined as

- **SemiModal**( ). Other applications can be started or activated, restrictions apply only to the parent of the current view.
- **ApplicationModal**( ). Other applications can be started or activated, restrictions apply only to views of the current application.
- **SystemModal**( ). No other applications can be started or activated.

Views can be started as modal views by implementing the class method in your Domain Process. This method has the following constants as possible return values:

- MicFwSystemModal
- MicFwFullApplicationModal
- MicFwPrimaryApplicationModal.

2.4. Quick Tour of Tools

This chapter provides an overview of the frameworks browsers' capabilities in application development and debugging. To show this we will develop a short example which has an intentional bug implemented.

The **Object Net Browser** and the **Domain Processes Browser** support application development.

The **Framework Logger,** the **Connectors Browser**, and the **Transactions Browser** make debugging easier. These browsers allow the developer to monitor the activities of a running application.

The **Connectors Browser** displays the runtime result of the declaration with the **Domain Processes Browser**.

The **Transactions Browser** supplies information about the Transaction Contexts during runtime. The hierarchy of the contexts and changes in the instance variables can be made transparent. The Transaction Contexts are also declared with the help of the **Domain Processes Browser**.

2.4.1. The Example

We will try to implement the following scenario :

- At the beginning, a view opens displaying one Push Button.
- After engaging the control panel, a second view with an empty entry field and a List filled with names opens
- If something is entered in the entry field and the entry field is left, then the entry will be added to the List unless it already exists.
- If a line is selected in the List, then the selection should appear in the editing field.
- The actions should run in a transaction and the transaction should be able to be aborted with the abort-button.
- The transaction should be finalized by clicking the OK-button.

The example consists of

- two view classes
- DemoStartView
- DemoView
- two Domain Process classes
- DemoStartProcess
- DemoProcess
- two Domain Object classes
- DemoBase
- DemoInitBase
- and one Viewport class
- DemoViewPort.

Occurrences of development errors are marked with an exclamation point (!).

2.4.2. Development

The environment for this example is VisualAge for Smalltalk and the Composition Editor.

2.4.2.1. Preliminary Activities

Before we can use the development browsers, it is necessary that the classes developed in the design phase are placed with the Class Browser in a subclass of and that the view is created with the GUI-Builder.

For this project, an entry template is created with a button which starts a second view with an entry field, a List, two Labels and two Push Buttons.

The class of the first view is named

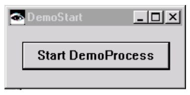
The class of the second view is named

Both view classes are platform-specific and are customarily installed in the respective platform.

The following was declared regarding the names for the widgets:

2.4.2.1.1. DemoStartView:

Parttype <Label>	Partname
Button <StartDemoProcess>	childProcConn_openView_



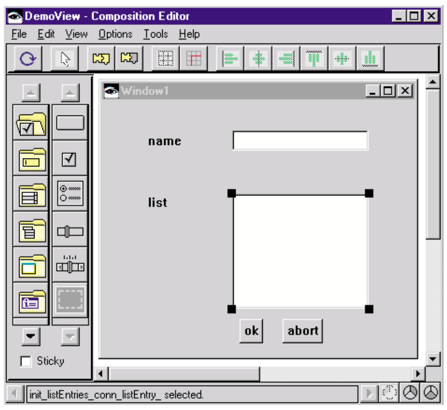
2.4.2.1.2. DemoView:

Parttype <Label>	Partname
Label <name>	conn_nameLabel_
Label <list>	listLabel_ (!)
TextEntryField	name_ (!)
Listbox	conn_listEntries_conn_listEntry_ (!)
Button <ok>	commitAndCloseProcess (!)
Button <abort>	abortClose_ (!)

The classes and are implemented as subclasses of , the classes and as subclasses of , and the class as a subclass of .

In order to create the dependency of the class to its Domain Object, the class method is implemented with the name of the corresponding Domain Object:

```
DemoViewPort >> portName
   ^'DemoBaseObject' "(!)"
```



init_listEntries_conn_listEntry_ selected.

2.4.2.2. Declaring Variables and Typing

With the **Object Net Browser** you can declare variables of Domain Object and Domain Process classes.

A Domain Object is referred to as an instance of a subclass of . Domain Processes are instances of subclasses of .

The **Object Net Browser** is activated using the Menu **micFrameworks** → **Browse Object Net** of the System Transcript. In the appearing dialog you can filter the list of Domain Object and Domain Process classes. If you accept the wildcard (*), all Domain Object classes and Domain Process classes will be shown in the following dialog box. The browser is opened by double clicking on the selected class; select

.

Add a variable named to the class by selecting from the Menu **Variable** → **Add Instance Variable**. The variable is added to the class in the same manner. The instance variables appear in the right pane.

The marked variable can be typed by clicking the right mouse button and selecting the Menu item **Set Type** in the Popup Menu. The object type for this instance variable is selected through the now opened Type Editor. The instance variable receives Float (!), as the valid object type and set the validate flag.

Type the variable to an OrderedCollection.

After typing, the instance variables are validated in the Application-Framework by activating the option **Validate Type** via the check-box.

Finally, the transact-flag should be set for every attribute.

The settings are made persistent via the Menu **Class** → **Save all Changes**, i.e. the declared instance variables, the corresponding access methods and the class method will be generated.

2.4.2.3. Declaring Views, Connections and Transactions

With the **Domain Processes Browser** the following tasks can be handled:

- Connecting the Domain Process with its Child Process
- Connecting the Domain Object Classes
- Connecting Domain Processes with Views
- Setting a Transaction Context

To start the **Domain Processes Browser** on the , select **micFrameworks -> Open Domain Processes Browser** in the System Transcript Menu. In the appearing dialog you can filter the Domain Processes classes shown in the following dialog box. Here, double clicking on will open the **Domain Processes Browser**. The selected class appears in the left and middle pane; its inheritance tree up to and including the superclass, which is a subclass of is also displayed.

The Default Process Connection is already set with the Connection Name of the Domain Process in the second row in the right pane of the browser, in our case with the Connection . The name of the Default Process Connection corresponds by default with the name of the assigned Domain Process class.

The **transaction behavior** of the selected Domain Process is displayed in the lower area of the right pane. You can decide whether the Domain Process can participate in transaction handling. By clicking the **Transaction handling** row and selecting the ‘Defined’-radio button, further options are made available. (To set transaction behavior see ‘Defining Transaction Behavior’ in the Domain Processes Browser-chapter).

The Domain Process and the declared Default Base Connection appear in the **Status Line** on the bottom of the browser. Further information on the status line is the hint ‘ , which means that for this Domain Process no view has yet been defined.

This can be changed by clicking on the **Default View** Definition row in the 7th row of the right pane, where a drop down box opens a selection list of the available views.

No let’s do the following:

2.4.2.3.1. Connect the Domain Process *DemoStartProcess* with the child Domain Process *DemoProcess*

To create a child Domain Process click the right mouse button in the middle pane; select the item **Add child connection from the appearing drop down menu. There, select the class *DemoProcess and click OK*. In the second row of the right pane change ‘newChildProcessConnection’ to ‘childProcConn’.**

2.4.2.3.2. Connect the *DemoProcess* with the Domain Object classes *DemoInitBase* and *DemoBase*

In order to connect the Domain Objects to first select the in the left pane. In the middle pane click the right mouse push button. Select ‘**Add base connection**’ from the drop down menu. There, select the Domain Object . In the second row of the right pane change the Connection name to ‘conn’. This first Connection is automatically the Default Base Connection for .

The Default Process Connection is defined by the edited Domain Process itself and cannot be changed. The Default Base Connection can be declared. After declaring the second Base Connection with the name and the Domain Model , this Connection becomes the Default Base Connection by selecting ‘**Set as default base**’ from the drop down menu of the middle pane.

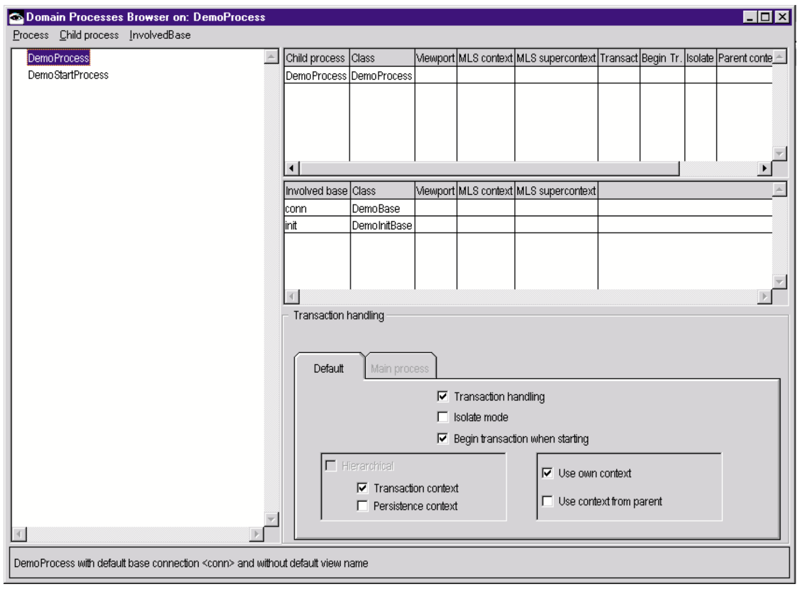
2.4.2.3.3. Connect the Domain Processes with their view classes

To connect a Domain Process with its view class, choose the Domain Process . In order to assign a view class, select the 7th row of the right pane. Open the drop down box and select from the list of available views.

Now connect the to the in the same way.

2.4.2.3.4. Setting of a simple Transaction Context

It is not necessary to assign a Transaction Context to the Domain Process because no Domain Object is directly changed in it.



Child process	Class	Viewport	MLS context	MLS supercontext	Transact	Begin Tr.	Isolate	Parent conte
DemoProcess	DemoProcess							

Involved base	Class	Viewport	MLS context	MLS supercontext
conn	DemoBase			
init	DemoInitBase			

- ☑ Transaction handling
- ☐ Isolate mode
- ☑ Begin transaction when starting
- ☐ Hierarchical
  - ☑ Transaction context
  - ☐ Persistence context
- ☑ Use own context
- ☐ Use context from parent

DemoProcess with default base connection <conn> and without default view name

A Transaction Context is assigned to the Domain Process . This process should use its own Transaction Context (Use Own Context). Also it behaves as a child process transaction, so it should be set by the Connection of the parent process .

As shown in the picture, select in the left pane, then select childProcConn in the middle pane (the childProcConn-Connection refers to the). In the right pane, change the ConnectionTransaction handling to ‘ ’ and finally change the isolate mode to ‘ ’.

2.4.2.3.5. Finalization

All declarations made in the Domain Processes Browser refer to the selected Domain Process. Selecting the Menu item **Save All Changes** in the **Browser** Menu compiles the current state of declarations into that Domain Process’ class method . This is performed for all of the Domain Processes which have unsaved changes in the current browsing session.

The method createApplicationDescription should not be modified or deleted, since all changes will be lost when **Save All Changes** is performed from the Domain Processes Browser the next time.

2.4.2.4. Additional Activities

The method is implemented in the class to initialize its Domain Objects.

```
transactionStarted
   self conn: DemoInitBase new. "(!)"
   self init: DemoBase new. "(!)"
```

The method `initialize` has to be reimplemented in the class in order to supply the List with data:

```
initialize
   super initialize.
   self listEntries: ( OrderedCollection new
      add: 'Henry';
      add: 'Julia';
      add: 'Walter'; yourself)
```

The methods in the Viewport must also be implemented with the Class Browser:

```
listEntry
”this method set the selection of the list. List selections are Collections (in
contrast to choice controls)So this method have to return a Collection”
  ^(OrderedCollection new)add: self model name; yourself
```

This method answers the name of the Domain Object to the List.

```
listEntry: anOrderedCollection
  ( anOrderedCollection isNil or:
    [ anOrderedCollection isEmpty ] )
    ifTrue: [ ^ self ].
      self model name: anOrderedCollection first.
```

This method places the selected name from the List into the Domain Object.

```
listLabel
  ^'ourList'

nameLabel
  ^'ourName'
```

These two methods place the Label text into the view.

```
name: aString
  self model name: aString.
  ( self model listEntries includes: aString )
  ifFalse:”(!)” [ self model listEntries:
    ( self model listEntries add: aString; yourself ) ]”(!)”
```

The method places the name entered in the Text field into the model. If the name is not yet in the List, it will be added.

2.4.3. Testing and Debugging

2.4.3.1. Starting the Demo

This example can be started from the workspace with the following command:

```
DemoStartProcess new openView
```

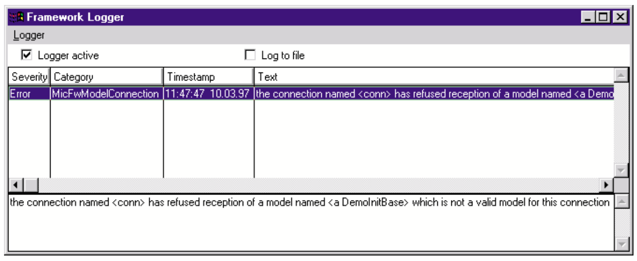
2.4.3.2. Logging Protocol

After starting the demo application and clicking the button "Start DemoProcess" a walkback takes place. In the Debugger, the following message appears:

'Inv1alid Model: The instance <a DemoInitBase> is not a valid model for the connection named <conn> which requires a <DemoBase>'.

Further information can be obtained through the **Framework Logger**. To start the logger, select the Menu item **Browse Log** from the Menu **micFrameworks.** The Logger offers insight into the processes in the framework.

The reason for the walkback described above is recorded with the Severity #Error in the Logger. The text states that the attempt was made to assign a different Domain Object to the Connection , that was declared in the DomainProcessesBrowser.



```
Severity  Category               Timestamp           Text
Error     MicFwModelConnection   11:47:47  10.03.97  the connection named <conn> has refused reception of a model named <a Demo

the connection named <conn> has refused reception of a model named <a DemoInitBase> which is not a valid model for this connection
```

The linking of new Domain Objects to Connections took place in the reimplemented API method of the class . Reviewing the method's code it shows that the Domain Objects were switched. The attempt was made to link the object to the Connection and the object to the Connection .

After switching the two Domain Objects:

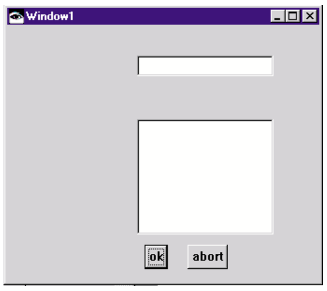
```
transactionStarted
```

```
   self conn: DemoBase new.
   self init: DemoInitBase new.
```

it is possible to open the DemoView.

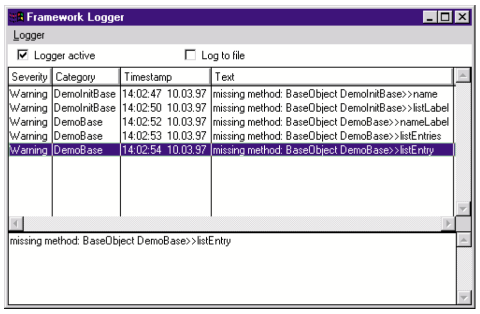
2.4.3.3. Browsing Aspect Mappings

Now the Debugger tells you that a method named nameLabel was missed, after clicking on the button **StartDemoProcess**. After resuming (click the resume button in the debugger tool) a lot more missing accesssors are notified. Then the view appears.



No data appears in the List even through the method in the class was re-implemented. Furthermore, the Label widgets display nothing instead of and .

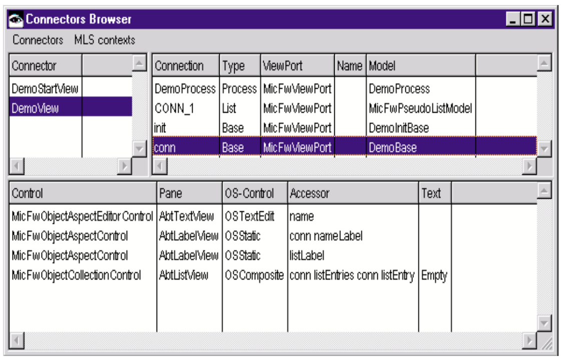
When viewing the Framework Logger, five different #Warnings appear in addition to the Severity #Info. The associated text reveals that the accessors , , and were not found when starting the Domain Process for the corresponding widgets.



Severity	Category	Timestamp	Text
Warning	DemoInitBase	14:02:47 10.03.97	missing method: BaseObject DemoInitBase>>name
Warning	DemoInitBase	14:02:50 10.03.97	missing method: BaseObject DemoInitBase>>listLabel
Warning	DemoBase	14:02:52 10.03.97	missing method: BaseObject DemoBase>>nameLabel
Warning	DemoBase	14:02:53 10.03.97	missing method: BaseObject DemoBase>>listEntries
Warning	DemoBase	14:02:54 10.03.97	missing method: BaseObject DemoBase>>listEntry

```
missing method: BaseObject DemoBase>>listEntry
```

After rebooting the application from the workspace, the List is filled correctly.



Connector
DemoStartView
DemoView

Connection	Type	ViewPort	Name	Model
DemoProcess	Process	MicFwViewPort		DemoProcess
CONN_1	List	MicFwViewPort		MicFwPseudoListModel
init	Base	MicFwViewPort		DemoInitBase
conn	Base	MicFwViewPort		DemoBase

Control	Pane	OS-Control	Accessor	Text
MicFwObjectAspectEditorControl	AbtTextView	OSTextEdit	name	
MicFwObjectAspectControl	AbtLabelView	OSStatic	conn nameLabel	
MicFwObjectAspectControl	AbtLabelView	OSStatic	listLabel	
MicFwObjectCollectionControl	AbtListView	OSComposite	conn listEntries conn listEntry	Empty

The Connectors Browser is started in the Menu **micFrameworks** using the Menu item **Browse View Connectors**. The Connectors Browser has two drop down Menus. In the **Connectors** Menu you can update the content of the browser. The **MLS contexts** Menu is only relevant when using the MLS system.

The opened Connector appears in the left List with the name DemoView, the class name of the view. In the right List, the Connections are displayed along with information on the models connected. These Connections are the ones defined in the Domain Processes Browser. In the List below, information regarding the widgets appears.

After running the example, appears in the Connector column in the left List. After selecting the entry, the Connections which have been defined in this Connector will appear in the right List with its models. The type of the models appears in the column **Type**.

- One of this Connections is the Default Process Connection as it was pre-defined in the Domain Processes Browser. The corresponding model displayed in the most right column has no user defined Viewport. Thus, the name of the Viewport class is which can be seen in the Viewport column. This Viewport was defined by the framework.
- The second Connection, has (like the Connection a generic Viewport from the class , even though the Viewport was defined. The reason for this is, that the instance from the class cannot find a Viewport which returns the string 'DemoBase' in the class method . Therefore the Viewport class method must be corrected to:

```
DemoViewPort>>portName
    ^'DemoBase'
```

- An other entry of the Connection column is init, the Default Base Connection.
- The last Connection was created at run time of the framework for the List, which is shown in the column **Type.** Clicking on this item the corresponding List Control appears in the control list of the Connectors Browser.
- At the Accessor column you can check which methods were called by the List when it want to fill its contents and the selection.
- Remembering the naming convention, you know that the control asks its content by getting the model using the first name part and gets its content at this model using its second name part.
- The second name part here is but the model at the Connection is and DemoBase has not implemented the method that returns the OrderedCollection of objects which should be transferred to the List. The Connection has the information for the List. The List widget, however, is searching for them in the Connection . The widget name defines the Connection the data comes from. Therefore, the name of the widget should be changed to
- After clicking on the Connection , information about the Aspects of the view appears in the List below. In the left column below the name Control, is listed. This class is the abstract class of the framework which "observes" the real Widget . The real Widget appears in the column 'Pane', the platform widget in the 'OS-Control' column. The assignment of a widget to a Domain Process takes place through the defined accessor name for the widgets in the GUI-Builder.

Linked to the Domain Object you will find the widgets and to which the messages

```
missing method: BaseObject DemoInitBase >> listLabel
missing method: BaseObject DemoInitBase >> name
```

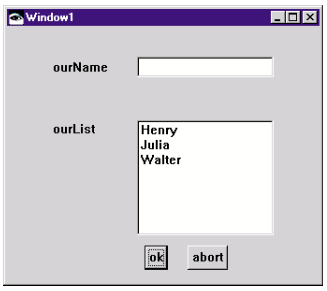
were referred in the Framework Logger. This error is caused by the incorrect assignment of the widget through the naming convention. If you exit the entry field , a walkback will be initiated with the message

```
a method named <name:> was missed in the class DemoInitBase
```

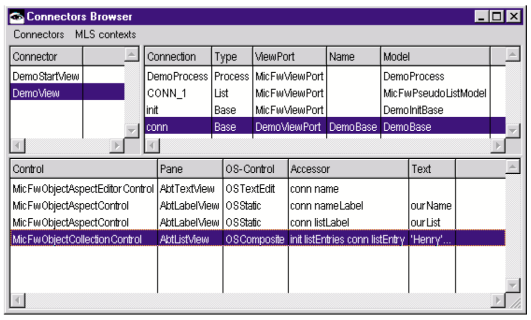
The widget must be assigned to the accessor which is located in the Viewport of the Domain Object . This Domain Object is linked to the Connection and not to the as it is presumed when entering the widget name and leaving out the Connection Name . The correct widget name must be .

The entry field must also be attached to the Connection . The read accessor method is defined in the Domain Object and the write accessor method is defined in the corresponding Viewport. The widget name for the entry field must also have the Connection Name as prefix.

After rebooting the application from the workspace, the List is filled correctly.



When viewing the Connectors Browser, which had to be updated first (Menu Connectors -> Update), entries will no longer appear in the List below after selecting the Connection even though the List has been assigned to the Connection . The reason for this is, that the Lists are handled differently than other Controls: if the selection of a List is connected to a Model Aspect then the List is related to the selection Connection. If you select the Connection ”conn” you can discover your List Control at the Control list below. The List Control is displaying the expected information.


Connectors Browser

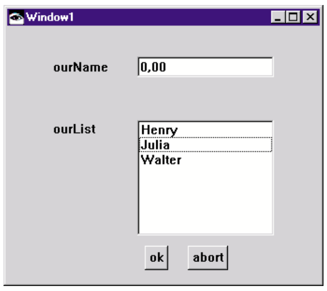
Connectors MLS contexts

Connector
DemoStartView
DemoView

Connection	Type	ViewPort	Name	Model
DemoProcess	Process	MicFwViewPort		DemoProcess
CONN_1	List	MicFwViewPort		MicFwPseudoListModel
init	Base	MicFwViewPort		DemoInitBase
conn	Base	DemoViewPort	DemoBase	DemoBase

Control	Pane	OS-Control	Accessor	Text
MicFwObjectAspectEditorControl	AbtTextView	OSTextEdit	conn name	
MicFwObjectAspectControl	AbtLabelView	OSStatic	conn nameLabel	ourName
MicFwObjectAspectControl	AbtLabelView	OSStatic	conn listLabel	ourList
MicFwObjectCollectionControl	AbtListView	OSComposite	init listEntries conn listEntry	'Henry'...

One click in the List of the application does not invoke the string to appear in the entry field. Rather, a Float with the value "0,0" displays.


Window1

ourName: 0,00

ourList: Henry, Julia, Walter

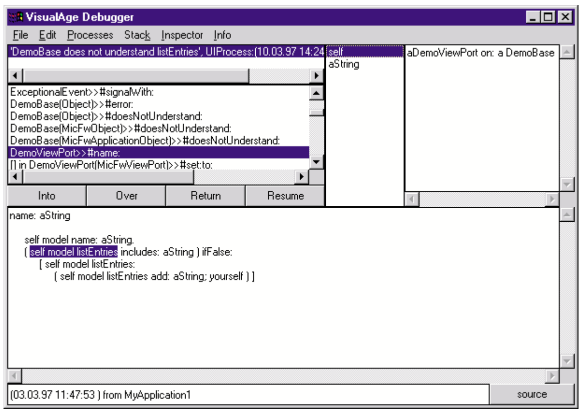
The selection, however, is a string whereby the attribute in the class expects a Float, furthermore the validation for the attribute was activated. In order to have the string displayed, it is necessary to change the type of the attribute of the class in the Object Net Browser to a Type.

Now selecting a name in the List (ourList) results being displayed in the upper entry field (ourName).

In reverse, an entry in the entry field should be entered into the List after exiting the field under the condition that the entry is not yet included in the List.

Actually if the attempt is made, the application will reply with a walkback and the message . According to the information in the **Connectors Browser,** the declaration should now be correct.

The cause of this error must be searched for in the implemented methods. After viewing the Debugger stack more closely, the origin for this walkback is the method


VisualAge Debugger

```
'DemoBase does not understand listEntries', UIProcess:(10.03.97 14:24
```

```
self
aString
```

```
aDemoViewPort on: a DemoBase
```

```
ExceptionalEvent>>#signalWith:
DemoBase(Object)>>#error:
DemoBase(Object)>>#doesNotUnderstand:
DemoBase(MicFwObject)>>#doesNotUnderstand:
DemoBase(MicFwApplicationObject)>>#doesNotUnderstand:
DemoViewPort>>#name:
[] in DemoViewPort(MicFwViewPort)>>#set:to:
```

```
name: aString

   self model name: aString.
   ( self model listEntries includes: aString ) ifFalse:
      [ self model listEntries:
         ( self model listEntries add: aString; yourself ) ]
```

```
(03.03.97 11:47:53 ) from MyApplication1
```

In it, the method is sent to the receiver . The method returns the model of the Viewport. In this case, it is the object from the class . This class does not have the method implemented; instead it has the class , which object is linked to the Connection .

The method has to be changed the following way:

```
name: aString
| mod |
self model name: aString.
mod:= self modelAtConnection: #init.
( mod listEntries includes: aString ) ifFalse:
   [ mod listEntries:
      ( mod listEntries add: aString; yourself ) ]
```

The attempt to exit the application by pressing the button is unsuccessful: the view stays open even though the widget name for the button is which itself is implemented in the framework. Viewing the Connectors Browser after selecting the Connection in the right List, one can see that there is no accessor for the with the text . This widget is consequently not connected. The result, after reviewing the widget names, is that the name does not have the mandatory underscore. The change of the widget name in leads, to the closing of the view after resetting the application.

As a final step, the button needs to be tested. After clicking the abort button, a walkback immediately occurs with the message

```
a method named <abortClose> was missed in class DemoProcess
```

The Framework Logger comments on the error with the remark: . The Connectors Browser at the Default Process Connection reveals that for the with the text the Accessor is registered as . Since buttons are always assigned to a Domain Process, the widget is assigned to the correct Connection. Therefore, it can be assumed that the widget name was not spelled correctly. As expected, the search for an method in the framework of the classes is unsuccessful. However, there is a method with the name , which is suitable for our purpose. As a result the widget has to be renamed into . After changing the widget name through the GUI-Builder, this widget is also connected. The application is finished.

2.4.3.4. Transaction Controlling

The Transactions Browser is started via the Menu **micFrameworks** with the selection **Browse Transactions**. When declaring variables with the Object Net Browser, it is determined whether or not they can participate in transaction handling. The Domain Processes Browser determines whether a Transaction Context should be automatically started for this Domain Process.

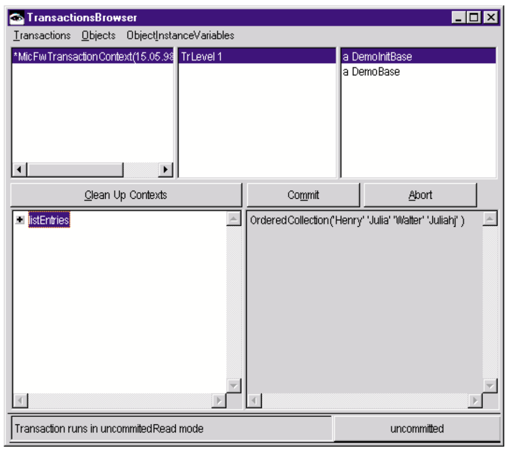
When viewing the Transactions Browser**,** You should see marked with a leading star in the left List. Otherwise, no transaction handling was arranged in the Domain Processes Browser for this Domain Process.

After the selection of the Transaction Context in the Transactions Browser, the transaction level appears in the next List. If the transaction level gets selected, the instances of the class, which take part in transaction handling, appear in the right List.

In our example, if an entry is selected in the DemoView List, then the instance of the class appears in the right List of the Transactions Browser.

After selecting in the Transaction Browser the instance variable of this class is displayed in the lower left List. Clicking the variable with the mouse lets the contents of the variable be displayed in the lower right List.

After selecting in the Transaction Browser the instance variable of this class is displayed in the lower left List. Clicking the variable with the mouse lets the contents of the variable be displayed in the lower right List.



*MicFwTransactionContext(15.05.9

TrLevel 1

a DemoInitBase
a DemoBase

listEntries

OrderedCollection('Henry' 'Julia' 'Walter' 'Juliahj')

Transaction runs in uncommitedRead mode

uncommitted

Using the committed / uncommitted button located on the status line, it is possible to trace the changes of the contents of the instance variable.

2.4.3.5. Common Problems

The information in the **Connectors Browser** makes it easier to understand unexpected behavior of the framework. Some situations are described below:

Problem 1

You have implemented a **method in a Viewport class** and you wonder why it **is not called** even though you have named a widget accordingly. Worse scenario: A Debugger is also activated with the message
.

or

You confirm a command control (for example: a button), with the result that a Debugger with the message appears, even though you have implemented the method in a Viewport.

View / Display in the Connectors Browser:

In the Viewport column of the Domain Object / Domain Process appears, even though a specific class was defined for the Domain Object / Domain Process.

Cause

When running the Domain Process the Viewport for the Domain Object / Domain Process was not found. The class method , which contains the class name of the corresponding Domain Object / Domain Process as a String,

- was not implemented,
- was implemented as an instance method or
- the name of the Domain Object / Domain Process was accidentally misspelled.
- the name of the Domain Object / Domain Process was not enclosed by apostrophes

A second possibility to connect a Viewport to a model is to declare it in the DomainProcessesBrowser at the Connection of the Domain Object / Domain Process to its parent Domain Process. You have to check this as well.

Problem 2

No objects are set in the List.

Cause

The widget name does not comply with the list convention. Either the name of the Connection for the list entry is not correct or the name of the accessor method is incorrect.

Problem 3

When entering data in the view **no changes can be observed in TransactionsBrowser's** data even though the Domain Process has started a transaction context.

View / Display in the Connectors Browser

Displayed in the column of the Base Connection in question is

Cause:

The Connection is empty. The Domain Process is responsible for attaching a valid Domain Object to the Connection (via the model write accessor).

Problem 4

When entering data in the view, **no instance variable can be seen in the TransactionsBrowser** even though its contents should have changed and the Domain Process has started a Transaction Context.

View / Display in the Connectors Browser

In the column in the List below no entry corresponding to the name of the instance variable can be seen for the Connection in question.

Cause

The name of the widget was not entered in the GUI-Builder (e.g. the of VisualAge™) according to the conventions with underlining following and was therefore not registered when starting the Domain Process.

Problem 5

When leaving the entry field the Debugger appears with the message
. In the **Transactions Browser** the instance variable cannot be seen even though its contents should have changed and the Domain Process has started a Transaction Context.

View / Display in the Connectors Browser:

For the Connection in question, a Domain Process is assigned and all accessors are registered.

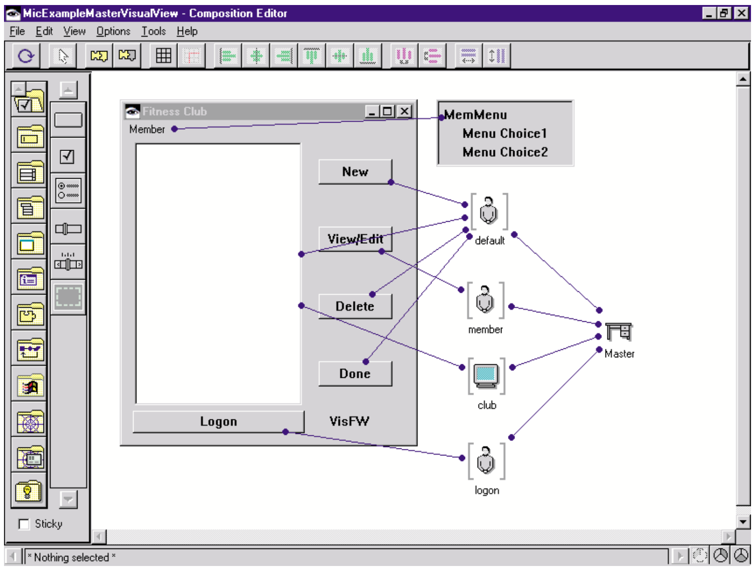
Cause

The name of the widget does not correspond to a method or instance variable of the Domain Object. Either an incorrect Domain Object was placed in the Connection or the name of the widget was incorrectly specified.

2.5. Connect to models using Non-Visual Parts

To further facilitate the process of connecting views and models as well as to provide a method of visualizing structural elements of an application, the Frameworks also extend the VisualAge Composition Editor.

Let's first look at an example of what this means in practice. The example is based on the Fitness Club application that is included as a coded example. After that, the extended capabilities of the Composition Editor will be explained in detail.



shows how the extended Composition Editor helps to visualize important Aspects of an application. These are :

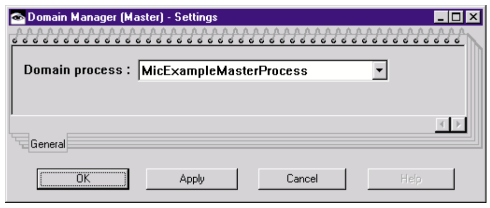
- : the icon labeled ' represents a Domain Process (in this case)
- the icons labeled ' , ' , ' and ' represent Viewports

Process hierarchy : the wires between the Viewports of Domain Processes and the master process ( ) indicate a child process relationship (e.g. 'logon' is a child process of the master process)

- the links between Viewports and GUI controls indicate a link between an Aspect of the respective Viewport and the GUI element. For example, the link from the 'Logon' button to the ' means that the button is linked to the Aspect of that (functionally equivalent to naming it ')

2.5.1. Domain Processes

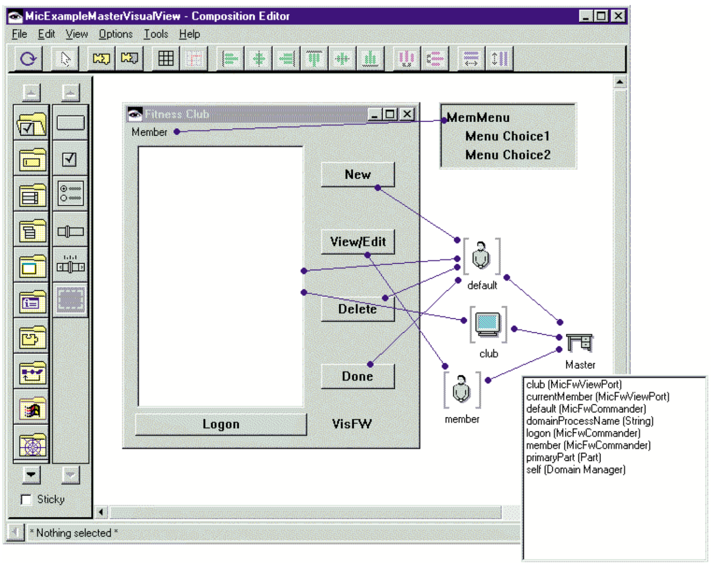
Domain Processes can be used as non-visual parts in the Composition Editor. To do so, select the section 'micado Frameworks' section in the parts catalog on the far left of the Composition Editor. Then select the symbol (Domain Manager) and place it on the canvas. Double-clicking on it will open the settings dialog, which simply consists of a Listbox allowing you to select a Domain Process from those currently defined in your image:


Domain Manager (Master) - Settings

Domain process : MicExampleMasterProcess

2.5.2. Viewports

The s associated with a Domain Process (i.e. the child processes and the involved Base Connections defined in the DomainProcessesBrowser) are accessible as tear-off attributes of the domain manager:



club (MicFwViewPort)
currentMember (MicFwViewPort)
default (MicFwCommander)
domainProcessName (String)
logon (MicFwCommander)
member (MicFwCommander)
primaryPart (Part)
self (Domain Manager)

2.5.3. Aspect Connections

The Aspects of s are represented as connectable features of the corresponding edit parts. Depending on their type, the GUI parts also have one or two extra features. The first, which is called ‘micContents’ represents the contents of the part. The other, named ‘micSelection’, represents the selection of the part. In the example in the above figure, the ‘members’ Aspect of the ‘club’ is connected to the ‘micContents’ feature of the Listbox, indicating that the Listbox will be filled with the members of the club. A detailed description of a complete example will follow in the next chapter.

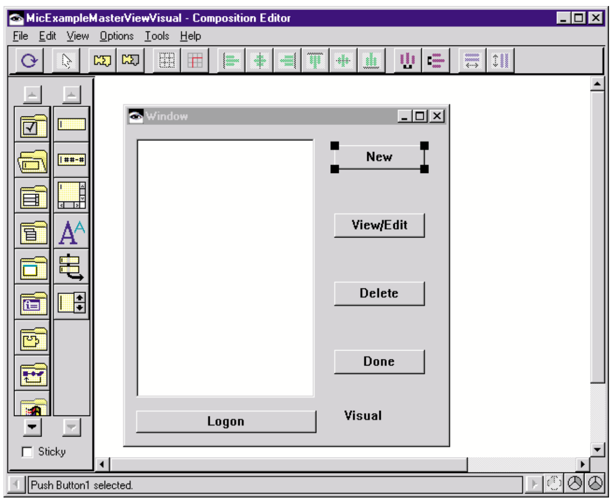
2.5.4. Example

Our objective in this chapter is to take you through an example using all the new features in the Composition Editor which we presented in the previous chapter.

The example will be an extension of the Fitness Club Example. In order to work through the example shown here, it is vitally important that you are familiar with the contents of this chapter.

We will replace the views from the Fitness Club application with views created using the new features in the Composition Editor. First, we have to create a new application to hold the code we will be creating here. Open the VisualAge Organizer and create an application named . Set the prerequisites of the new application to include (it may be necessary to activate the full Menus in the VisualAge Organizer first; this is done by selecting **Full Menus** from the **Options** Menu).

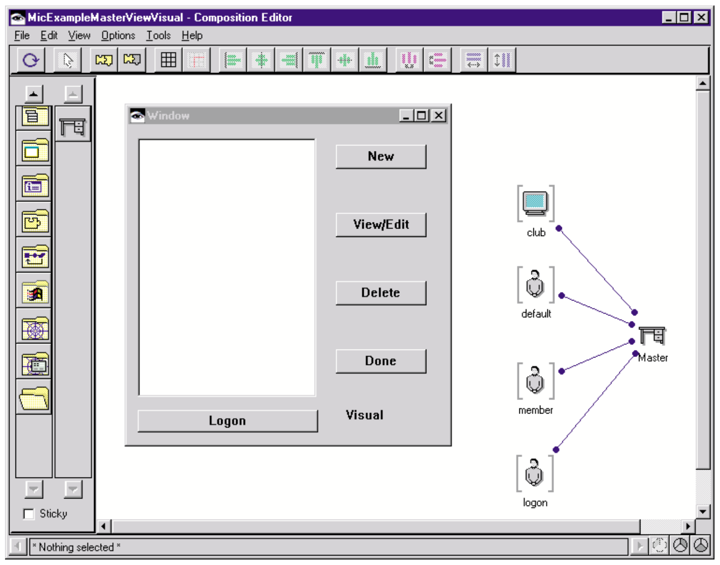
Now, create a visual part named . Place the Controls needed for the Fitness Club master Window onto the window without paying attention to the name (the system will use default names like). You will end up with a screen looking like this:


MicExampleMasterViewVisual - Composition Editor
File Edit View Options Tools Help
Window
New
View/Edit
Delete
Done
Logon
Visual
Sticky

Push Button1 selected.

Next, choose a Domain Manager part ( ) from the micado Frameworks section of the parts catalog and place it next to the Window. Double-click it and select MicExampleMasterProcess from the Drop-down List in the settings dialog. To help identify the part, you may also want to rename it to something like Master. Now, tear off the following attributes one by one : logon, club, member and default. These represent the Viewports of the associated child processes and Domain Objects respectively.

Now your screen should look like this:

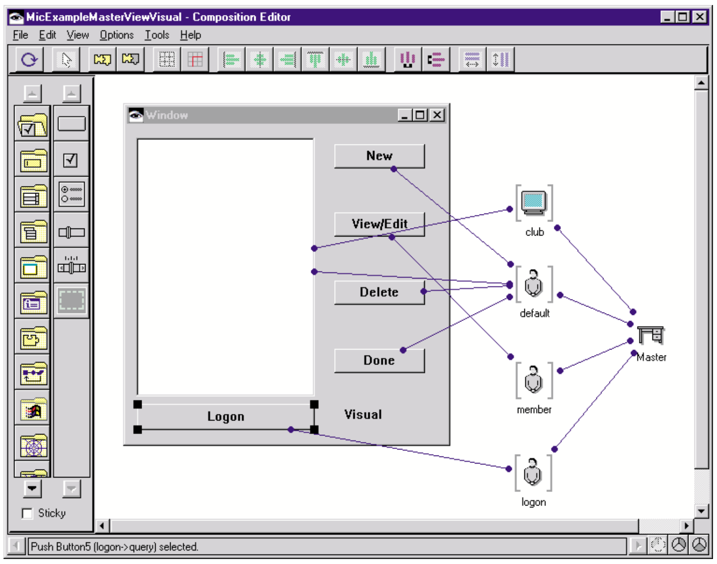

MicExampleMasterViewVisual - Composition Editor
File Edit View Options Tools Help
Window
New
View/Edit
Delete
Done
Logon
Visual
club
default
Master
member
logon
Sticky

* Nothing selected *

Now you're ready to connect the GUI parts to the s. Let's start with the button. Select the Viewport named and right-click it. From the Popup Menu, select (**Connect....**→ **newMember**) and drag the resulting wire to the button. After you drop it there, select as the feature you want to connect to (Note : if you are using VisualAge 3.0a, select instead of).

You can verify the Connection information by double-clicking the button to open the Connection settings. You can manipulate the information here, if you want.

Now connect the remaining buttons to their respective Viewport Aspects. Finally, we have to connect the List. Right-click it and select (**Connect...**→**micContents**) from the Popup Menu (Note: if you are using VisualAge 3.0a, select **items** instead of **micContents**). Connect this feature to the feature of the icon. To connect the selection of the List to the appropriate Aspect, connect on the List to on the Viewport icon (Note : if you are using VisualAge 3.0a, select instead of). The completed Window should look like this:


MicExampleMasterViewVisual - Composition Editor

Push Button5 (logon->query) selected.

We must now change the so that it will make use of the newly created view. Change the default view name with the Domain Processes Browser.

To replace the member editing screen go to the VisualAge Organizer again. Create a new visual part named MicExampleMemberViewVisual. Select a Domain Manager part ( ) from the **micado Frameworks** section of the parts catalog and place it next to the Window. Assign it the Domain Process named MicExampleMemberProcess and rename it to Member.

Now place an entry field inside the Window. Double-click it to get to the settings dialog window. From the Drop-down List in the section select . Type for the at the settings at . Please note that the options in the List are evaluated based on the Domain Process you have assigned to the Domain Manager part associated with your Window.

And go on in the same way you have done making a view for the .

3

Programming Reference

3.1. Introduction

Today’s Smalltalk applications are often designed according to the Model View Controller principle. The Application Framework supports the MVC principle by promoting the separation of responsibilities in the area of presentation (views), dialog control and models (Domain Objects and process logic).

In this chapter the basic techniques, classes and their protocols to be used regularly by application developers are described. This comprises some aspects of the approach when designing the application as well as the fundamental details of the framework's API. Ideally, the reader will be skilled enough after reading this chapter to easily build standard applications without using special features like dynamic Controls, late model access and Cached Views.

API information in this section

Base API methods are described in this section in detail. A subset of all base methods are listed in a table with the listed methods described in more detail following the table. The table format is typically similar to the following:

Type	Implementor	Method
Base method type	MicFwClass	baseApiMethod

Details about the methods listed in the table include the information from the API appendix in the following format:

`MicFwClass>>method`

3.2. Domain Object

3.2.1. Overview: Instance methods

Implementor	Message
MicFwDomainObject	openView: aViewClass

3.2.2. Method descriptions

`MicFwDomainObject>>openView:aViewClass`

3.3. Domain Process

3.3.1. Overview: Class methods

Type	Implementor	Message
Declaring	MicFwDomainProcess	modality

3.3.2. Overview: Instance methods

Type	Implementor	Message
Declaring	MicFwDomainProcess	modality
Initialising	MicFwDomainProcess	initializeInvolvedBaseConnections
Initialising	MicFwDomainProcess	initializeChildProcessConnections
Opening	MicFwDomainProcess	openView
Opening	MicFwDomainProcess	openView: aViewClass
Services	MicFwDomainProcess	startInteractionForAspect:anAspect
Services	MicFwDomainProcess	startInteractionForAspect: anAspect inConnectionNamed: connectionName
Transaction Behavior	MicFwDomainProcess	abortAndBegin
Transaction Behavior	MicFwDomainProcess	abortAndCloseProcess
Transaction Behavior	MicFwDomainProcess	abortAndCloseView
Transaction Behavior	MicFwDomainProcess	commitAndBegin
Transaction Behavior	MicFwDomainProcess	commitAndCloseProcess
Transaction Behavior	MicFwDomainProcess	commitAndCloseView
Transaction Behavior	MicFwDomainProcess	commitTransactionByDefault
Transaction Behavior	MicFwDomainProcess	startTransaction
Property	MicFwDomainProcess	mlsContext
Closing	MicFwDomainProcess	canClose: closeInfo
Closing	MicFwDomainProcess	closeProcess
Closing	MicFwDomainProcess	closeView
Closing	MicFwDomainProcess	viewClosed

3.3.3. Method descriptions

3.3.3.1. Declaring

First of all you have to implement your process classes as direct or indirect subclasses of MicFwDomainProcess. After this you can declare all base and child Process Connections via the Domain Processes Browser.

You can determine how Domain Processes of the receiver class will open their views by default. The method

MicFwDomainProcess>>modality

MicFwDomainProcess class>>modality

3.3.3.2. Initialising

MicFwDomainProcess>>initializeInvolvedBaseConnections

MicFwDomainProcess>>initializeChildProcessConnections

3.3.3.3. Opening

These methods open a view of the view class passed as argument or of the default view class (argument nil or no argument).

MicFwDomainProcess>>openView

MicFwDomainProcess>>openView: aViewClass

3.3.3.4. Services

MicFwDomainProcess>>startInteractionForAspect:anAspect

MicFwDomainProcess>>startInteractionForAspect: anAspect inConnection-Named: connectionName

3.3.3.5. Transaction Behavior

The behavior of the Domain Objects and processes explained above, can be extended by using the transactions offered by the Object Behavior and Persistence Framework. The declarations for the transaction behavior can be made using the Domain Processes Browser.

If flagged in the Domain Processes Browser, transactions will be started automatically during the opening of the Connector. This means if the Domain Process wants to use its own Transaction Context, then it will be created.

Afterwards the framework makes sure that each read and write access will be performed by the Transaction Context of the Domain Process. (s.a. [OBFw]).

MicFwDomainProcess>>abortAndBegin

MicFwDomainProcess>>abortAndCloseProcess

MicFwDomainProcess>>abortAndCloseView

MicFwDomainProcess>>commitAndBegin

MicFwDomainProcess>>commitAndCloseProcess

MicFwDomainProcess>>commitAndCloseView

MicFwDomainProcess>>commitTransactionByDefault

MicFwDomainProcess>>startTransaction

3.3.3.6. Property

MicFwDomainProcess>>mlsContext

3.3.3.7. Closing

In order to close a view and clean up all involved instances like views, Connections, viewports, Domain Objects and Domain Processes, the Model View Connector must be closed. During closing, it is concerned with closing and detaching all the connected objects

MicFwDomainProcess>>canClose: closeInfo

MicFwDomainProcess>>closeProcess

MicFwDomainProcess>>closeView

MicFwDomainProcess>>viewClosed

3.4. Viewport

3.4.1. Overview: Instance methods

Type	Implementor	Message
Model Navigation	MicFwViewPort	model
Model Navigation	MicFwViewPort	modelAtDefaultBaseConnection
Model Navigation	MicFwViewPort	modelAtDefaultProcessConnection
Controlling Subaspects	MicFwViewPort	<aspectName>BackgroundColor
Controlling Subaspects	MicFwViewPort	<aspectName>Checked
Controlling Subaspects	MicFwViewPort	<aspectName>Editable
Controlling Subaspects	MicFwViewPort	<aspectName>Enabled
Controlling Subaspects	MicFwViewPort	<aspectName>ForegroundColor
Controlling Subaspects	MicFwViewPort	<aspectName>HelpText
Controlling Subaspects	MicFwViewPort	<aspectName>HoverHelpText
Controlling Subaspects	MicFwViewPort	<aspectName>Label
Controlling Subaspects	MicFwViewPort	<aspectName>Readable
Controlling Subaspects	MicFwViewPort	<aspectName>Visible
Services	MicFwViewPort	isInterestedInChangesOf: anAspect
Services	MicFwViewPort	isInterestedInUpdatesOf: anAspect
Services	MicFwViewPort	localize: aLocale(MLS)

3.4.2. Method descriptions

3.4.2.1. Model Navigation

With the Viewport you can combine several processing steps which are independent of each other, and which even may be implemented in several Domain Processes.

Example

Pressing the OK button in an edit view results in storing the entered contract of insurance on one hand, and relating it to the selected policy holder as well on the other hand. At the same time the insurance data should be transmitted to the host and should be printed on an attached printer. There are special Domain Processes in the system that provide these functions.

The model navigation API

`MicFwViewPort>>model`

`MicFwViewPort>>modelAtDefaultBaseConnection`

`MicFwViewPort>>modelAtDefaultProcessConnection`

Example

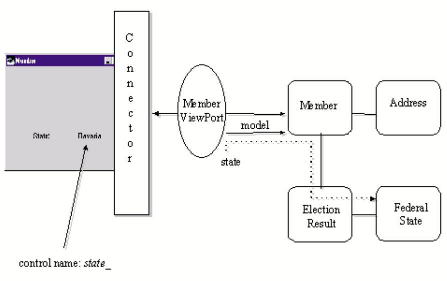
As an example, consider an election studio with members and election results. The Member Viewport enables access to a small object net shown below.

In the viewport, model answers the associated Domain Object. For example, if the state name is to be displayed in a text field named state_, then the method state in the member Viewport looks like this:

```
state
```

```
    ^self model electionResult state
```

REMARK: You should avoid to navigate to models of other Connectors, which is possible using the advanced API. Even this technical possibility exists, each process should have all the information in its own Connections, which are necessary to perform all its tasks.



3.4.2.2. Controlling Subaspects

The basic Subaspects of a Control can be set by using the following methods.

`MicFwViewPort>><aspectName>BackgroundColor`

`MicFwViewPort>><aspectName>Checked`

`MicFwViewPort>><aspectName>Editable`

`MicFwViewPort>><aspectName>Enabled`

`MicFwViewPort>><aspectName>ForegroundColor`

`MicFwViewPort>><aspectName>HelpText`

`MicFwViewPort>><aspectName>HoverHelpText`

`MicFwViewPort>><aspectName>Label`

`MicFwViewPort>><aspectName>Readable`

`MicFwViewPort>><aspectName>Visible`

3.4.2.3. Services

`MicFwViewPort>>isInterestedInChangesOf: anAspect`

`MicFwViewPort>>isInterestedInUpdatesOf: anAspect`

`MicFwViewPort>>localize: aLocale(MLS)`

3.5. View Parts

3.5.1. Introduction

3.5.1.1. Controlling parts in views

The views of an application are normally developed by means of graphical tools like Composition Editor, Window Builder or PARTS Workbench (tools from different providers). The result from this process is a set of views with a variety of Controls, stored in proprietary structures and formats. Aside from exceptional circumstances, the developer is strongly discouraged from using any of the (non-portable) features provided by a GUI tool to do anything else than the pure layout - like drawing event-action links between parts and the like. These things contradict the architecture induced by the Application Framework and make the growing application hard to maintain and extend, and they are widely covered in a more general fashion by the framework.

3.5.1.2. Controlling a view part using partName Naming Convention

A naming convention is introduced to make connecting the GUI elements with models as easy as possible:

During opening a view, a Control contained in the view is connected to a model if and only if the the Control's name ends with an underscore. In this case, the name of the Control defines the Connection(s) and selectors to be used for communication with connected Domain Objects and processes or their viewports.

3.5.1.2.1. Components of a part name

The exact structure of the partName for a component depends on the component type. The following possible components exist:

3.5.1.2.1.1. connectionName

The connection to a View Port and a model that implements the methods specified by the second part of the name ("accessorMethod").

3.5.1.2.1.2. accessorMethod

This part of the name has 2 functions:

- Accessor method for the content of the group control.The accessor method must return a MicFwMetaPartCollection, which is the dynamic contents of the group control.
- Specify the subaspects for the Group Control. A subaspect example would be the method "accessorMethodVisible", which can be used to control whether the GroupControl is visible or hidden.

The MetaParts in the MetaPartCollection will be displayed as Menu Items, Buttons, Forms, or Notebook pages depending on the MetaPart class. The MetaParts will be known to VA as children of the group control widget, allowing for example a Notebook to have dynamic pages.

3.5.1.2.1.3. baseConnectionName

For parts/widgets within the Group Control:

The base connection is taken for an inner widget , if the base is not specified in the name of the widget . This base connection name overrules the default mechanism of the framework which would be the insertion of #defaultBase.

For example, if a Group Control contains an entry field named "number_" (ie, with no base connection specified), the base connection will be specified by "baseConnectionName". This is the equivalent of an entry field with the name "baseConnectionName_number_".

This is useful for example when a Form should be used several times on a view, but the data for the Forms is accessed via different base connections. In this case the base connection name for the widgets in the Form is not specified, but is rather specified by the third part of the name of the Form.

3.5.1.2.1.4. baseDefaultProcessName

A connection specification of the process for the group control. The group control is connected to the connector of this specified process. Because of this, all child processes and base connections of the process are accessible to the group control and its inner widgets.

3.5.1.2.2. How to connect a static column

A static column connection has 2 parts.

The process for the column controls the column.

In this case the accessor would have the following format:

```
defaultProcess_column1_
```

To display an aspect with the name "#id"of the lists objects, a column1 method must return the aspect as the content of the column. For example:

```
column1
   ^#id
```
The label subaspect is used to set the label of the column:
```
column1Label
   ^'- ID -'
```
The subaspect #editable determines whethe or not the aspect is editable:
```
column1Editable
   ^ false
```
The editability must not not be set within the VA settings view to false!

3.5.1.2.3. How to connect a dynamic column

A container details view is connected as:

```
defaultProcess_list_defaultProcess_listSelection_asListEntry_
```
The columns are determined with the ~ItemAspects subaspect of the contents of the list (here >list<)-
```
listItemAspects
   currentItemAspects isNil ifTrue: [
      currentItemAspects := MicFwMetaPartCollection new
      add: (MicFwMetaAspect new
         accessorString: 'defaultProcess col1';
         title: 'Name';
         yourself);
      yourself)].
   ^currentItemAspects
```
This code will add a column and the aspect to be displayed is returned by the col1 method of the default process (as returned by the 'defaultProcess col1').

To display within the dynamic column the aspect #name of the object, the method col1 in the default process or in the default process viewport must be implemented as following:
```
col1
   ^ #name
```
A variation of this would be to use a MetaAspect that directly sets the content of the aspect and the width of the column. Example:
```
MicFwMetaPartCollection new
   add: (
      MicFwMetaAspect new
         aspect: #name;
         title: '-name-';
         columnWidth: 80;
         yourself);
   yourself
```

3.5.1.2.4. How to connect embedded parts

Embedded parts are currently only possible within container detail view.

3.5.1.2.4.1. An example of directing editting with embedded parts

A List with objects with viewport A should be displayed in a container details view. The first column is named 'defaultProcess_column1_'. The default process implements:
```
column1
   ^ #aspectXY
```
The connection for the direct editing is in viewport A.

A MicFwEmbeddedMetaPart will be returned with the aspect name of the column and the subaspect ~EmbeddedPart.

3.5.1.2.4.2. Examples:

ViewPort A implements:
```
column1EmbeddedPart
   ^  MicFwEmbeddedMetaPart new
         accessorString: 'defaultItem name'
         type: ##TEXT
         initiallySelected: true;
         yourself
```

```
column1EmbeddedPart
   ^  MicFwEmbeddedMetaPart new
        accessorString: 'defaultItem toggleButton'
        type: ##TOGGLEBUTTON;
        yourself

column1EmbeddedPart
^  MicFwEmbeddedMetaPart new
   accessorString: 'baseList theEntireList baseSelection theSelection firstName
VOID VOID'
        type: ##COMBOBOX;
        yourself

column1EmbeddedPart
   ^  MicFwEmbeddedMetaPart new
      accessorString: 'defaultItem entireList defaultItem theSelection VOID'
        type: ##RADIOBUTTONS;
        yourself

column1EmbeddedPart
   ^  MicFwEmbeddedMetaPart new
      accessorString: 'defaultItem entireList defaultItem theSelection VOID'
        type: ##DROPDOWNLIST;
        yourself

column1EmbeddedPart
   ^  MicFwEmbeddedMetaPart new
      accessorString: ' '
        type: ##NULL;
        yourself
```

As shown in the examples, there are six types of embedded parts:

- ##TEXT -> Entry field
- ##TOGGLEBUTTON -> Toggle button
- ##COMBOBOX -> Combo box
- ##RADIOBUTTONS -> Radio button set
- ##DROPDOWNLIST -> Drop down list
- ##NULL -> an non editable embedded part

The structure of the accessors is the same as for "normal" widgets. However, there is a reserved accessor name part #defaultItem, which has a specific meaning in the same manner as #defaultProcess.

The accessor name part #defaultItem corresponds to the item of the selected row, thus making it unnecessary to go back through the selection again.

For example, you can see in the first embedded part that the name of the selected item is requested (-> 'defaultItem name').

3.5.1.3. Controlling a view part using subaspect methods

Sub-aspects methods consisting of the connectionName of the part and and the name of a supported subAspect can be defined in the ViewPort for the View. These subaspects methods typically control such characteristics of a view control such as whether or not the part is enabled, visible, or the label of the part.

3.5.1.4. Overview of supported parts

3.5.1.4.1. Buttons

- ‘Push Button’ (page 106)
- ‘Toggle Button’ (page 107)
- ‘Radio Button Set’ (page 108)
- ‘Scale’ (page 109)
- ‘Slider’ (page 110)
- ‘Hot Spot’ (page 111)

3.5.1.4.2. Data Entry

- ‘Text’ (page 112)
- ‘Multi-line Edit’ (page 113)
- ‘Label’ (page 114)
- ‘Spin Button’ (page 115)
- ‘Mic Status Bar Part’ (page 117)
- ‘Mic Standard Status Bar Part’ (page 118)

3.5.1.4.3. Lists

- ‘List’ (page 119)
- ‘Multiple Select List’ (page 120)
- ‘Drop-down List’ (page 121)
- ‘Combo Box’ (page 122)
- ‘Container Icon Area’ (page 124)
- ‘Container Icon List’ (page 125)
- ‘Container Flowed Icon List’ (page 126)
- ‘Container Icon Tree’ (page 127)
- ‘Container Details’ (page 128)
- ‘Container Details Tree’ (page 129)
- ‘Packeting Container Details’ (page 130)
- ‘Container Details Column’ (page 131)

3.5.1.4.4. Menus

- ‘Popup Menu’ (page 132)
- ‘Menu Choice’ (page 133)
- ‘Menu Cascade’ (page 134)
- ‘Menu Toggle’ (page 135)
- ‘Mic Fw Toolbar View’ (page 136)
- ‘Mic Fw Framed Toolbar View’ (page 137)

3.5.1.4.5. Canvas

- ‘Window’ (page 138)
- ‘Form’ (page 139)
- ‘Group Box’ (page 140)
- ‘Scrolled Window’ (page 141)
- ‘PM Notebook’ (page 142)
- ‘Windows Notebook’ (page 143)
- ‘Notebook Page’ (page 144)

3.5.2. Push Button

3.5.2.1. partName

3.5.2.1.1. Syntax

```
[connectionName_]
commandMethod_
```

If connectionName_ omitted or declared as VOID_: The Default Process Connection will be used.

3.5.2.1.2. Supported variations

3.5.2.1.2.1. All parts of partName specified

```
connectionName_
commandMethod_
```

3.5.2.1.2.2. connectionName not specified

```
VOID_
commandMethod_
```

3.5.2.2. Supported Subaspects

- Enabled[Boolean]
- Label[String]
- Visible[Boolean]

3.5.3. Toggle Button

3.5.3.1. partName

3.5.3.1.1. Syntax

```
[connectionName_]
accessorName_
```

Content of the Control is aBoolean.

If connectionName omitted or declared as VOID_: The Default Base Connection will be used.

3.5.3.1.2. Supported variations

3.5.3.1.2.1. All parts of partName specified

```
connectionName_
accessorName_
```

3.5.3.1.2.2. connectionName not specified

```
VOID_
accessorName_
```

3.5.3.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Label[String]
- Visible[Boolean]

3.5.4. Radio Button Set

3.5.4.1. partName

3.5.4.1.1. Syntax

```
[connectionNameContents_]
accessorMethodContents_
[connectionNameSelection_]
accessorMethodSelection_
[displayMethodItem_]
```

Content of the Control is aCollection.

Selection of the Control is anObject.

If connectionName omitted or declared as VOID_: The Default Base Connection will be used.

The default presentation method is asListEntry.

3.5.4.1.2. Supported variations

3.5.4.1.2.1. All parts of partName specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
displayMethodItem_
```

3.5.4.1.2.2. displayMethod not specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
```

3.5.4.1.2.3. connectionNameContents / displayMethod not specified

```
VOID_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
```

3.5.4.1.2.4. connectionNameContents / connectionNameSelection / displayMethod not specified

```
VOID_
accessorMethodContents_
VOID_
accessorMethodSelection_
VOID_
```

3.5.4.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.5. Scale

3.5.5.1. partName

3.5.5.1.1. Syntax

```
[connectionNameContents_]
accessorMethodContents_
[connectionNameSelection_]
accessorMethodSelection_
```

If connectionName omitted or declared as VOID_: The Default Base Connection will be used.

Content of this Control is anInterval.

Selection of this Control is aNumber

3.5.5.1.2. Supported variations

3.5.5.1.2.1. All parts of partName specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
```

3.5.5.1.2.2. connectionNameContents not specified

```
VOID_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
```

3.5.5.1.2.3. connectionNameContents / connectionNameSelection not specified

```
VOID_
accessorMethodContents_
VOID_
accessorMethodSelection_
```

3.5.5.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.6. Slider

3.5.6.1. partName

3.5.6.1.1. Syntax

```
[connectionNameContents_]
accessorMethodContents_
[connectionNameSelection_]
accessorMethodSelection_
```

Content of this Control is anInterval.

Selection of this Control is aNumber

If connectionName omitted or declared as VOID_: The Default Base Connection will be used.

3.5.6.1.2. Supported variations

3.5.6.1.2.1. All parts of partName specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
```

3.5.6.1.2.2. connectionNameContents not specified

```
VOID_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
```

3.5.6.1.2.3. connectionNameContents / connectionNameSelection not specified

```
VOID_
accessorMethodContents_
VOID_
accessorMethodSelection_
```

3.5.6.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Editable[Boolean]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.7. Hot Spot

3.5.7.1. partName

3.5.7.1.1. Syntax

```
[connectionName_]
commandMethod_
```

If connectionName is omitted or declared as VOID_: The Default Process Connection will be used.

3.5.7.1.2. Supported variations

3.5.7.1.2.1. All parts of partName specified

```
connectionName_
commandMethod_
```

3.5.7.1.2.2. connectionName not specified

```
VOID_
commandMethod_
```

3.5.7.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.8. Text

3.5.8.1. partName

3.5.8.1.1. Syntax

```
[connectionName_]
accessorMethod_
```

The content of this Control is aString.

If connectionName is omitted or declared as VOID_: The Default Base Connection will be used.

3.5.8.1.2. Supported variations

3.5.8.1.2.1. All parts of partName specified

```
connectionName_
accessorMethod_
```

3.5.8.1.2.2. connectionName not specified

```
VOID_
accessorMethod_
```

3.5.8.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Editable[Boolean]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.9. Multi-line Edit

3.5.9.1. partName

3.5.9.1.1. Syntax

```
[connectionName_]
accessorMethod_
```

The content of this Control is aString.

If connectionName is omitted or declared as VOID_: The Default Base Connection will be used.

3.5.9.1.2. Supported variations

3.5.9.1.2.1. All parts of partName specified

```
connectionName_
accessorMethod_
```

3.5.9.1.2.2. connectionName not specified

```
VOID_
accessorMethod_
```

3.5.9.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Editable[Boolean]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.10. Label

3.5.10.1. partName

3.5.10.1.1. Syntax

```
[connectionName_]
accessorMethod_
```

The content of this Control is aString.

If connectionName is omitted or declared as VOID_: The Default Base Connection will be used.

3.5.10.1.2. Supported variations

3.5.10.1.2.1. All parts of partName specified

```
connectionName_
accessorMethod_
```

3.5.10.1.2.2. connectionName not specified

```
VOID_
accessorMethod_
```

3.5.10.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.11. Spin Button

3.5.11.1. partName

3.5.11.1.1. Syntax

```
[connectionNameContents_]
accessorMethodContents_
[connectionNameSelection_]
[accessorMethodSelection_]
[displayMethodItem_]
[connectionNameEntryField_]
[accessorMethodEntryField_]
```

The default presentation method is .

The content of this Control is aCollection.

The selection of this Control is anObject.

If connectionName is omitted or declared as VOID_: The Default Base Connection will be used.

3.5.11.1.2. Supported variations

3.5.11.1.2.1. All parts of partName specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
displayMethodItem_
connectionNameEntryField_
accessorMethodEntryField_
```

3.5.11.1.2.2. connectionNameSelection / accessorMethodSelection not specified

```
connectionNameContents_
accessorMethodContents_
VOID_
VOID_
displayMethodItem_
connectionNameEntryField_
accessorMethodEntryField_
```

3.5.11.1.2.3. connectionNameEntryField / accessorMethodEntryField not specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
displayMethodItem_
VOID_
VOID_
```

3.5.11.1.2.4. displayMethod / connectionNameEntryField / accessorMethodEntryField not specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
VOID_
VOID_
```

3.5.11.1.2.5. connectionNameSelection / displayMethod / connectionNameEntryField / accessorMethodEntryField not specified

```
connectionNameContents_
accessorMethodContents_
VOID_
accessorMethodSelection_
VOID_
VOID_
VOID_
```

3.5.11.1.2.6. connectionNameContents / connectionNameSelection / displayMethod / connectionNameEntryField / accessorMethodEntryField not specified

```
VOID_
accessorMethodContents_
VOID_
accessorMethodSelection_
VOID_
VOID_
VOID_
```

3.5.11.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Editable[Boolean]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.12. Mic Status Bar Part

3.5.12.1. partName

3.5.12.1.1. Syntax

```
[connectionName_]
accessorMethod_
```

The content of this Control is aString.

If connectionName is omitted or declared as VOID_: The Default Base Connection will be used.

3.5.12.1.2. Supported variations

3.5.12.1.2.1. All parts of partName specified

```
connectionName_
accessorMethod_
```

3.5.12.1.2.2. connectionName not specified

```
VOID_
accessorMethod_
```

3.5.12.2. Supported Subaspects

- Visible[Boolean]

3.5.13. Mic Standard Status Bar Part

3.5.13.1. partName

3.5.13.1.1. Syntax

```
[connectionName_]
accessorMethod_
```

If connectionName is omitted or declared as VOID_: The Default Base Connection will be used.

3.5.13.1.2. Supported variations

3.5.13.1.2.1. All parts of partName specified

```
connectionName_
accessorMethod_
```

3.5.13.1.2.2. connectionName not specified

```
VOID_
accessorMethod_
```

3.5.13.2. Supported Subaspects

- Visible[Boolean]

3.5.14. List

3.5.14.1. partName

3.5.14.1.1. Syntax

```
[connectionNameContents_]
accessorMethodContents_
[connectionNameSelection_]
[accessorMethodSelection_]
[displayMethodItem_]
```

The default presentation method is .

The content of this Control is aCollection.

The selection of this Control is anObject.

If connectionName is omitted or declared as VOID_: The Default Base Connection will be used.

3.5.14.1.2. Supported variations

3.5.14.1.2.1. All parts of partName specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
displayMethodItem_
```

3.5.14.1.2.2. displayMethod not specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
```

3.5.14.1.2.3. connectionNameContents / displayMethod not specified

```
VOID_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
```

3.5.14.1.2.4. connectionNameContents / connectionNameSelection / displayMethod not specified

```
VOID_
accessorMethodContents_
VOID_
accessorMethodSelection_
VOID_
```

3.5.14.1.2.5. connectionNameContents / connectionNameSelection / accessorMethodSelection / displayMethod not specified

```
VOID_
accessorMethodContents_
VOID_
VOID_
VOID_
```

3.5.14.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.15. Multiple Select List

3.5.15.1. partName

3.5.15.1.1. Syntax

```
[connectionNameContents_]
accessorMethodContents_
[connectionNameSelection_]
[accessorMethodSelection_]
[displayMethodItem_]
```

The default presentation method is .

The content of this Control is aCollection.

The selection of this Control is aCollection.

If connectionName is omitted or declared as VOID_: The Default Base Connection will be used.

3.5.15.1.2. Supported variations

3.5.15.1.2.1. All parts of partName specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
displayMethodItem_
```

3.5.15.1.2.2. displayMethod not specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
```

3.5.15.1.2.3. connectionNameContents / displayMethod not specified

```
VOID_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
```

3.5.15.1.2.4. connectionNameContents / connectionNameSelection / displayMethod not specified

```
VOID_
accessorMethodContents_
VOID_
accessorMethodSelection_
VOID_
```

3.5.15.1.2.5. connectionNameContents / connectionNameSelection / accessorMethodSelection / displayMethod not specified

```
VOID_
accessorMethodContents_
VOID_
VOID_
VOID_
```

3.5.15.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.16. Drop-down List

3.5.16.1. partName

3.5.16.1.1. Syntax

```
[connectionNameContents_]
accessorMethodContents_
[connectionNameSelection_]
accessorMethodSelection_
[displayMethodItem_]
```

The default presentation method is .

The content of this Control is aCollection.

The selection of this Control is anObject.

If connectionName is omitted or declared as VOID_: The Default Base Connection will be used.

3.5.16.1.2. Supported variations

3.5.16.1.2.1. All parts of partName specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
displayMethodItem_
```

3.5.16.1.2.2. displayMethod not specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
```

3.5.16.1.2.3. connectionNameSelection / displayMethod not specified

```
connectionNameContents_
accessorMethodContents_
VOID_
accessorMethodSelection_
VOID_
```

3.5.16.1.2.4. connectionNameContents / connectionNameSelection / displayMethod not specified

```
VOID_
accessorMethodContents_
VOID_
accessorMethodSelection_
VOID_
```

3.5.16.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.17. Combo Box

3.5.17.1. partName

3.5.17.1.1. Syntax

```
[connectionNameContents_]
accessorMethodContents_
[connectionNameSelection_]
[accessorMethodSelection_]
[displayMethodItem_]
[connectionNameEntryField_]
[accessorMethodEntryField_]
```

The default presentation method is .

The content of this Control is aCollection.

The selection of this Control is anObject.

If connectionName is omitted or declared as VOID_: The Default Base Connection will be used.

3.5.17.1.2. Supported variations

3.5.17.1.2.1. All parts of partName specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
displayMethodItem_
connectionNameEntryField_
accessorMethodEntryField_
```

3.5.17.1.2.2. connectionNameSelection / accessorMethodSelection not specified

```
connectionNameContents_
accessorMethodContents_
VOID_
VOID_
displayMethodItem_
connectionNameEntryField_
accessorMethodEntryField_
```

3.5.17.1.2.3. connectionNameEntryField / accessorMethodEntryField not specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
displayMethodItem_
VOID_
VOID_
```

3.5.17.1.2.4. displayMethodItem / connectionNameEntryField / accessorMethodEntryField not specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
VOID_
VOID_
```

3.5.17.1.2.5. connectionNameSelection / displayMethodItem / connectionNameEntryField / accessorMethodEntryField not specified

```
connectionNameContents_
accessorMethodContents_
VOID_
accessorMethodSelection_
VOID_
VOID_
VOID_
```

3.5.17.1.2.6. connectionNameContents / connectionNameSelection / displayMethodItem / connec-

tionNameEntryField / accessorMethodEntryField not specified

```
VOID_
accessorMethodContents_
VOID_
accessorMethodSelection_
VOID_
VOID_
VOID_
```

3.5.17.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Editable[Boolean]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.18. Container Icon Area

3.5.18.1. partName

3.5.18.1.1. Syntax

```
[connectionNameContents_]
accessorMethodContents_
[connectionNameSelection_]
[accessorMethodSelection_]
[displayMethodItem_]
```

The default presentation method is .

The content of this Control is aCollection.

The selection of this Control is aCollection.

If connectionName is omitted or declared as VOID_: The Default Base Connection will be used.

3.5.18.1.2. Supported variations

3.5.18.1.2.1. All parts of partName specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
displayMethodItem_
```

3.5.18.1.2.2. displayMethod not specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
```

3.5.18.1.2.3. connectionNameContents / displayMethod not specified

```
VOID_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
```

3.5.18.1.2.4. connectionNameContents / connectionNameSelection / displayMethod not specified

```
VOID_
accessorMethodContents_
VOID_
accessorMethodSelection_
VOID_
```

3.5.18.1.2.5. connectionNameContents / connectionNameSelection / accessorMethodSelection / displayMethod not specified

```
VOID_
accessorMethodContents_
VOID_
VOID_
VOID_
```

3.5.18.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Editable[Boolean]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.19. Container Icon List

3.5.19.1. partName

3.5.19.1.1. Syntax

```
[connectionNameContents_]
accessorMethodContents_
[connectionNameSelection_]
[accessorMethodSelection_]
[displayMethodItem_]
```

The default presentation method is .

The content of this Control is aCollection.

The selection of this Control is aCollection.

If connectionName is omitted or declared as VOID_: The Default Base Connection will be used.

3.5.19.1.2. Supported variations

3.5.19.1.2.1. All parts of partName specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
displayMethodItem_
```

3.5.19.1.2.2. displayMethod not specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
```

3.5.19.1.2.3. connectionNameContents / displayMethod not specified

```
VOID_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
```

3.5.19.1.2.4. connectionNameContents / connectionNameSelection / displayMethod not specified

```
VOID_
accessorMethodContents_
VOID_
accessorMethodSelection_
VOID_
```

3.5.19.1.2.5. connectionNameContents / connectionNameSelection / accessorMethodSelection / displayMethod not specified

```
VOID_
accessorMethodContents_
VOID_
VOID_
VOID_
```

3.5.19.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Editable[Boolean]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.20. Container Flowed Icon List

3.5.20.1. partName

3.5.20.1.1. Syntax

```
[connectionNameContents_]
accessorMethodContents_
[connectionNameSelection_]
[accessorMethodSelection_]
[displayMethodItem_]
```

The default presentation method is .

The content of this Control is aCollection.

The selection of this Control is aCollection.

If connectionName is omitted or declared as VOID_: The Default Base Connection will be used.

3.5.20.1.2. Supported variations

3.5.20.1.2.1. All parts of partName specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
displayMethodItem_
```

3.5.20.1.2.2. displayMethod not specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
```

3.5.20.1.2.3. connectionNameContents / displayMethod not specified

```
VOID_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
```

3.5.20.1.2.4. connectionNameContents / connectionNameSelection / displayMethod not specified

```
VOID_
accessorMethodContents_
VOID_
accessorMethodSelection_
VOID_
```

3.5.20.1.2.5. connectionNameContents / connectionNameSelection / accessorMethodSelection / displayMethod not specified

```
VOID_
accessorMethodContents_
VOID_
VOID_
VOID_
```

3.5.20.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Editable[Boolean]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.21. Container Icon Tree

3.5.21.1. partName

3.5.21.1.1. Syntax

```
[connectionNameContents_]
accessorMethodContents_
[connectionNameSelection_]
[accessorMethodSelection_]
[displayMethodItem_]
```

The default presentation method is .

The content of this Control is aCollection.

The selection of this Control is aCollection.

If connectionName is omitted or declared as VOID_: The Default Base Connection will be used.

3.5.21.1.2. Supported variations

3.5.21.1.2.1. All parts of partName specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
displayMethodItem_
```

3.5.21.1.2.2. displayMethod not specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
```

3.5.21.1.2.3. connectionNameContents / displayMethod not specified

```
VOID_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
```

3.5.21.1.2.4. connectionNameContents / connectionNameSelection / displayMethod not specified

```
VOID_
accessorMethodContents_
VOID_
accessorMethodSelection_
VOID_
```

3.5.21.1.2.5. connectionNameContents / connectionNameSelection / accessorMethodSelection / displayMethod not specified

```
VOID_
accessorMethodContents_
VOID_
VOID_
VOID_
```

3.5.21.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.22. Container Details

3.5.22.1. partName

3.5.22.1.1. Syntax

```
[connectionNameContents_]
accessorMethodContents_
[connectionNameSelection_]
[accessorMethodSelection_]
[displayMethodItem_]
```

The content of this Control is aCollection.

The selection of this Control is aCollection.

If connectionName is omitted or declared as VOID_: The Default Base Connection will be used.

3.5.22.1.2. Supported variations

3.5.22.1.2.1. All parts of partName specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
displayMethodItem_
```

3.5.22.1.2.2. displayMethod not specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
```

3.5.22.1.2.3. connectionNameContents / displayMethod not specified

```
VOID_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
```

3.5.22.1.2.4. connectionNameContents / connectionNameSelection / displayMethod not specified

```
VOID_
accessorMethodContents_
VOID_
accessorMethodSelection_
VOID_
```

3.5.22.1.2.5. connectionNameContents / connectionNameSelection / accessorMethodSelection / displayMethod not specified

```
VOID_
accessorMethodContents_
VOID_
VOID_
VOID_
```

3.5.22.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Editable[Boolean]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.23. Container Details Tree

3.5.23.1. partName

3.5.23.1.1. Syntax

```
[connectionNameContents_]
accessorMethodContents_
[connectionNameSelection_]
[accessorMethodSelection_]
[displayMethodItem_]
```

The default presentation method is .

The content of this Control is aCollection.

The selection of this Control is aCollection.

If the Connection Name is omitted or declared as VOID_, the Default Base Connection will be used.

3.5.23.1.2. Supported Variations

3.5.23.1.2.1. All parts of partName specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
displayMethodItem_
```

3.5.23.1.2.2. displayMethod not specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
```

3.5.23.1.2.3. connectionNameContents / displayMethod not specified

```
VOID_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
```

3.5.23.1.2.4. connectionNameContents / connectionNameSelection / displayMethod not specified

```
VOID_
accessorMethodContents_
VOID_
accessorMethodSelection_
VOID_
```

3.5.23.1.2.5. connectionNameContents / connectionNameSelection / accessorMethodSelection / displayMethod not specified

```
VOID_
accessorMethodContents_
VOID_
VOID_
VOID_
```

3.5.23.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Editable[Boolean]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.24. Packeting Container Details

3.5.24.1. partName

3.5.24.1.1. Syntax

```
[connectionNameContents_]
accessorMethodContents_
[connectionNameSelection_]
[accessorMethodSelection_]
[displayMethodItem_]
```

The content of this Control is aCollection.

The selection of this Control is aCollection.

If the Connection Name is omitted or declared as VOID_, the Default Base Connection will be used.

3.5.24.1.2. Supported Variations

3.5.24.1.2.1. All parts of partName specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
displayMethodItem_
```

3.5.24.1.2.2. displayMethod not specified

```
connectionNameContents_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
```

3.5.24.1.2.3. connectionNameContents / displayMethod not specified

```
VOID_
accessorMethodContents_
connectionNameSelection_
accessorMethodSelection_
VOID_
```

3.5.24.1.2.4. connectionNameContents / connectionNameSelection / displayMethod not specified

```
VOID_
accessorMethodContents_
VOID_
accessorMethodSelection_
VOID_
```

3.5.24.1.2.5. connectionNameContents / connectionNameSelection / accessorMethodSelection / displayMethod not specified

```
VOID_
accessorMethodContents_
VOID_
VOID_
VOID_
```

3.5.24.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Editable[Boolean]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.25. Container Details Column

3.5.25.1. partName

3.5.25.1.1. Syntax

```
[connectionName_]
accessorMethod_
```

The content of this Control is aSymbol which is the attributeName of the column.

If the Connection Name is omitted or declared as VOID_, the Default Base Connection will be used.

3.5.25.1.2. Supported Variations

3.5.25.1.2.1. All parts of partName specified

```
connectionName_
accessorMethod_
```

3.5.25.1.2.2. connectionName not specified

```
VOID_
accessorMethod
```

3.5.25.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Editable[Boolean]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Label[String]
- Visible[Boolean]

3.5.26. Popup Menu

3.5.26.1. partName

3.5.26.1.1. Syntax

```
[connectionName_]
commandMethod_
```

If the Connection Name is omitted or declared as VOID_, the Default Base Connection will be used.

3.5.26.1.2. Supported Variations

3.5.26.1.2.1. All parts of partName specified

```
connectionName_
commandMethod_
```

3.5.26.1.2.2. connectionName not specified

```
VOID_
commandMethod_
```

3.5.26.2. Supported Subaspects

- Accelerator[CwAccelerator]
- AcceleratorText[String]
- Enabled[Boolean]
- Label[String]

3.5.27. Menu Choice

3.5.27.1. partName

3.5.27.1.1. Syntax

```
[connectionName_]
commandMethod_
```

If connectionName omitted or declared as VOID_: The Default Process Connection will be used.

3.5.27.1.2. Supported Variations

3.5.27.1.2.1. All parts of partName specified

```
connectionName_
commandMethod_
```

3.5.27.1.2.2. connectionName not specified

```
VOID_
commandMethod_
```

3.5.27.2. Supported Subaspects

- Accelerator[CwAccelerator]
- AcceleratorText[String]
- Checked[Boolean]
- Enabled[Boolean]
- Label[String]

3.5.28. Menu Cascade

3.5.28.1. partName

3.5.28.1.1. Syntax

```
[connectionName_]
commandMethod_
```

If connectionName omitted or declared as VOID_: The Default Process Connection will be used.

3.5.28.1.2. Supported Variations

3.5.28.1.2.1. All parts of partName specified

```
connectionName_
commandMethod_
```

3.5.28.1.2.2. connectionName not specified

```
VOID_
commandMethod_
```

3.5.28.2. Supported Subaspects

- enabled[Boolean]
- Label[String]

3.5.29. Menu Toggle

3.5.29.1. partName

3.5.29.1.1. Syntax

```
[connectionName_]
accessorName_
```

Content of the Control is aBoolean.

If the Connection Name is omitted or declared as VOID_, the Default Base Connection will be used.

3.5.29.1.2. Supported Variations

3.5.29.1.2.1. All parts of partName specified

```
connectionName_
accessorName_
```

3.5.29.1.2.2. connectionName not specified

```
VOID_
accessorName_
```

3.5.29.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Label[String]
- Visible[Boolean]

3.5.30. Mic Fw Toolbar View

3.5.30.1. partName

3.5.30.1.1. Syntax

```
[connectionName_]
accessorMethod_
[baseConnectionName_]
[baseDefaultProcessName_]
```

Content of the Control is aMicFwMetaPartCollection.

If connectionName omitted or declared as VOID_: The Default Process Connection will be used.

3.5.30.1.2. Supported Variations

3.5.30.1.2.1. All parts of partName specified

```
connectionName_
accessorMethod_
baseConnectionName_
baseDefaultProcessName_
```

3.5.30.1.2.2. baseConnectionName not specified

```
connectionName_
accessorMethod_
VOID_
baseDefaultProcessName_
```

3.5.30.1.2.3. baseConnectionName / baseDefaultProcess not specified

```
connectionName_
accessorMethod_
VOID_
VOID_
```

3.5.30.1.2.4. connectionName / baseConnectionName / baseDefaultProcess not specified

```
VOID_
accessorMethod_
VOID_
VOID_
```

3.5.30.1.2.5. connectionName / baseConnectionName not specified

```
VOID_
accessorMethod_
VOID_
baseDefaultProcessName
```

3.5.30.1.2.6. connectionName / accessorMethod / baseConnectionName not specified

```
VOID_
VOID_
VOID_
baseDefaultProcessName
```

3.5.30.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.31. Mic Fw Framed Toolbar View

3.5.31.1. partName

3.5.31.1.1. Syntax

```
[connectionName_]
accessorMethod_
[baseConnectionName_]
[baseDefaultProcessName_]
```

Content of the Control is aMicFwMetaPartCollection.

If connectionName omitted or declared as VOID_: The Default Process Connection will be used.

3.5.31.1.2. Supported Variations

3.5.31.1.2.1. All parts of partName specified

```
connectionName_
accessorMethod_
baseConnectionName_
baseDefaultProcessName_
```

3.5.31.1.2.2. baseConnectionName not specified

```
connectionName_
accessorMethod_
VOID_
baseDefaultProcessName_
```

3.5.31.1.2.3. baseConnectionName / baseDefaultProcessName not specified

```
connectionName_
accessorMethod_
VOID_
VOID_
```

3.5.31.1.2.4. connectionName / baseConnectionName / baseDefaultProcessName not specified

```
VOID_
accessorMethod_
VOID_
VOID_
```

3.5.31.1.2.5. connectionName / baseConnectionName not specified

```
VOID_
accessorMethod_
VOID_
baseDefaultProcessName
```

3.5.31.1.2.6. connectionName / accessorMethod / baseConnectionName not specified

```
VOID_
VOID_
VOID_
baseDefaultProcessName
```

3.5.31.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.32. Window

3.5.32.1. partName

3.5.32.1.1. Syntax

```
[connectionName_]
[accessorMethod_]
[baseConnectionName_]
[baseDefaultProcessName_]
```

Content of the Control is aMicFwMetaPartCollection.

If connectionName omitted or declared as VOID_: The Default Process Connection will be used.

3.5.32.1.2. Supported Variations

3.5.32.1.2.1. All parts of partName specified

```
connectionName_
accessorMethod_
baseConnectionName_
baseDefaultProcessName_
```

3.5.32.1.2.2. baseConnectionName not specified

```
connectionName_
accessorMethod_
VOID_
baseDefaultProcessName_
```

3.5.32.1.2.3. baseConnectionName / baseDefaultProcessName not specified

```
connectionName_
accessorMethod_
VOID_
VOID_
```

3.5.32.1.2.4. connectionName / baseConnectionName / baseDefaultProcessName not specified

```
VOID_
accessorMethod_
VOID_
VOID_
```

3.5.32.1.2.5. connectionName / baseConnectionName not specified

```
VOID_
accessorMethod_
VOID_
baseDefaultProcessName
```

3.5.32.1.2.6. connectionName / accessorMethod / baseConnectionName not specified

```
VOID_
VOID_
VOID_
baseDefaultProcessName
```

3.5.32.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Label[String]
- Visible[Boolean]

3.5.33. Form

3.5.33.1. partName

3.5.33.1.1. Syntax

```
[connectionName_]
[accessorMethod_]
[baseConnectionName_]
[baseDefaultProcessName_]
```

Content of the Control is aMicFwMetaPartCollection.

If connectionName omitted or declared as VOID_: The Default Process Connection will be used.

3.5.33.1.2. Supported Variations

3.5.33.1.2.1. All parts of partName specified

```
connectionName_
accessorMethod_
baseConnectionName_
baseDefaultProcessName_
```

3.5.33.1.2.2. baseConnectionName not specified

```
connectionName_
accessorMethod_
VOID_
baseDefaultProcessName_
```

3.5.33.1.2.3. baseConnectionName / baseDefaultProcessName not specified

```
connectionName_
accessorMethod_
VOID_
VOID_
```

3.5.33.1.2.4. connectionName / baseConnectionName / baseDefaultProcessName not specified

```
VOID_
accessorMethod_
VOID_
VOID_
```

3.5.33.1.2.5. connectionName / baseConnectionName not specified

```
VOID_
accessorMethod_
VOID_
baseDefaultProcessName
```

3.5.33.1.2.6. connectionName / accessorMethod / baseConnectionName not specified

```
VOID_
VOID_
VOID_
baseDefaultProcessName
```

3.5.33.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.34. Group Box

3.5.34.1. partName

3.5.34.1.1. Syntax

```
[connectionName_]
[accessorMethod_]
[baseConnectionName_]
[baseDefaultProcessName_]
```

Content of the Control is aMicFwMetaPartCollection.

If connectionName omitted or declared as VOID_: The Default Process Connection will be used.

3.5.34.1.2. Supported Variations

3.5.34.1.2.1. All parts of partName specified

```
connectionName_
accessorMethod_
baseConnectionName_
baseDefaultProcessName_
```

3.5.34.1.2.2. baseConnectionName not specified

```
connectionName_
accessorMethod_
VOID_
baseDefaultProcessName_
```

3.5.34.1.2.3. baseConnectionName / baseDefaultProcessName not specified

```
connectionName_
accessorMethod_
VOID_
VOID_
```

3.5.34.1.2.4. connectionName / baseConnectionName / baseDefaultProcessName not specified

```
VOID_
accessorMethod_
VOID_
VOID_
```

3.5.34.1.2.5. connectionName / baseConnectionName not specified

```
VOID_
accessorMethod_
VOID_
baseDefaultProcessName
```

3.5.34.1.2.6. connectionName / accessorMethod / baseConnectionName not specified

```
VOID_
VOID_
VOID_
baseDefaultProcessName
```

3.5.34.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Label[String]
- Visible[Boolean]

3.5.35. Scrolled Window

3.5.35.1. partName

3.5.35.1.1. Syntax

```
[connectionName_]
[accessorMethod_]
[baseConnectionName_]
[baseDefaultProcessName_]
```

Content of the Control is aMicFwMetaPartCollection.

If connectionName omitted or declared as VOID_: The Default Process Connection will be used.

3.5.35.1.2. Supported Variations

3.5.35.1.2.1. All parts of partName specified

```
connectionName_
accessorMethod_
baseConnectionName_
baseDefaultProcessName_
```

3.5.35.1.2.2. baseConnectionName not specified

```
connectionName_
accessorMethod_
VOID_
baseDefaultProcessName_
```

3.5.35.1.2.3. baseConnectionName / baseDefaultProcessName not specified

```
connectionName_
accessorMethod_
VOID_
VOID_
```

3.5.35.1.2.4. connectionName / baseConnectionName / baseDefaultProcessName not specified

```
VOID_
accessorMethod_
VOID_
VOID_
```

3.5.35.1.2.5. connectionName / baseConnectionName not specified

```
VOID_
accessorMethod_
VOID_
baseDefaultProcessName
```

3.5.35.1.2.6. connectionName / accessorMethod / baseConnectionName not specified

```
VOID_
VOID_
VOID_
baseDefaultProcessName
```

3.5.35.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.36. PM Notebook

3.5.36.1. partName

3.5.36.1.1. Syntax

```
[connectionName_]
[accessorMethod_]
[baseConnectionName_]
[baseDefaultProcessName_]
```

Content of the Control is aMicFwMetaPartCollection.

If connectionName omitted or declared as VOID_: The Default Process Connection will be used.

3.5.36.1.2. Supported Variations

3.5.36.1.2.1. All parts of partName specified

```
connectionName_
accessorMethod_
baseConnectionName_
baseDefaultProcessName_
```

3.5.36.1.2.2. baseConnectionName not specified

```
connectionName_
accessorMethod_
VOID_
baseDefaultProcessName_
```

3.5.36.1.2.3. baseConnectionName / baseDefaultProcessName not specified

```
connectionName_
accessorMethod_
VOID_
VOID_
```

3.5.36.1.2.4. connectionName / baseConnectionName / baseDefaultProcessName not specified

```
VOID_
accessorMethod_
VOID_
VOID_
```

3.5.36.1.2.5. connectionName / baseConnectionName not specified

```
VOID_
accessorMethod_
VOID_
baseDefaultProcessName
```

3.5.36.1.2.6. connectionName / accessorMethod / baseConnectionName not specified

```
VOID_
VOID_
VOID_
baseDefaultProcessName
```

3.5.36.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.37. Windows Notebook

3.5.37.1. partName

3.5.37.1.1. Syntax

```
[connectionName_]
[accessorMethod_]
[baseConnectionName_]
[baseDefaultProcessName_]
```

Content of the Control is aMicFwMetaPartCollection.

If connectionName omitted or declared as VOID_: The Default Process Connection will be used.

3.5.37.1.2. Supported Variations

3.5.37.1.2.1. All parts of partName specified

```
connectionName_
accessorMethod_
baseConnectionName_
baseDefaultProcessName_
```

3.5.37.1.2.2. baseConnectionName not specified

```
connectionName_
accessorMethod_
VOID_
baseDefaultProcessName_
```

3.5.37.1.2.3. baseConnectionName / baseDefaultProcessName not specified

```
connectionName_
accessorMethod_
VOID_
VOID_
```

3.5.37.1.2.4. connectionName / baseConnectionName / baseDefaultProcessName not specified

```
VOID_
accessorMethod_
VOID_
VOID_
```

3.5.37.1.2.5. connectionName / baseConnectionName not specified

```
VOID_
accessorMethod_
VOID_
baseDefaultProcessName
```

3.5.37.1.2.6. connectionName / accessorMethod / baseConnectionName not specified

```
VOID_
VOID_
VOID_
baseDefaultProcessName
```

3.5.37.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Visible[Boolean]

3.5.38. Notebook Page

3.5.38.1. partName

3.5.38.1.1. Syntax

```
[connectionName_]
[accessorMethod_]
[baseConnectionName_]
[baseDefaultProcessName_]
```

Content of the Control is aMicFwMetaPartCollection.

If connectionName omitted or declared as VOID_: The Default Process Connection will be used.

3.5.38.1.2. Supported Variations

3.5.38.1.2.1. All parts of partName specified

```
connectionName_
accessorMethod_
baseConnectionName_
baseDefaultProcessName_
```

3.5.38.1.2.2. baseConnectionName not specified

```
connectionName_
accessorMethod_
VOID_
baseDefaultProcessName_
```

3.5.38.1.2.3. baseConnectionName / baseDefaultProcessName not specified

```
connectionName_
accessorMethod_
VOID_
VOID_
```

3.5.38.1.2.4. connectionName / baseConnectionName / baseDefaultProcessName not specified

```
VOID_
accessorMethod_
VOID_
VOID_
```

3.5.38.1.2.5. connectionName / baseConnectionName not specified

```
VOID_
accessorMethod_
VOID_
baseDefaultProcessName
```

3.5.38.1.2.6. connectionName / accessorMethod / baseConnectionName not specified

```
VOID_
VOID_
VOID_
baseDefaultProcessName
```

3.5.38.2. Supported Subaspects

- BackgroundColor[CgRGBColor]
- Enabled[Boolean]
- ForegroundColor[CgRGBColor]
- Label (PM Notebook)[String]
- TabLabel[String]
- TabType (PM Notebook)[Symbol(#major / #minor)]
- Visible[Boolean]
- LabelBackgroundColor (only OS/2)[CgRGBColor]

4

Advanced Programming Reference

4.1. Introduction

In this chapter the advanced techniques, classes and their protocols to be used regularly by application developers are described. This comprises more aspects of the approach when designing the application as well as additional details of the framework's API. Ideally, the reader will be skilled enough after reading this chapter to deal with advanced techniques like dynamic Processes, late model access and Cached Views.

API information in this section

Advanced API methods are described in this section in detail. A subset of methods are listed in a table with the listed methods described in more detail following the table. The table format is typically similar to the following:

Type	Implementor	Method
Advanced API method type (Declaring, Initializing, Services, etc.)	**MicFwClass**	**advancedApiMethod**
Base method type (described in the non-advanced programming reference earlier in this document)	MicFwClass	baseApiMethod

Details about the methods listed in the table include the information from the API appendix in the following format:

`MicFwClass>>method`

4.2. Domain Object

4.2.1. Overview: Class methods

Implementor	Message
MicFwModelObject	permitWriteDuringReadWhile: aBlock
MicFwModelObject	permitWriteDuringReadWhile: aBlock

Due to the fact that read access to Framework model is traced by the Viewports for the automated update mechanism, certain restrictions have to be observed. So there should be no write access to model aspects during a traced read, because write access may cause significant performance loss due to excessive update events or even confuse the update mechanism.

Therefore write access during a traced read will raise a resumable Exception named

.

However, there are some cases when write access during a traced read may be necessary or desirable form the view of the application, especially when a lazy initialization is performed. Therefore write during a traced read can be temporarily permitted with the methods

4.2.2. Method descriptions

In this sense, a method performing lazy initialization may look like:

```
MyModel >> lazyAspect
self basicLazyAspect isNil ifTrue: [
   self permitWriteDuringReadWhile: [
      self basicLazyAspect: self initialValueForLazyAspect
   ]
].
^self basicLazyAspect
```

4.3. Domain Process

4.3.1. Overview: Class methods

Type	Implementor	Method
Declaring	**MicFwDomainProcess**	**defaultHierarchicalPersistent-TransactionPolicyClass**
Declaring	**MicFwDomainProcess**	**defaultHierarchicalTransac-tionPolicyClass**
Declaring	**MicFwDomainProcess**	**defaultInheritedTransactionPolicyClass**
Declaring	**MicFwDomainProcess**	**defaultPersistentTransactionPolicyClass**
Declaring	**MicFwDomainProcess**	**defaultProcessContextFor: anInstance asChildOf: aParentProcess named: aCon-Name**
Declaring	**MicFwDomainProcess**	**defaultTransactionPolicyClass**
Declaring	**MicFwDomainProcess**	**defaultTransactionPolicyFor: aChildPro-cess asChildOf: aParentProcess named: conNameSymbol**
Declaring	**MicFwModelObject**	**defaultValidationRule**
Declaring	**MicFwDomainProcess**	**defaultWindowingPolicyClass**
Declaring	**MicFwDomainProcess**	**defaultWindowingPolicyFor: aProcessIn-stance asChildOf: aParentProcess named: conName viewClass: aView-ClassHint parentViewHolder: aDomain-Process modality: aModality**
Declaring	MicFwDomainProcess	modality
Declaring	**MicFwDomainProcess**	**processContextEvaluator**
Declaring	**MicFwModelObject**	**writeDuringReadIgnored**
Initializing	**MicFwModelObject**	**initializeAuthorizationTable**
Initializing	**MicFwModelObject**	**initializeValidationTable**
Starting	**MicFwDomainProcess**	**start**
Services	**MicFwModelObject**	**authorizeRead: aspect using: rule**
Services	**MicFwModelObject**	**authorizeRead: aspect using: rule ifPro-hibitedReturn: value**
Services	**MicFwModelObject**	**authorizeReadUsing: rule**
Services	**MicFwModelObject**	**authorizeWrite: aspect using: rule**
Services	**MicFwModelObject**	**authorizeWriteUsing: rule**
Services	**MicFwDomainProcess**	**authorizePerform: anAspect using: aRule**
Services	**MicFwDomainProcess**	**authorizePerformUsing: aRule**
Services	**MicFwDomainProcess**	**validatePerform: anAspect using: aRule**
Services	**MicFwModelObject**	**validateWrite: aspect using: rule**
Property	**MicFwDomainProcess**	**isAuthorizationActive**

Type	Implementor	Method
Property	MicFwDomainProcess	isExecAuthorizationActive
Property	MicFwDomainProcess	isExecValidationActive
Property	MicFwDomainProcess	isTechnicalDomainProcess
Property	MicFwDomainProcess	isValidationActive
Process Context	MicFwDomainProcess	processContextEvaluator
Process Context	MicFwDomainProcess	defaultProcessContextFor: anInstance asChildOf: aParentProcess named: conName
MicFwViewCachingWindowingPolicy	MicFwViewCachingWindowingPolicy	clearCache
Transaction Policy	MicFwDomainProcess	defaultTransactionPolicyFor: anInstance asChildOf: aParentProcess named: conName
Transaction Policy	MicFwDomainProcess	defaultTransactionPolicyClass
Trancaction Policy	MicFwDomainProcess	defaultPersistentTransactionPolicyClass
Trancaction Policy	MicFwDomainProcess	defaultHierarchicalPersistentTransactionPolicyClass
Trancaction Policy	MicFwDomainProcess	defaultHierarchicalTransactionPolicyClass
Trancaction Policy	MicFwDomainProcess	defaultInheritedTransactionPolicyClass
Advanced API Methods	MicFwDomainProcess	defaultTransactionPolicyFor: anInstance asChildOf: aParentProcess named: conName
Advanced API Methods	MicFwDomainProcess	defaultWindowingPolicyForChildOf: aParentProcess named: conName viewClass: aViewClassHint parentViewHolder: aDomainProcess modality: aModality
Advanced API Methods	MicFwDomainProcess	processContextEvaluator

4.3.2. Overview: Instance methods

Type	Implementor	Message
Declaring	MicFwDomainProcess	modality
Declaring	MicFwDomainProcess	permitsPerformAccessTo: aspect
Declaring	MicFwDomainProcess	stayOpenWithParent
Declaring	MicFwDomainProcess	validatesPerformAccessTo: anAspect
Creating	MicFwDomainProcess	addChildProcess: aDomainProcessClass
Creating	MicFwDomainProcess	addChildProcess: aDomainProcessClass named: aSymbol

Type	Implementor	Message
Creating	**MicFwDomainProcess**	**addChildProcess: aDomainProcessClass named: aSymbol withContext: aContext withSuperContext: aSuperContext**
Creating	**MicFwDomainProcess**	**addNewChildProcess: DomainProcess-Class**
Creating	**MicFwDomainProcess**	**addNewChildProcess: aDomainProcess-Class named: aSymbol**
Creating	**MicFwDomainProcess**	**addNewChildProcess: aDomainProcess-Class named: aSymbol withContext: aContext withSuperContext: aSuperCon-text**
Creating	**MicFwDomainProcess**	**created**
Creating	**MicFwDomainProcess**	**windowingPolicyClass**
Initialising	MicFwDomainProcess	initializeInvolvedBaseConnections
Initialising	MicFwDomainProcess	initializeChildProcessConnections
Initializing	**MicFwDomainProcess**	**startingChildProcess: aChildProcess**
Starting	**MicFwDomainProcess**	**childProcessesAreStarting**
Starting	**MicFwDomainProcess**	**connectorActivated**
Starting	**MicFwDomainProcess**	**prepareChildForInteraction: aChildPro-cess**
Starting	**MicFwDomainProcess**	**start**
Opening	MicFwDomainProcess	openView
Opening	MicFwDomainProcess	openView: aViewClass
Opening	**MicFwDomainProcess**	**openView: aViewClass withParentPro-cess: aDomainProcess**
Opening	**MicFwDomainProcess**	**openViewWithParentProcess: aDomain-Process**
Services	**MicFwDomainProcess**	**activateConnectorWithRefresh**
Services	**MicFwDomainProcess**	**addPresetChoiceConnectionNamed: aSymbol**
Services	**MicFwDomainProcess**	**addPresetHierarchyListConnec-tionNamed: aSymbol**
Services	**MicFwDomainProcess**	**addPresetListConnectionNamed: aSym-bol**
Services	**MicFwDomainProcess**	**canStartInteractionForAspect: anAspect**
Services	**MicFwDomainProcess**	**canStartInteractionForAspect: anAspect inConnectionNamed: connectionName**
Services	**MicFwDomainProcess**	**currentChoice**
Services	**MicFwDomainProcess**	**deferUpdateInAllViewsWhile: aBlock**
Services	**MicFwDomainProcess**	**deferViewUpdateWhile: aBlock**
Services	**MicFwDomainProcess**	**hierarchicalPersistentTransac-tionPolicyClass**

Type	Implementor	Message
Services	**MicFwDomainProcess**	**hierarchicalTransactionPolicyClass**
Services	**MicFwDomainProcess**	**inheritedTransactionPolicyClass**
Services	**MicFwDomainProcess**	**modelChosen: aConnection**
Services	**MicFwDomainProcess**	**modelItemChosen: aConnection**
Services	**MicFwDomainProcess**	**modelSelected: aConnection**
Services	**MicFwDomainProcess**	**modelViewConnector**
Services	**MicFwDomainProcess**	**persistentTransactionPolicyClass**
Services	MicFwDomainProcess	startInteractionForAspect:anAspect
Services	MicFwDomainProcess	startInteractionForAspect: anAspect inCon-nectionNamed: connectionName
Services	**MicFwDomainProcess**	**transactionPolicyClass**
Services	**MicFwDomainProcess**	**parent**
Services	**MicFwModelObject**	**permitWriteDuringReadWhile: aBlock**
Services	**MicFwDomainProcess**	**processContext**
Services	**MicFwDomainProcess**	**reactivateConnector**
Services	**MicFwDomainProcess**	**reactivateView**
Services	**MicFwDomainProcess**	**reactivateViewInteraction**
Services	**MicFwDomainProcess**	**readModelAtConnection: aConnection-Name**
Services	**MicFwDomainProcess**	**refreshAllViews**
Services	**MicFwDomainProcess**	**refreshView**
Services	**MicFwDomainProcess**	**selectAspect: anAspect**
Services	**MicFwDomainProcess**	**selectProcess**
Services	**MicFwDomainProcess**	**windowingPolicy**
Services	**MicFwDomainProcess**	**writeModelAtConnection: aSymbol value: aModel**
Transaction Behavior	MicFwDomainProcess	abortAndBegin
Transaction Behavior	**MicFwDomainProcess**	**abortAndBeginAlone**
Transaction Behavior	**MicFwDomainProcess**	**abortAndBeginTouched**
Transaction Behavior	**MicFwDomainProcess**	**abortAndClose**
Transaction Behavior	MicFwDomainProcess	abortAndCloseProcess
Transaction Behavior	MicFwDomainProcess	abortAndCloseView
Transaction Behavior	**MicFwDomainProcess**	**aboutToAbortTransaction**
Transaction Behavior	**MicFwDomainProcess**	**aboutToCloseTransaction: aTransaction-CloseMode**
Transaction Behavior	**MicFwDomainProcess**	**aboutToCommitTransaction**
Transaction Behavior	**MicFwDomainProcess**	**aboutToStartTransaction**

Type	Implementor	Message
Transaction Behavior	**MicFwDomainProcess**	**activateTransactionContextFor: aBlock**
Transaction Behavior	**MicFwDomainProcess**	**canAbortAndClose**
Transaction Behavior	**MicFwDomainProcess**	**canCommitAndClose**
Transaction Behavior	MicFwDomainProcess	commitAndBegin
Transaction Behavior	**MicFwDomainProcess**	**commitAndBeginAlone**
Transaction Behavior	**MicFwDomainProcess**	**commitAndBeginTouched**
Transaction Behavior	**MicFwDomainProcess**	**commitAndClose**
Transaction Behavior	MicFwDomainProcess	commitAndCloseProcess
Transaction Behavior	MicFwDomainProcess	commitAndCloseView
Transaction Behavior	MicFwDomainProcess	commitTransactionByDefault
Transaction Behavior	**MicFwDomainProcess**	**innerAbortTransaction**
Transaction Behavior	**MicFwDomainProcess**	**innerCommitTransaction**
Transaction Behavior	**MicFwDomainProcess**	**innerStartTransaction**
Transaction Behavior	**MicFwDomainProcess**	**performAbortAndClose**
Transaction Behavior	**MicFwDomainProcess**	**performClose**
Transaction Behavior	**MicFwDomainProcess**	**performCommitAndClose**
Transaction Behavior	**MicFwDomainProcess**	**persistenceObjectManager**
Transaction Behavior	**MicFwDomainProcess**	**prepareAbort**
Transaction Behavior	**MicFwDomainProcess**	**prepareCommit**
Transaction Behavior	**MicFwDomainProcess**	**resetTransactionClose**
Transaction Behavior	MicFwDomainProcess	startTransaction
Transaction Behavior	**MicFwDomainProcess**	**transactionAborted**
Transaction Behavior	**MicFwDomainProcess**	**transactionClosed: aTransactionClose-Mode**
Transaction Behavior	**MicFwDomainProcess**	**transactionCommitted**
Transaction Behavior	**MicFwDomainProcess**	**transactionPolicy**
Transaction Behavior	**MicFwDomainProcess**	**transactionStarted**
Property	**MicFwDomainProcess**	**accessesConnectionNamed: aConnec-tionName**
Property	**MicFwDomainProcess**	**childProcesses**
Property	**MicFwDomainProcess**	**hasConnectionNamed: aConnectionName**
Property	**MicFwDomainProcess**	**hasDynamicChildProcessNamed: aCon-nectionName**
Property	**MicFwDomainProcess**	**hasViewInteraction**
Property	**MicFwDomainProcess**	**isOpen**
Property	**MicFwDomainProcess**	**isViewOpen**
Property	MicFwDomainProcess	mlsContext

Type	Implementor	Message
Property	**MicFwDomainProcess**	**nameInParent**
Closing	**MicFwDomainProcess**	**canClose**
Closing	**MicFwDomainProcess**	**canClose: aCloseInfo**
Closing	MicFwDomainProcess	canClose: closeInfo
Closing	**MicFwDomainProcess**	**childProcessFinished: aChildProcess named: aConnectionName withResult: aProcessResult**
Closing	MicFwDomainProcess	closeProcess
Closing	**MicFwDomainProcess**	**closeRequested: aCloseInfo**
Closing	MicFwDomainProcess	closeView
Closing	**MicFwDomainProcess**	**connectorDeactivated**
Closing	MicFwDomainProcess	viewClosed
Closing	**MicFwDomainProcess**	**beNotifiedOfDocumentsClose**
Closing	**MicFwDomainProcess**	**defaultClose**
Closing	**MicFwDomainProcess**	**processClosed**
Closing	**MicFwDomainProcess**	**processResult**
Closing	**MicFwDomainProcess**	**removeChildProcess: aChildProcess**
Closing	**MicFwDomainProcess**	**removeChildProcessNamed: aConnec-tionName**
Closing	**MicFwDomainProcess**	**topLevelProcessClosed**
Closing	**MicFwDomainProcess**	**viewClosed**
Creating and Using of Dynamic Domain Pro-cesses	**MicFwDomainProcess**	**addNewChildProcess: aDomainProcess-Class**
Creating and Using of Dynamic Domain Pro-cesses	**MicFwDomainProcess**	**addNewChildProcess: aDomainProcess-Class named: aSymbol**
Creating and Using of Dynamic Domain Pro-cesses	**MicFwDomainProcess**	**addNewChildProcess: aDomainProcess-Class named: aSymbol withContext: aContext withSuperContext: aSuperCon-text**
Creating and Using of Dynamic Domain Pro-cesses	**MicFwDomainProcess**	**startingChildProcess: aChildProcess**
Creating and Using of Dynamic Domain Pro-cesses	**MicFwDomainProcess**	**removeChildProcess: aChildProcess**
Creating and Using of Dynamic Domain Pro-cesses	**MicFwDomainProcess**	**removeChildProcessNamed: aConnec-tionName**
Process Context	**MicFwDomainProcess**	**processContext**
Process Context	**MicFwAbstractDomain-ProcessContext**	**copyEnvFromParent: aDomainProcess-Context**

Type	Implementor	Message
Process Context	MicFwAbstractDomain-ProcessContext	createChildProcessContextFor: aDomain-Process
Process Context	MicFwAbstractDomain-ProcessContext	getEnv: var
Process Context	MicFwAbstractDomain-ProcessContext	processFinished
Process Context	MicFwAbstractDomain-ProcessContext	setDomainProcess: aDomainProcess
Process Context	MicFwAbstractDomain-ProcessContext	setEnv: var to: value
Process Context	MicFwAbstractDomain-ProcessContext	transactionPolicy
Process Context	MicFwAbstractDomain-ProcessContext	transactionPolicy: aTransactionPolicy
Process Context	MicFwAbstractDomain-ProcessContext	windowingPolicy
Process Context	MicFwAbstractDomain-ProcessContext	windowingPolicy: aWindowingPolicy
MicFwAbstractWindow-ingPolicy	MicFwAbstractWindow-ingPolicy	abstractView
MicFwAbstractWindow-ingPolicy	MicFwAbstractWindow-ingPolicy	abstractView: anAbstractView
MicFwAbstractWindow-ingPolicy	MicFwAbstractWindow-ingPolicy	interceptAbortClose
MicFwAbstractWindow-ingPolicy	MicFwAbstractWindow-ingPolicy	interceptClosePlatformView
MicFwAbstractWindow-ingPolicy	MicFwAbstractWindow-ingPolicy	interceptCommitClose
MicFwAbstractWindow-ingPolicy	MicFwAbstractWindow-ingPolicy	interceptOpenPlatformView
MicFwAbstractWindow-ingPolicy	MicFwAbstractWindow-ingPolicy	interceptPrepareClose
MicFwAbstractWindow-ingPolicy	MicFwAbstractWindow-ingPolicy	modality
MicFwAbstractWindow-ingPolicy	MicFwAbstractWindow-ingPolicy	modality: aModality
MicFwAbstractWindow-ingPolicy	MicFwAbstractWindow-ingPolicy	newPlatformViewFor: anAbstractView-Class
MicFwAbstractWindow-ingPolicy	MicFwAbstractWindow-ingPolicy	parentView
MicFwAbstractWindow-ingPolicy	MicFwAbstractWindow-ingPolicy	parentView: anAbstractView
MicFwAbstractWindow-ingPolicy	MicFwAbstractWindow-ingPolicy	viewClass

Type	Implementor	Message
MicFwAbstractWindow-ingPolicy	MicFwAbstractWindow-ingPolicy	viewClass: aClassHint
MicFwViewCachingWin-dowingPolicy	MicFwViewCachingWin-dowingPolicy	resetViewForCache
Transaction Policy	MicFwDomainProcess	transactionPolicy
Advanced API Methods	MicFwDomainProcess	addPresetChoiceConnectionNamed: aSymbol withContext: aContext withSu-perContext: aSuperContext
Advanced API Methods	MicFwDomainProcess	addPresetHierarchyListConnec-tionNamed: aSymbol withContext:aCon-text withSuperContext: aSuperContext
Advanced API Methods	MicFwDomainProcess	addPresetListConnectionNamed: aSym-bol withContext: aContext withSuperCon-text: aSuperContext

4.3.3. Method descriptions

4.3.3.1. Declaring

```
MicFwDomainProcess class>>defaultHierarchicalPersistent-
TransactionPolicyClass

MicFwDomainProcess class>>defaultHierarchicalTransactionPolicyClass

MicFwDomainProcess class>>defaultInheritedTransactionPolicyClass

MicFwDomainProcess class>>defaultPersistentTransactionPolicyClass

MicFwDomainProcess class>>defaultProcessContextFor: anInstance
asChildOf: aParentProcess named: aConName


MicFwDomainProcess class>>defaultTransactionPolicyClass

MicFwDomainProcess class>>defaultTransactionPolicyFor: aChildProcess
asChildOf: aParentProcess named: conNameSymbol


MicFwModelObject class>>defaultValidationRule

MicFwDomainProcess class>>defaultWindowingPolicyClass




MicFwDomainProcess class>>defaultWindowingPolicyFor: aProcessInstance
asChildOf: aParentProcess named: conName viewClass: aViewClassHint
parentViewHolder: aDomainProcess modality: aModality

MicFwDomainProcess class>>processContextEvaluator
```

MicFwModelObject class>>writeDuringReadIgnored

MicFwDomainProcess>>permitsPerformAccessTo: aspect

MicFwDomainProcess>>stayOpenWithParent

MicFwDomainProcess>>validatesPerformAccessTo: anAspect

4.3.3.2. Creating

MicFwDomainProcess>>addChildProcess: aDomainProcessClass
MicFwDomainProcess>>addChildProcess: aDomainProcessClass named: aSymbol
MicFwDomainProcess>>addChildProcess: aDomainProcessClass named: aSymbol withContext: aContext withSuperContext: aSuperContext

MicFwDomainProcess>>addNewChildProcess: DomainProcessClass
MicFwDomainProcess>>addNewChildProcess: aDomainProcessClass named: aSymbol
MicFwDomainProcess>>addNewChildProcess: aDomainProcessClass named: aSymbol withContext: aContext withSuperContext: aSuperContext

MicFwDomainProcess>>created

MicFwDomainProcess>>windowingPolicyClass

4.3.3.3. Initializing

MicFwModelObject class>>initializeAuthorizationTable

MicFwModelObject class>>initializeValidationTable

4.3.3.3.0.1. During the connector initialization

The base objects are initialized (<initializeInvolvedBaseConnections>).

The **static** child processes are created. The message <created> is sent to a child process immediately after the child process is created and the initialization of the child process connector.

All child processes receives <starting>.

After a dynamic child process is created:

During the connector initialization receives the child <initializeInvolvedBaseConnections> and <initialize-ChildProcessConnections>.

The parent process sends the <created> message to the child process

The child process in response sends the the <startingChildProcess: aChildProcess> to the parent.

The child process then receives <starting>

MicFwDomainProcess>>startingChildProcess: aChildProcess

4.3.3.4. Starting

MicFwDomainProcess class>>start

MicFwDomainProcess>>childProcessesAreStarting

`MicFwDomainProcess>>connectorActivated`

`MicFwDomainProcess>>prepareChildForInteraction: aChildProcess`

`MicFwDomainProcess>>start`

4.3.3.5. Opening

`MicFwDomainProcess>>openView: aViewClass withParentProcess: aDomain-Process`

`MicFwDomainProcess>>openViewWithParentProcess: aDomainProcess`

4.3.3.6. Services

`MicFwModelObject class>>authorizeRead: aspect using: rule`

`MicFwModelObject class>>authorizeRead: aspect using: rule ifProhibite-dReturn: value`

`MicFwModelObject class>>authorizeReadUsing: rule`

`MicFwModelObject class>>authorizeWrite: aspect using: rule`

`MicFwModelObject class>>authorizeWriteUsing: rule`

`MicFwDomainProcess class>>authorizePerform: anAspect using: aRule`

`MicFwDomainProcess class>>authorizePerformUsing: aRule`

`MicFwDomainProcess class>>validatePerform: anAspect using: aRule`

`MicFwModelObject class>>validateWrite: aspect using: rule`

`MicFwDomainProcess>>activateConnectorWithRefresh`

`MicFwDomainProcess>>addPresetChoiceConnectionNamed: aSymbol`

`MicFwDomainProcess>>addPresetHierarchyListConnectionNamed: aSymbol`

`MicFwDomainProcess>>addPresetListConnectionNamed: aSymbol`

`MicFwDomainProcess>>canStartInteractionForAspect: anAspect`

`MicFwDomainProcess>>canStartInteractionForAspect: anAspect inConnec-tionNamed: connectionName`

`MicFwDomainProcess>>currentChoice`

MicFwDomainProcess>>deferUpdateInAllViewsWhile: aBlock

MicFwDomainProcess>>deferViewUpdateWhile: aBlock

MicFwDomainProcess>>hierarchicalPersistentTransactionPolicyClass

MicFwDomainProcess>>hierarchicalTransactionPolicyClass

MicFwDomainProcess>>inheritedTransactionPolicyClass

MicFwDomainProcess>>modelChosen: aConnection

MicFwDomainProcess>>modelItemChosen: aConnection

MicFwDomainProcess>>modelSelected: aConnection

MicFwDomainProcess>>modelViewConnector

MicFwDomainProcess>>persistentTransactionPolicyClass

MicFwDomainProcess>>transactionPolicyClass

MicFwDomainProcess>>parent

MicFwModelObject>>permitWriteDuringReadWhile: aBlock

MicFwDomainProcess>>processContext

MicFwDomainProcess>>reactivateConnector

MicFwDomainProcess>>reactivateView

MicFwDomainProcess>>reactivateViewInteraction

MicFwDomainProcess>>readModelAtConnection: aConnectionName

MicFwDomainProcess>>refreshAllViews

MicFwDomainProcess>>refreshView

MicFwDomainProcess>>selectAspect: anAspect

MicFwDomainProcess>>selectProcess

MicFwDomainProcess>>windowingPolicy

MicFwDomainProcess>>writeModelAtConnection: aSymbol value: aModel

4.3.3.7. Transaction Behavior

For more information of hierarchical- and non hierarchical transaction contexts see the .
Also see the <canClose: aCloseInfo>, <closeRequested: aCloseInfo> methods, which can influence the

transaction behavior of processes.

`MicFwDomainProcess>>abortAndBeginAlone`

`MicFwDomainProcess>>abortAndBeginTouched`

`MicFwDomainProcess>>abortAndClose`

`MicFwDomainProcess>>aboutToAbortTransaction`

`MicFwDomainProcess>>aboutToCloseTransaction: aTransactionCloseMode`

`MicFwDomainProcess>>aboutToCommitTransaction`

`MicFwDomainProcess>>aboutToStartTransaction`

`MicFwDomainProcess>>activateTransactionContextFor: aBlock`

`MicFwDomainProcess>>canAbortAndClose`

`MicFwDomainProcess>>canCommitAndClose`

`MicFwDomainProcess>>commitAndBeginAlone`

`MicFwDomainProcess>>commitAndBeginTouched`

`MicFwDomainProcess>>commitAndClose`

`MicFwDomainProcess>>innerAbortTransaction`

`MicFwDomainProcess>>innerCommitTransaction`

`MicFwDomainProcess>>innerStartTransaction`

`MicFwDomainProcess>>performAbortAndClose`

`MicFwDomainProcess>>performClose`

`MicFwDomainProcess>>performCommitAndClose`

`MicFwDomainProcess>>persistenceObjectManager`

`MicFwDomainProcess>>prepareAbort`

`MicFwDomainProcess>>prepareCommit`

`MicFwDomainProcess>>resetTransactionClose`

`MicFwDomainProcess>>transactionAborted`

`MicFwDomainProcess>>transactionClosed: aTransactionCloseMode`

`MicFwDomainProcess>>transactionCommitted`

`MicFwDomainProcess>>transactionPolicy`

`MicFwDomainProcess>>transactionStarted`

4.3.3.8. Property

`MicFwDomainProcess class>>isAuthorizationActive`

`MicFwDomainProcess class>>isExecAuthorizationActive`

`MicFwDomainProcess class>>isExecValidationActive`

`MicFwDomainProcess class>>isTechnicalDomainProcess`

`MicFwDomainProcess class>>isValidationActive`

`MicFwDomainProcess>>accessesConnectionNamed: aConnectionName`

`MicFwDomainProcess>>childProcesses`

`MicFwDomainProcess>>hasConnectionNamed: aConnectionName`

`MicFwDomainProcess>>hasDynamicChildProcessNamed: aConnectionName`

`MicFwDomainProcess>>hasViewInteraction`

`MicFwDomainProcess>>isOpen`

`MicFwDomainProcess>>isViewOpen`

`MicFwDomainProcess>>nameInParent`

4.3.3.9. Closing

Other interesting methods are <stayOpenWithParent>.

About <aCloseInfo>

<aCloseInfo> is a where the <closeMode> is one of

- COMMIT -> commit the current transaction and close the view
- ABORT -> abort the current transaction and close the view
- DEFAULT -> generic close request (like system close button pressed)

and the <info> is set to the receiver, i.e. the Domain Process which has received the close command.

API methods

`MicFwDomainProcess>>canClose`

`MicFwDomainProcess>>canClose: aCloseInfo`

`MicFwDomainProcess>>childProcessFinished: aChildProcess named: aConnectionName withResult: aProcessResult`

`MicFwDomainProcess>>closeRequested: aCloseInfo`

`MicFwDomainProcess>>connectorDeactivated`

`MicFwDomainProcess>>beNotifiedOfDocumentsClose`

`MicFwDomainProcess>>defaultClose`

`MicFwDomainProcess>>processClosed`

`MicFwDomainProcess>>processResult`

`MicFwDomainProcess>>removeChildProcess: aChildProcess`

`MicFwDomainProcess>>removeChildProcessNamed: aConnectionName`

`MicFwDomainProcess>>topLevelProcessClosed`

`MicFwDomainProcess>>viewClosed`

4.3.3.10. Creating and Using of Dynamic Domain Processes

In some special cases you might use dynamic (child) processes. The fundamental difference to declared processes is, that you initiate the creation and closing of the process instance within your application. Therefore a dynamic Process may not always be represented in your process hierachy. Once started a dynamic and a static process react and behave in the same manner.

Hint: You should alway consider if you ony solve the problem by using a dynamic Process. Extreme use of dynamic Processes leads normaly to problems within your application design..

`MicFwDomainProcess>>addNewChildProcess: aDomainProcessClass`

`MicFwDomainProcess>>addNewChildProcess: aDomainProcessClass named: aSymbol`

`MicFwDomainProcess>>addNewChildProcess: aDomainProcessClass named: aSymbol withContext: aContext withSuperContext: aSuperContext`

`MicFwDomainProcess>>startingChildProcess: aChildProcess`

`MicFwDomainProcess>>removeChildProcess: aChildProcess`

`MicFwDomainProcess>>removeChildProcessNamed: aConnectionName`

4.3.3.11. Process Context

Each Domain Process is acciociated with one process context. These are instances of one of the subclasses of MicFwAbstractDomainProcessContext.

The context set the operational envrionment for the process instance. So he defines for example the transactional behavior and provides the nessesary view, if the process need one in the specific situation. The process context also hold / provide additional information for the Domain Process lifetime.

`MicFwDomainProcess>>processContext`

`MicFwDomainProcess class>>processContextEvaluator`

`MicFwDomainProcess class>>defaultProcessContextFor: anInstance asChildOf: aParentProcess named: conName`

`MicFwAbstractDomainProcessContext>>copyEnvFromParent: aDomainProcessContext`

`MicFwAbstractDomainProcessContext>>createChildProcessContextFor: aDomainProcess`

`MicFwAbstractDomainProcessContext>>getEnv: var`

`MicFwAbstractDomainProcessContext>>processFinished`

`MicFwAbstractDomainProcessContext>>setDomainProcess: aDomainProcess`

`MicFwAbstractDomainProcessContext>>setEnv: var to: value`

`MicFwAbstractDomainProcessContext>>transactionPolicy`

`MicFwAbstractDomainProcessContext>>transactionPolicy: aTransactionPolicy`

`MicFwAbstractDomainProcessContext>>windowingPolicy`

`MicFwAbstractDomainProcessContext>>windowingPolicy: aWindowingPolicy`

4.3.3.12. Windowing Policy

Each process context hold a so called windowing policy. This policy is responsible to create a proper view instance for the process. Windowing policies can be set for each process instance individually.

```
Object
| MicFwAbstractWindowingPolicy
| | MicFwStandardWindowingPolicy
| | | MicFwViewCachingWindowingPolicy
```

4.3.3.13. MicFwAbstractWindowingPolicy

This is the abstract windowing policy class. Each process may have one instance.

`MicFwAbstractWindowingPolicy>>abstractView`

`MicFwAbstractWindowingPolicy>>abstractView: anAbstractView`

`MicFwAbstractWindowingPolicy>>interceptAbortClose`

`MicFwAbstractWindowingPolicy>>interceptClosePlatformView`

`MicFwAbstractWindowingPolicy>>interceptCommitClose`

MicFwAbstractWindowingPolicy>>interceptOpenPlatformView

MicFwAbstractWindowingPolicy>>interceptPrepareClose

MicFwAbstractWindowingPolicy>>modality

MicFwAbstractWindowingPolicy>>modality: aModality

MicFwAbstractWindowingPolicy>>newPlatformViewFor: anAbstractViewClass

MicFwAbstractWindowingPolicy>>parentView

MicFwAbstractWindowingPolicy>>parentView: anAbstractView

MicFwAbstractWindowingPolicy>>viewClass

MicFwAbstractWindowingPolicy>>viewClass: aClassHint

4.3.3.14. MicFwStandardWindowingPolicy

Default policy for Domain Processes. Simply return a new VA view instance.

4.3.3.15. MicFwViewCachingWindowingPolicy

A simple view caching windowing policy. Provides some additional functionality to maintain previous Cached Views.

MicFwViewCachingWindowingPolicy class>>clearCache

MicFwViewCachingWindowingPolicy>>resetViewForCache

4.3.3.15.0.1. Related Classes / Methods

`MicFwDomainProcess class>> defaultWindowingPolicyClass`

4.3.3.16. Transaction Policy

The transaction policy controls and represents the transactional behavior of the Domain Process instance. Therefore exists various specialized classes as shown in the class hierachy below:

```
Object
| MicFwAbstractTransactionPolicy
| | MicFwNullTransactionPolicy
| | | MicFwStandardTransactionPolic
| | | | MicFwAbstractPersistentTransactionPolicy
| | | | | MicFwOldPersistentTransactionPolicy
| | | | | | MicFwHierarchicalOldPersistentTransactionPolicy
| | | | MicFwHierarchicalStandardTransactionPolicy
| | | | MicFwInheritedTransactionPolicy
| | | | MicFwTechnicalStandardTransactionPolicy
```

The   is set for processes with no transactional behavior. The other classes represent the supported transaction mode within the framework.

Transaction policies took over the complete control of the transactional behavior (i.e commit, abort, activate). Therefore a process could never direct perform these operations.

MicFwDomainProcess class>>defaultTransactionPolicyFor: anInstance asChildOf: aParentProcess named: conName
MicFwDomainProcess class>>defaultTransactionPolicyClass
MicFwDomainProcess class>>defaultPersistentTransactionPolicyClass
MicFwDomainProcess class>>defaultHierarchicalPersistent-TransactionPolicyClass
MicFwDomainProcess class>>defaultHierarchicalTransactionPolicyClass
MicFwDomainProcess class>>defaultInheritedTransactionPolicyClass

MicFwDomainProcess>>transactionPolicy

4.3.3.17. Advanced API Methods

4.3.3.17.1. Public Class Methods

```
MicFwDomainProcess class>>defaultTransactionPolicyFor: anInstance
asChildOf: aParentProcess named: conName

MicFwDomainProcess class>>defaultWindowingPolicyForChildOf: aParentPro-
cess named: conName viewClass: aViewClassHint parentViewHolder: aDo-
mainProcess modality: aModality

MicFwDomainProcess class>>processContextEvaluator
```

4.3.3.17.2. Public Instance Methods

```
MicFwDomainProcess>>addPresetChoiceConnectionNamed: aSymbol withCon-
text: aContext withSuperContext: aSuperContext
MicFwDomainProcess>>addPresetHierarchyListConnectionNamed: aSymbol
withContext:aContext withSuperContext: aSuperContext
MicFwDomainProcess>>addPresetListConnectionNamed: aSymbol withContext:
aContext withSuperContext: aSuperContext
```

4.4. Viewport

4.4.1. Overview: Class methods

Type	Implementor	Message
Declaring	**MicFwView-Port**	**isDefaultViewPort**

4.4.2. Overview: Instance methods

Type	Implementor	Message
Declaring	**MicFwViewPort**	**defaultEmbeddedPartValueOf: dispatch-erAspect**
Declaring	**MicFwViewPort**	**defaultMaxSearchLevelValueOf: dispatch-erAspect**
Initialize	**MicFwViewPort**	**initialize**
Model Navigation	MicFwViewPort	model
Model Navigation	**MicFwViewPort**	**modelAtConnection: aConnectionName**
Model Navigation	MicFwViewPort	modelAtDefaultBaseConnection
Model Navigation	MicFwViewPort	modelAtDefaultProcessConnection
Controlling Sub-aspects	**MicFwViewPort**	**<aSourceAspect><anOperation>: aListOfView-Ports onto: aViewPortOrNil**
Controlling Sub-aspects	**MicFwViewPort**	**<aSourceAspect><anOperation><aTar-getAspect>: aListOfViewPorts onto: aViewPor-tOrNil**
Controlling Sub-aspects	**MicFwViewPort**	**<aspectName>Accelerator**
Controlling Sub-aspects	**MicFwViewPort**	**<aspectName>AcceleratorText**
Controlling Sub-aspects	MicFwViewPort	<aspectName>BackgroundColor
Controlling Sub-aspects	MicFwViewPort	<aspectName>Checked
Controlling Sub-aspects	MicFwViewPort	<aspectName>Editable
Controlling Sub-aspects	MicFwViewPort	<aspectName>Enabled
Controlling Sub-aspects	MicFwViewPort	<aspectName>ForegroundColor
Controlling Sub-aspects	MicFwViewPort	<aspectName>HelpText
Controlling Sub-aspects	MicFwViewPort	<aspectName>HoverHelpText

Type	Implementor	Message
Controlling Sub-aspects	MicFwViewPort	<aspectName>Label
Controlling Sub-aspects	MicFwViewPort	<aspectName>Readable
Controlling Sub-aspects	MicFwViewPort	<aspectName>Visible
Controlling Sub-aspects	**MicFwViewPort**	**<anContentAspectOfAGroupControl>Child-PaneConnectionMode**
Controlling Sub-aspects	**MicFwViewPort**	**<columnAspectName>EmbeddedPart**
Controlling Sub-aspects	**MicFwViewPort**	**<detailsViewContentAspectName>ItemAspects**
Controlling Sub-aspects	**MicFwViewPort**	**<anAspect>Manipulator**
Controlling Sub-aspects	**MicFwViewPort**	**<anAspect>MetaControlListUpdate: aMetaCon-trolList**
Controlling Sub-aspects	**MicFwViewPort**	**defaultAcceptedOperationsValueOf: anAspect**
Controlling Sub-aspects	**MicFwViewPort**	**defaultCanCopy: aList valueOf: aSourceAspect**
Controlling Sub-aspects	**MicFwViewPort**	**defaultCanLink: aList valueOf: aSourceAspect**
Controlling Sub-aspects	**MicFwViewPort**	**defaultCanMove: aList valueOf: aSourceAspect**
Controlling Sub-aspects	**MicFwViewPort**	**<anAspect>Checked**
Controlling Sub-aspects	**MicFwViewPort**	**defaultCheckedValueOf: dispatcherAspect**
Controlling Sub-aspects	**MicFwViewPort**	**defaultCopy: aList onto: aViewportOrNil val-ueOf: anAspect**
Controlling Sub-aspects	**MicFwViewPort**	**<anAspect>Editable**
Controlling Sub-aspects	**MicFwViewPort**	**defaultEditableValueOf: aDispatcherAspect**
Controlling Sub-aspects	**MicFwViewPort**	**<anAspect>Enabled**
Controlling Sub-aspects	**MicFwViewPort**	**defaultEnabledValueOf: aDispatcherAspect**
Controlling Sub-aspects	**MicFwViewPort**	**defaultLateModelAccessValueOf: dispatch-erAspect**
Controlling Sub-aspects	**MicFwViewPort**	**defaultLink: aList onto: aViewportOrNil valueOf: anAspect**
Controlling Sub-aspects	**MicFwViewPort**	**<anAspect>Manipulator**

Type	Implementor	Message
Controlling Sub-aspects	MicFwViewPort	defaultManipulatorValueOf: anAspect
Controlling Sub-aspects	MicFwViewPort	defaultMove: aList onto: aViewportOrNil valueOf: anAspect
Controlling Sub-aspects	MicFwViewPort	defaultProhibitedValueOf: dispatcherAspect
Controlling Sub-aspects	MicFwViewPort	defaultProvidedOperationsValueOf: anAspect
Controlling Sub-aspects	MicFwViewPort	<anAspect>Readable
Controlling Sub-aspects	MicFwViewPort	defaultReadableValueOf: aDispatcherAspect
Controlling Sub-aspects	MicFwViewPort	<anAspect>MetaControlListUpdate: aMetaControlList
Controlling Sub-aspects	MicFwViewPort	defaultUpdateOfMetaControlList: aMetaControlList forAspect: anAspect
Controlling Sub-aspects	MicFwViewPort	<anAspect>Visible
Controlling Sub-aspects	MicFwViewPort	defaultVisibleValueOf: aDispatcherAspect
Controlling Sub-aspects	MicFwViewPort	graphicsPresentation
Controlling Sub-aspects	MicFwViewPort	readOnlyAspect: anAspect
Controlling Sub-aspects	MicFwViewPort	wantsCachedReadOf: aDispatcherAspect
Controlling Sub-aspects	MicFwViewPort	wantsImmediateViewUpdateOf: anAspect
Aspect Value Conversion	MicFwViewPort	getMetaControlListForAspect: anAspect
Aspect Value Conversion	MicFwViewPort	getMetaControlListForAspect: anAspect ifAbsent: absenceBlock
Aspect Value Conversion	MicFwViewPort	listHierarchyParentViewPort
Aspect Value Conversion	MicFwViewPort	newMetaControlListForAspect: anAspect
Aspect Value Conversion	MicFwViewPort	removeMetaControlListForAspect: anAspect
Services	MicFwViewPort	<anContentAspectOfAGroupControl>ChildPaneConnectionMode defaultChildPaneConnectionModeValueOf: anAspect
Services	MicFwViewPort	canStartInteractionForAspect: anAspect
Services	MicFwViewPort	childViewPortBaseClass
Services	MicFwViewPort	contextAtConnectionNamed: aSymbol

Type	Implementor	Message
Services	**MicFwViewPort**	**converter**
Services	**MicFwViewPort**	**deferUpdateInAllViewsWhile: aBlock**
Services	**MicFwViewPort**	**deferViewUpdateWhile: aBlock**
Services	**MicFwViewPort**	**enterInteractionForAspect: anAspect coming-FromAspect: lastAspect of: lastViewPort**
Services	**MicFwViewPort**	**getLocaleNamed: name ofType: type fromCon-textAtConnectionNamed: aSymbol**
Services	MicFwViewPort	isInterestedInChangesOf: anAspect
Services	MicFwViewPort	isInterestedInUpdatesOf: anAspect
Services	**MicFwViewPort**	**itemViewPortBaseClassForAspect: anAspect**
Services	**MicFwViewPort**	**leaveInteractionForAspect: anAspect going-ToAspect: nextAspect of: nextViewPort**
Services	MicFwViewPort	localize: aLocale(MLS)
Services	**MicFwViewPort**	**localizeAllConnectionsTo: aLocale**
Services	**MicFwViewPort**	**localizeConnectionNamed: aString to: aLocale**
Services	**MicFwViewPort**	**resolveKey: aMicMlsKey forLocale: aLocale**
Services	**MicFwViewPort**	**returnContextNamed: aContextClassSymbol**
Services	**MicFwViewPort**	**readAccessRejectedForAspect: anAspect vetoValue: aProhibitedModel**
Services	**MicFwViewPort**	**shouldEmulateEditable**
Services	**MicFwViewPort**	**startInteraction: aConnection**
Services	**MicFwViewPort**	**startInteractionForAspect: anAspect**
Services	**MicFwViewPort**	**stopInteraction: aConnection**
Services	**MicFwViewPort**	**writeAccessRejectedForAspect: anAspect vetoValue: aProhibitedModel**
Overriding Default Behavior	**MicFwViewPort**	**defaultEditableValueOf: aDispatcherAspect**
Overriding Default Behavior	**MicFwViewPort**	**defaultEnabledValueOf: aDispatcherAspect**
Overriding Default Behavior	**MicFwViewPort**	**defaultLateModelAccessValueOf: aDispatch-erAspect**
Overriding Default Behavior	**MicFwViewPort**	**defaultLink: aList onto: aViewportOrNil valueOf: anAspect**
Overriding Default Behavior	**MicFwViewPort**	**defaultManipulatorValueOf: aDispatcherAspect**
Overriding Default Behavior	**MicFwViewPort**	**defaultMove: aList onto: aViewportOrNil val-ueOf: anAspect**
Overriding Default Behavior	**MicFwViewPort**	**defaultProhibitedValueOf: aDispatcherAspect**
Overriding Default Behavior	**MicFwViewPort**	**defaultProvidedOperationsValueOf: anAspect**

Type	Implementor	Message
Overriding Default Behavior	MicFwViewPort	defaultReadableValueOf: aDispatcherAspect
Overriding Default Behavior	MicFwViewPort	defaultShowTouchedValueOf: anAspect
Overriding Default Behavior	MicFwViewPort	defaultUndefined: anAbstractValue
Overriding Default Behavior	MicFwViewPort	defaultUpdateOfMetaControlList: aMetaControl-List forAspect: anAspect
Overriding Default Behavior	MicFwViewPort	defaultVisibleValueOf: aDispatcherAspect
Late Model Access	MicFwViewPort	<anContentAspectOfAGroupControl>Child-PaneConnectionMode
Late Model Access	MicFwViewPort	defaultChildPaneConnectionModeValueOf: anAspect

4.4.3. Method descriptions

4.4.3.1. Declaring

```
MicFwViewPort class>>isDefaultViewPort

MicFwViewPort>>defaultEmbeddedPartValueOf: dispatcherAspect

MicFwViewPort>>defaultMaxSearchLevelValueOf: dispatcherAspect
```

4.4.3.2. Initialize

```
MicFwViewPort >> initialize
```

4.4.3.3. Model Navigation

```
MicFwViewPort>>modelAtConnection: aConnectionName
```

4.4.3.4. Controlling Subaspects

```
MicFwViewPort>><aspectName>Accelerator

MicFwViewPort>><aspectName>AcceleratorText

MicFwViewPort>><anContentAspectOfAGroupControl>ChildPaneConnectionMode

MicFwViewPort>><columnAspectName>EmbeddedPart

MicFwViewPort>><detailsViewContentAspectName>ItemAspects

MicFwViewPort>><anAspect>Manipulator

MicFwViewPort>><anAspect>MetaControlListUpdate: aMetaControlList
MicFwViewPort>>defaultCanMove: aList valueOf: aSourceAspect
```

```
MicFwViewPort>>defaultCheckedValueOf: dispatcherAspect
```

```
MicFwViewPort>>defaultCopy: aList onto: aViewportOrNil valueOf:
anAspect
```

```
MicFwViewPort>><anAspect>Editable
MicFwViewPort>>defaultEditableValueOf: aDispatcherAspect
```

```
MicFwViewPort>>defaultEnabledValueOf: aDispatcherAspect
```

```
MicFwViewPort>>defaultLateModelAccessValueOf: dispatcherAspect
```

```
MicFwViewPort>>defaultLink: aList onto: aViewportOrNil valueOf:
anAspect
```

```
MicFwViewPort>>defaultManipulatorValueOf: anAspect
```

```
MicFwViewPort>>defaultMove: aList onto: aViewportOrNil valueOf:
anAspect
```

```
MicFwViewPort>>defaultProhibitedValueOf: dispatcherAspect
```

```
MicFwViewPort>>defaultProvidedOperationsValueOf: anAspect
```

```
MicFwViewPort>>defaultUpdateOfMetaControlList: aMetaControlList
forAspect: anAspect
```

```
MicFwViewPort>>defaultVisibleValueOf: aDispatcherAspect
```

```
MicFwViewPort>>graphicsPresentation
```

```
MicFwViewPort>>readOnlyAspect: anAspect
```

```
MicFwViewPort>>wantsCachedReadOf: aDispatcherAspect
```

```
MicFwViewPort>>wantsImmediateViewUpdateOf: anAspect
```

4.4.3.5. Aspect Value Conversion

```
MicFwViewPort>>getMetaControlListForAspect: anAspect
```

`MicFwViewPort>>getMetaControlListForAspect: anAspect ifAbsent: absence-Block`

`MicFwViewPort>>listHierarchyParentViewPort`

`MicFwViewPort>>newMetaControlListForAspect: anAspect`

`MicFwViewPort>>removeMetaControlListForAspect: anAspect`

4.4.3.6. Services

`MicFwViewPort>><anContentAspectOfAGroupControl>ChildPaneConnectionMode defaultChildPaneConnectionModeValueOf: anAspect`

`MicFwViewPort>>canStartInteractionForAspect: anAspect`

`MicFwViewPort>>childViewPortBaseClass`

`MicFwViewPort>>contextAtConnectionNamed: aSymbol`

`MicFwViewPort>>converter`

`MicFwViewPort>>deferUpdateInAllViewsWhile: aBlock`

`MicFwViewPort>>deferViewUpdateWhile: aBlock`

`MicFwViewPort>>enterInteractionForAspect: anAspect comingFromAspect: lastAspect of: lastViewPort`

`MicFwViewPort>>getLocaleNamed: name ofType: type fromContextAtConnec-tionNamed: aSymbol`

`MicFwViewPort>>itemViewPortBaseClassForAspect: anAspect`

`MicFwViewPort>>leaveInteractionForAspect: anAspect goingToAspect: nex-tAspect of: nextViewPort`

`MicFwViewPort>>localizeAllConnectionsTo: aLocale`

`MicFwViewPort>>localizeConnectionNamed: aString to: aLocale`

`MicFwViewPort>>resolveKey: aMicMlsKey forLocale: aLocale`

`MicFwViewPort>>returnContextNamed: aContextClassSymbol`

`MicFwViewPort>>readAccessRejectedForAspect: anAspect vetoValue: aPro-hibitedModel`

`MicFwViewPort>>shouldEmulateEditable`

`MicFwViewPort>>startInteraction: aConnection`

`MicFwViewPort>>startInteractionForAspect: anAspect`

MicFwViewPort>>stopInteraction: aConnection

MicFwViewPort>>writeAccessRejectedForAspect: anAspect vetoValue: aProhibitedModel

4.4.3.7. Overriding Default Behavior

The application framework provides a default behavior for some Subaspects. If this default behavior is suitable for your application you must not implement the corresponding Subaspect methods to control the behavior of your application.

If you want to change this default behavior for a specific Subaspect you must reimplement the following methods in the class:

MicFwViewPort>>defaultEditableValueOf: aDispatcherAspect

MicFwViewPort>>defaultEnabledValueOf: aDispatcherAspect

MicFwViewPort>>defaultLateModelAccessValueOf: aDispatcherAspect

MicFwViewPort>>defaultLink: aList onto: aViewportOrNil valueOf: anAspect

MicFwViewPort>>defaultManipulatorValueOf: aDispatcherAspect

MicFwViewPort>>defaultMove: aList onto: aViewportOrNil valueOf: anAspect

MicFwViewPort>>defaultProhibitedValueOf: aDispatcherAspect

MicFwViewPort>>defaultProvidedOperationsValueOf: anAspect

MicFwViewPort>>defaultReadableValueOf: aDispatcherAspect

MicFwViewPort>>defaultShowTouchedValueOf: anAspect

MicFwViewPort>>defaultUndefined: anAbstractValue

MicFwViewPort>>defaultUpdateOfMetaControlList: aMetaControlList forAspect: anAspect

MicFwViewPort>>defaultVisibleValueOf: aDispatcherAspect

4.4.3.8. Late Model Access

One problem within the application development is how to deal with huge and complex views. The application framework provides the possibility to defer the creation and connection of framework structures until they are necessary. This behavior is controlled by implementing the following protocols in the connected viewports:

Return an atom to control the connection mode Group Controls like Forms, Notebook Pages etc. .The Group Control presenting Aspect <anAspect> can be connected with the following modes (return values):

- EARLYCONNECT - Connect the models at startup. Default value.
- LATECONNECT - Connect the model when it is selected the first time (only Notebook Pages).
- BACKGROUNDCONNECT - Connect the models in a background process after startup (only Notebook Pages).
- LATEGROUPCONNECT - Late connection to models in Notebook Pages and Group Controls with content hidden.

4.4.3.9. Undefined Models

Within the framework exists a special object which represents similar to Smalltalk UndefinedObject an initial state for the Framework. This object is the . It is set by the in every structure which requires a model object and is not being initialized by the application.

4.4.3.10. Direct View Manipulation through Manipulators

A manipulator is an object intended for communicating information from within an application´s model to a view. Information contained in the manipulator will be used by the Adapter of the view.

The manipulator itself is a Subaspect of the model, just like ...enabled and ...visible. Since the manipulator is being evaluated in the Adapter (with the Adapter itself being passed to it as a parameter), it can be used for influencing Aspects of the view not connected to the Framework. This means that the manipulator has to contain all the necessary logic for providing behavior in the intent of the application.

An instance of a manipulator will be evaluated only once before it is discarded.. This means that for example for every update a new instance of the manipulator has to be created.

Valid Manipulators are :

- OneArgumentBlocks (the argument is the Adapter)
- DirectedMessages with one parameter (the argument is the Adapter)
- an instance of a subclass of (API:)

Example:

Setting the font size of Label connected to the Aspect #personName

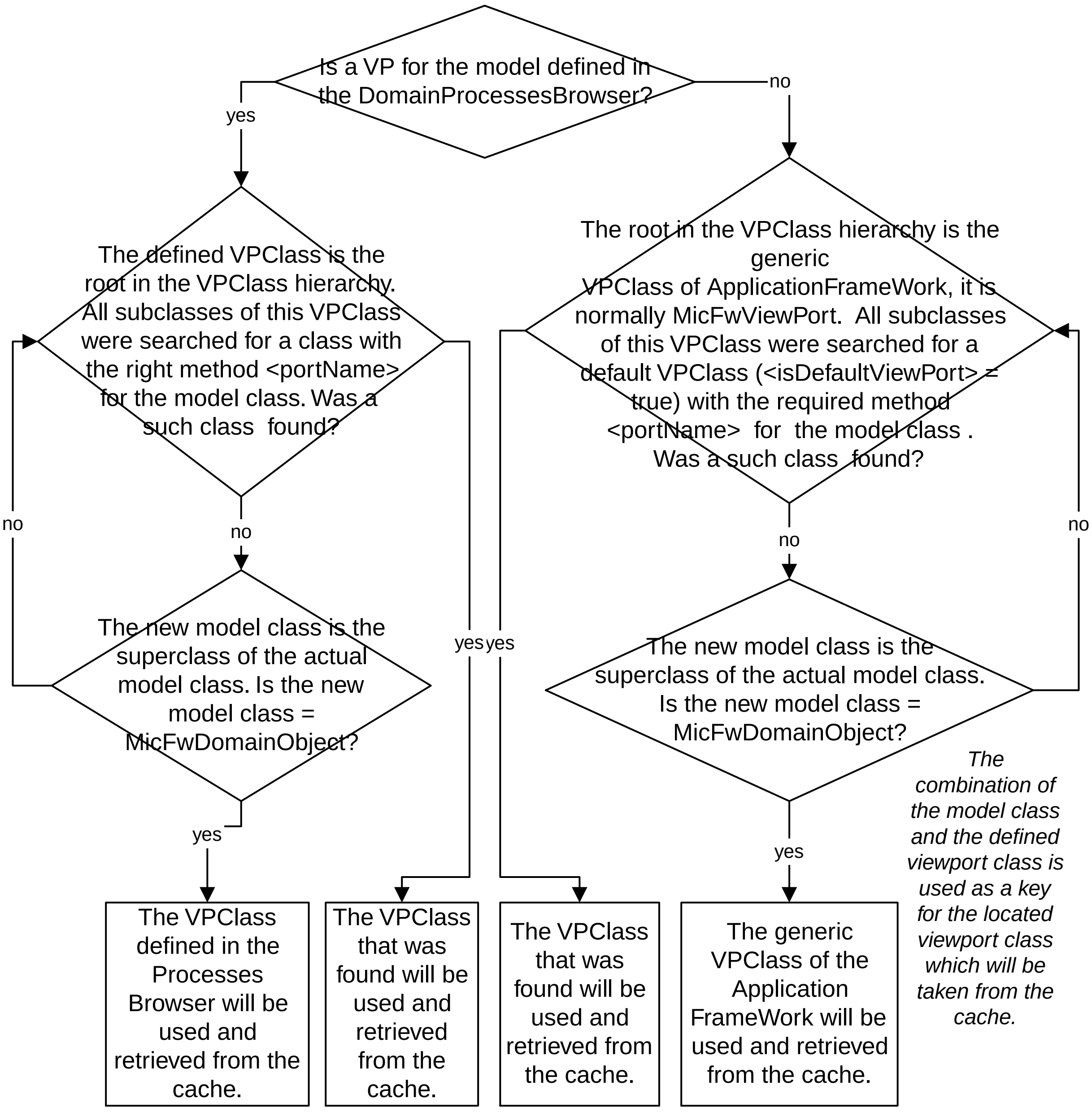
```
PersonViewPort >> personNameManipulator
  ^[:adapter | adapter setFontSizeTo: 20]
```

In this case the Adapter has to implement a method . Accessing the Adapter´s window from within the Manipulator should be avoided:

4.5. ViewPort-RequestBroker

4.5.1. How the ViewPortRequestBroker searches for a ViewPort

The ViewPortRequestBroker search the right ViewPort for DomainObjects and DomainProcesses. You can change this broker when you need other rules.



Is a VP for the model defined in the DomainProcessesBrowser?
yes
no
The defined VPClass is the root in the VPClass hierarchy. All subclasses of this VPClass were searched for a class with the right method <portName> for the model class. Was a such class found?
The root in the VPClass hierarchy is the generic VPClass of ApplicationFrameWork, it is normally MicFwViewPort. All subclasses of this VPClass were searched for a default VPClass (<isDefaultViewPort> = true) with the required method <portName> for the model class . Was a such class found?
no
no
no
no
yes
yes
The new model class is the superclass of the actual model class. Is the new model class = MicFwDomainObject?
The new model class is the superclass of the actual model class. Is the new model class = MicFwDomainObject?
yes
yes
The VPClass defined in the Processes Browser will be used and retrieved from the cache.
The VPClass that was found will be used and retrieved from the cache.
The VPClass that was found will be used and retrieved from the cache.
The generic VPClass of the Application FrameWork will be used and retrieved from the cache.
The combination of the model class and the defined viewport class is used as a key for the located viewport class which will be taken from the cache.

4.5.2. Using a custom ViewPortRequestBroker

When the tools are loaded in the development image, #technicalViewPort and #customViewPort will be added to the broker mapping. After the tools are loaded, a #MicFwViewPortRequestBrokerDispatcher is registered under the mapping #viewPort (see application #MicFwProcessesBrowser class >> loaded). In order to ensure that the correct ViewPortRequestBroker will be used for development tools, the processes of the development tools are marked with <isTechnicalDomainProcess>.

The #MicFwViewPortRequestBrokerDispatcher will then forward all calls to the #customerViewPort-Broker or the #technicalViewPort-Broker.

Any CustomViewPortRequestBroker's should be registered in the broker mapping. For example: self broker use: ACustomerVPBroker for: #customViewPort

A #MicFwViewPortRequestBroker or a #ACustomerVPBroker must be available for the runtime via the #viewPort mapping. See application #MicFwViewPorts class >> resetMappings

The following methods can be implemented to register #ACustomVPBroker for development and runtime separately:

```
Application: #UserApplication class >> packagingRulesFor: aRuleCollector
"..."
aRuleCollector replaceClassMethod: #registerForDevelopment with: #registerFor-
Runtime inClassNamed: #ACustomVPBroker

ACustomVPBroker class >> registerForRuntime
self broker use: ACustomVPBroker for: #viewPort;

ACustomVPBroker class >> registerForDevelopment
self broker use: ACustomVPBroker for: #customViewPort;

ACustomVPBroker class >> register
self registerForDevelopment
```

The method ’register’ will be called when the application is started.

Note: After the above has been completed: Loading the application #MicFwViewPorts can overwrite the Dispatcher (and causes errors).

4.6. Authorization

The Application Framework offers authorization mechanisms for both Domain Objects and Domain Processes. Managing authorization at this level of an application provides for greater consistency within the scope of an application than managing it at the GUI level does. Also, it helps in avoiding security loopholes.

For Domain Objects and Domain Processes, read and write accesses to an object´s attributes can be controlled. Only for Domain Processes, the right to perform Aspects of the object is controlled.

The authorization mechanism is class-based. This means that authorization rules will be the same for all instances of the controlled class.

Let´s look at validation for read and write accesses first. The Frameworks´ interface provides the following protocol for managing access to a Domain Object´s and Domain Processes attributes (in
class) :

Method	Method Description
authorizeReadUsing: aRule	authorize read access to all the object´s Aspects based on the result of evaluating <aRule>. <aRule> is either a block or a message which is valued at ‘true’ (to allow access) or ‘false’ (to deny access)
authorizeRead: aspect using: aRule	authorize read access to <aspect> of a Domain Object based on the result of evaluating <aRule>
authorizeRead: aspect using: aRule ifProhibitedReturn: value	authorize read access to <aspect> of a Domain Object based on the result of evaluating <aRule>. If the access is denied, return <value>
authorizeWriteUsing: aRule	same as above for write access
authorizeWrite: aspect using: aRule	same as above for write access

Note : if <aRule> is a block, it can have either zero or one argument. If it has one argument, the Domain Object the attribute of which is about to be read or written will be passed as the argument. This feature allows for authorization based on the Domain Object´s state, which might be interesting in contexts like "user can set an insurance policy´s status to 'accepted' only if the insured sum is less than $250,000".

Example:

Suppose we have a class with the following definition :

```
MicFwDomainObject subclass: #MicPerson
instanceVariableNames: 'name firstName '
classVariableNames: ''
poolDictionaries: ''
```

We want to allow changing a ´s name only on Sundays. The corresponding code would be

```
MicPerson authorizeWrite: #name using:
  [Date today dayIndex = 1]
```

If a statement like

```
aMicPerson name: 'Smith'
```

was executed on a weekday, the name would not be changed. Also, the Domain Object would trigger the event #prohibit. If the access is in fact authorized, the attribute´s value will be changed and the event #permit will be triggered by the Domain Object.

Suppose we want to limit read access to an attribute and return a designated value if access is prohibited. This would look like

```
MicPerson authorizeRead: #name
  using: [Date today dayIndex = 1]
  ifProhibitedReturn: '***'
```

Read access to the attribute #name will be allowed on Sundays. In all other cases, the string '***' will be returned instead of the person´s name.

For Authorization of perform Aspects in Domain Processes, the protocol is somewhat similar, the major difference being that now perform access is controlled.

- authorizePerformUsing: aRule
- allow execution of all the receiver class´ Aspects based on the result of evaluating the block (or message) <aRule>
- authorizePerform: aspect using: aRule
- allow execution of the receiver class´ <aspect> depending on <aRule>

Example

Suppose we want to control who can commit changes made to a person. We´ll assume there is a class User with a single instance that understands #canChange. We´ll also assume that editing a person is managed by a Domain Process called PersonEdit.

```
PersonEdit authorizePerform: #commit
   using: [User current canChange]
```

will do what we intend.

4.7. Validation

A common feature found in most applications is validation of user input, or, more generally, of attribute values altogether. The Application Framework supports validation of a Domain Object´s and a Domain Process attributes.

The protocol for controlling validation is extremely simple and consists only of a single method, implemented in   class :

validateWrite: anAspect using: aRule

This methods validates write access to <anAspect> by evaluating <aRule>. <aRule> can be a block or a message.

Example

Suppose we have a class Car that has, among others, an attribute named 'weight'. We want to make sure that weight only holds values between 1000 and 4000 lbs.

```
Car validateWrite: #weight aRule: [:value |
   (value notNil and: [value >= 1000]) and:
   [value <= 4000]]
```

The attempt to set the 'weight' attribute to anything outside this range will simply fail, i.e. the original value will not be changed.

The validated object (the instance of Car, in this case) triggers three different events. Just before the validation rule is evaluated, the event #validate is triggered. Depending upon the outcome of this evaluation, either #valid or #invalid is triggered.

The most common scenario is the handling of invalid write attempts. This is demonstrated in the sample application.

The validation of a perform aspect (e.g. button) effects the enabled state of controls.

4.8. Group Controls

The Application Framework distinguish two ways using Group Controls like Forms, Notebooks, Menus etc. The first is called static. Here you only control the properties like visibility, labeling, etc. of the specific view component. The second, called dynamic, additionally manipulates the content of such a Control. A notebook for example may change it's number of pages during some states of your application.

In this section we discuss the different ways of using Group Controls.

4.8.1. Dymanic Versus Hidden Static View Components

Using dynamic view components has two sides. The first is, that you gain a lot of flexibilty within your application. On the other hand, you will loose a lot of performance if you use the dynamic extensive. This results on the fact, that each time you exchange view components, widgets must be closed and openend, the framework must reconnect to the new structures and so on. These facts should be considered while designing the application apperance.

In many cases a dynamic view behavior can be emulated by using a show / hide concept of view components. These behavior could be easily implemented via the framework by implementing the visible protocol in the framework. The great advantage of this concept is the reaction time of the application for the user.

Using static views opens as well many tuning mechanisms provided by the framework. One of these is the use of Cached Views (see chapter Cached Views Cached Views).

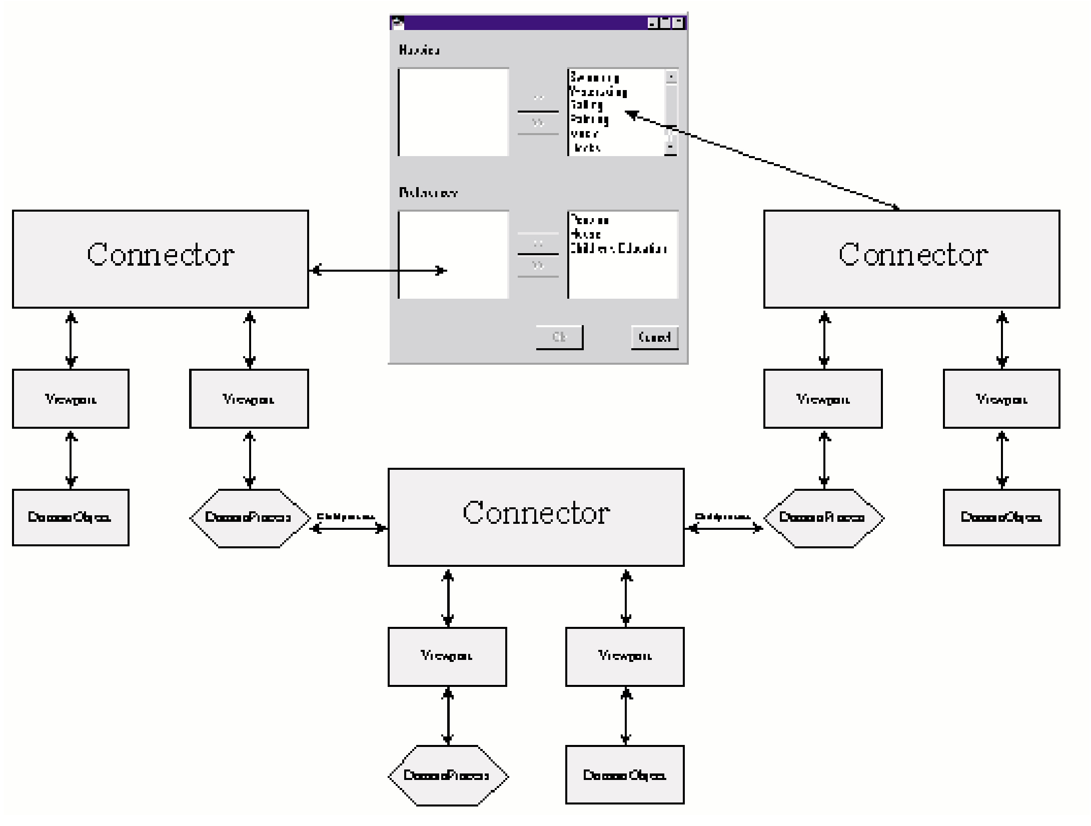
4.8.1.1. Using Static Group Controls

The Application Framework provides a mechanism for connecting Group Controls like Forms, Group Boxes and Notebook Pages in the same way.

All these Controls are connected via a four-part Model Accessor name:

`<accessed model>_<aspect>_<default base>_<default process>_`

- <accessed model>: Connection Name of the model accessed for getting the content of the control
- <aspect>: protocol for accessing the content and controlling the behavior
- <default base>: name of the Base Connection to be accessed by the child controls as default
- <default process>: name of the child Process Connection to be accessed by the child controls as default



Via the declared Aspect <aspect> you can control the Group Control with the Viewport methods for enabling, visiblity, etc. The Aspect can be also used to set the content of the Group Control (see Using Dynamic Group Controls Using Dynamic Group Controls), to implement a dynamic behavior.

Declaring a <default process> will include the following:

Child Controls for the connected part will access this Domain Process as default process and use its Model View Connector for resolving Connection Names. I.e. this implies that a separate Connector will be created for the part. Any Connection request from within that part will then be directed towards that Connector.

If no valid <default base> has been set, the Default Base Connection corresponding to the declared process will be used. Any transaction declared for the <default process> will be started at initialization.

In order to select a Group Control (Page, Form ...) the method selectAspect: should be used. It should be send to the process to which the page is connected. For example, if a page in a notebook is named

```
editAddress_secondPage_
```

you have to call

```
self editAddress selectAspect: #secondPage
```

in the parent process to bring this page to front.

The released version of the Application Frameworks contains a few examples. applications illustrates the reuse of Forms and Notebook Pages. These can be examined by the Examples Browser:

4.8.1.2. Using Dynamic Group Controls

In some cases it might be nessesary to exchange parts on the view dynamically or to add /remove new parts. The framework support this in an easy way. But you should always keep in mind, that you must pay a price for this feature. This price is a lack of performance, due to the fact that every exchange of view components causes a reconnect of framework structures (see section Dymanic Versus Hidden Static View Components Dymanic Versus Hidden Static View Components).

The content of all Group Controls (i.e. canvas) is an instance of . This collection reacts similar to an OrderedCollection, so you can use this api to modify the content of the MetapartCollection.

Each type of Group Control deals with other components. So the content of each metapart collection differs due to the connected view widget.

4.8.2. Overview: Instance methods

Type	Implementor	Message
Metapart Collections	**MicFwViewPort**	**getMetaControlListForAspect: anAspect**
Metapart Collections	**MicFwViewPort**	**getMetaControlListForAspect: anAspect ifAbsent: absenceBlock**
Metapart Collections	**MicFwViewPort**	**newMetaControlListForAspect: anAspect**
Metapart Collections	**MicFwViewPort**	**removeMetaControlListForAspect: anAspect**
Dynamic Group Controls	**MicFwMetaPart**	**accessorString: aString partClassName: aSymbol**
Dynamic Group Controls	**MicFwMetaPart**	**label: aLabel**
Dynamic Notebooks	**MicFwMetaPart**	**accessorString: aString partClassName: aSymbol**
Dynamic Notebooks	**MicFwMetaPart**	**icon: aIcon**
Dynamic Notebooks	**MicFwMetaPart**	**label: aLabel**
Dynamic Notebooks	**MicFwMetaPart**	**positionHint: aPositionHint**
Dynamic Notebooks	**MicFwMetaPart**	**tabType: aSymbol**
Dynamic Container Details View	**MicFwMetaAspect**	**aspect: aValue**
Dynamic Container Details View	**MicFwMetaAspect**	**columnWidth: aValue**

Type	Implementor	Message
Dynamic Container Details View	MicFwMetaAspect	positionHint: aValue
Dynamic Container Details View	MicFwMetaAspect	title: aValue

4.8.3. Method descriptions

4.8.3.1. Metapart Collections

A metapart collection, i.e. an instance of the class , is a specialized collection object, which maintain the changes within and enforces the update on components, which have changed. Therefore it provides a significant performace gain. All Group Controls requires MetapartCollections as content.

The exchange of view components is a view dependend Aspect and should be implemented within the responsible viewports. The Viewport class provide an api to create and access Metapartcollections for an Aspect. These methods are:

```
MicFwViewPort>>getMetaControlListForAspect: anAspect

MicFwViewPort>>getMetaControlListForAspect: anAspect ifAbsent: absence-
Block


MicFwViewPort>>newMetaControlListForAspect: anAspect

MicFwViewPort>>removeMetaControlListForAspect: anAspect
```

You can use the API of an OrderedCollection to add and remove Items. If you want to add more than one item, it is recommended to use the protocol.If you replace the complete collection, all former meta-parts will be removed from the system.

4.8.3.2. Dynamic Group Controls

The simplest Group Controls are Forms, Group boxes and Windows. These canvases usually group other widgets on the view. If a canvas is connected to the framework, one can set as the content of the canvas other Forms. These Forms are attached to the surrounding Form and are arranged so on each other. So it is not possible to arrage the inserted Forms in such a way that all are visible at the same time.

The content accessor for a canvas so return a metapart collection containing a number of . Each Metapart represents one Form to be inserted.

```
MicFwMetaPart>>accessorString: aString partClassName: aSymbol

MicFwMetaPart>>label: aLabel
```

Example

A Form on the view is connected under the Aspect details. We want to exchange the content of this Form dynamically. So we must implement in our Viewport a method similar to this:

```
Details
”return a form representing the details of our selected model”
| mpc |
mpc := self newMetaControlListForAspect: #details.
mpc add: (
   self model currentChoice isNaturalPerson
   ifTrue: [MicFwMetaPart new
      accessorSting: ‘VOID myDetail’
      partClassName: #MyNaturalPersonForm;
      yourself]
   ifFalse: [MicFwMetaPart new
      accessorString: ‘VOID myDetail’
```

```
        partClassName: #MyPersonForm;
        yourself]).
    ^mpc
```

4.8.3.3. Dynamic Menu

The Application Framework offers dynamic support for standard Menus and item Menus. Basically, every Menu is connected using the appropriate GUI tool, to e.g. a container or a Menu bar. In order to be able to control a Menu dynamically, i.e. to determine the current Menu items, the Menu must be connected to a Connector. This is done by using a two-part protocol:

```
accessed model_accessor protocol_
```

- <accessed model>:Connection Name where the Menu is connected
- <accessor protocol>:protocol for accessing the Menu item list

The Menu is connected via the Viewport - implementing the <accessor protocol> in the process itself will not show the desired results. This method is called at least once the first time the Menu is accessed. Depending on the model which is affected by that call, the method will be called everytime one of the affected models changes.

The method must return an MicFwMetaPartCollection which consists of   objects, where   can be any of:

- MicFwMetaCommand
- An instance of this class will hold the definition for a Menu item. The instance variables to be set are:
  - #accessor: This is the accessor for the command method within the respective process or viewport.
  - #label: This is the string to be displayed as the Menu item label.
  - #mnemonic: Is the underlined character in the menu label,.which allows the menu action to be performed.
  - #accelerator: Defineds the Shortcut to perform the menu action
  - #commandParameter: Defines the parameter object which is to be used for the menu command.

The following protocols are available for setting the instance variables:

- accessorString: anAccessorString label: aLabel
- accessorString: anAccessorString label: aLabel mnemonic: aMnemonic
- commandParameter: anObject
- manipulator: aBlock
- acceleratorText: aString
- acceleratorDescription: aDescriptionString. see: Dynamic Menu Item with Accelerator and Direct View Manipulation through Manipulators.
- MicFwMetaSubMenuRequester. An instance of this class will hold the definition for a cascade Menu item. The instance variables to be set are the same as in MicFwMetaCommand, except that #accessor specifies the Connection protocol for the submenu.
- MicFwMetaSeparator. An instance of this class will be represented by a Menu item separator in a Drop-down or Popup Menu. No instance variables have to be initialized.

Example:

```
menuItems
"Return a MetapartCollection of meta for the dynamic menu items."
| result |
result :=
  self getMetaControlListForAspect:#menuItems
    ifAbsent:[ (self newMetaControlListForAspect:#menuItems)
  add: ( MicFwMetaCommand new
    accessorString: 'testCommand'
    label: 'first test item';
    yourself);
  add: ( MicFwMetaSeparator new);
  add: ( MicFwMetaSubMenuRequester new
    accessorString: 'testSubMenuItems'
    label: 'my submenu';
    yourself); "THIS IS A CASCADED MENU ITEM"
  yourself].
^result

testSubMenuItems
```

```
"Return a MetapartCollection of meta controls for the dynamic menu items for
the dynamically created submenu declared above."
| result |
result :=
   self getMetaControlListForAspect:#testSubMenuItems
      ifAbsent:[ (self newMetaControlListForAspect:#testSubMenuItems)
   add: (MicFwMetaCommand new
      accessorString: 'subMenuTestCommand1'
      label: 'first submenu item';
      yourself);
   add: (MicFwMetaCommand new
      accessorString: 'subMenuTestCommand2'
      label: 'second submenu item';
      yourself);
   yourself ].
^result
```

For item Menus there is a new selection model. There is a method in the Domain Process for accessing the active model. Active means either being selected when the context Menu request key is pressed or being "cursored" with the right mouse button pressed. This method is . It returns nil if no item is active. If an item is active it returns the Domain Object corresponding to the active item.

Menu Command with a Parameter

If you need a menu command with a parameter then use the method <commandParameter: aParameter>.

Example

```
MicFwMetaCommand new
   accessorString: 'testCommand'
      label: 'first test item';
   commandParameter: aParameter;
   yourself.
```

When the menu item was pushed, the sends the message <testCommand: aParameter>.

4.8.3.4. Dynamic Menu Item with Accelerator

A menu item can be accessed by pressing a key-combination if an accelerator for the menu item has been defined. However, this applies only to menu items that are registered with the current view. In other words, for dynamic menus (ie, menus that change depending on the current view), only the accelerators whose menu items are accessible can be executed.

```
MicFwMetaCommand new
   accessorString: 'testCommand'
   label: ( MicFwAbstractLabel new
      labelString:  'Test';
      labelType: 2;
      yourself )
   mnemonic: $2;
   acceleratorDescription: 'Control+2';
   acceleratorText: 'Ctrl 2';
   yourself.
```

4.8.3.5. Using Abstract Label

Strings, CgPixMap's and AbtIcon's can be used as labels. However, its important that the correct label type is set for the abstract label. The label types are defined in the PoolDictionary CwConstants. Valid types are **XmPIXMAP**, **XmSTRING**, **XmICON**.

String label examples:

```
MicFwMetaCommand new
   accessorString: 'testCommand'
   label: 'Test';
   yourself
```

or:

```
MicFwMetaCommand new
   accessorString: 'testCommand'
   label: (MicFwAbstractLabel new
      labelString: 'Test';
      labelType: 2;"XmSTRING");
```

```
    yourself
```

PixMap's or Icons to be displayed must be defined with a GraphicsDescriptor. The definition is set using

MicFwGraphicsDescriptorMapper >> use: aResponsibleObject for: aCategorieSymbol

Where <aResponsibleObject> is the graphic and <aCategorieSymbol> the key. <aResponsibleObject> can be a CgPixMap or an AbtIcon.

Example for a button in a tool bar:

```
MicFwGraphicsDescriptorMapper
    default use: aCgPixMap for: #pixMapA;
    default use: aCgPixMap for: #pixMapB.
pixLabel := MicFwAbstractLabel new
    disabledGraphicsDescriptor: #pixMapA;
    enabledGraphicsDescriptor: #pixMapB;
    labelType: 1."XmPIXMAP"
MicFwGraphicsDescriptorMapper
    default use: anAbtIcon for: #abtIconA;
    default use: anAbtIcon for: #abtIconB.
iconLabel := MicFwAbstractLabel new
    disabledGraphicsDescriptor: #abtIconA;
    enabledGraphicsDescriptor: #abtIconB;
    labelType: 3."XmICON"
```

Such a Label object can be used, for example, in a toolbar. The label that corresponds the status (enabled/disabled) of the menu item will be automatically displayed.

4.8.3.6. Dynamic Notebooks

Notebooks can have a Model Accessor (means you have to name the Notebook widget in GUI e.g. "pages_") which allows the Control of dynamically adding and removing Notebook Pages.

This accessor which is placed in a Viewport (e.g. the Viewport of your defaultProcess of your view) must return an metapart collection of instances of class MicFwMetaPart.

You can add dynamic pages to a notebook by following this List:

- name your Notebook <connection>_<accessor>_
- implement a method <accessor> in the viewport of <connection> which returns a metapart collection

`MicFwMetaPart>>accessorString: aString partClassName: aSymbol`

`MicFwMetaPart>>icon: aIcon`

`MicFwMetaPart>>label: aLabel`

`MicFwMetaPart>>positionHint: aPositionHint`

`MicFwMetaPart>>tabType: aSymbol`

- MAJOR
- MINOR
- NONE.

Tab types are ignored by a windows Notebook, due to the fact all pages are major.

Example:

Pattern person and its roles: Notebook is named "person_roles_".

```
Roles
"return a MetaPartCollection of meta information of our dynamicPages"
|result|
result := self getMetaControlListForAspect: #roles
    ifNil: [self newMetaControlListForAspect: #roles].
self model roles do:[:aRole| (aRole isMemberOf: Manager)
    ifTrue:[
        result add: MicFwMetaPart new
            accessorString: 'VOID VOID VOID VOID'
```

```
      partClassName: #ManagerView
      label: ( MicFwAbstractLabel new
        labelString: aRole asString ;
        labelType: 2;
        yourself);
    yourself].
ifFalse:[
  result add:
    MicFwMetaPart new
      accessorString: 'VOID VOID VOID VOID'
      partClassName: #ClientView
    label: aRole asString;
  yourself]].
^result
```

In contrast to the name convention at the composition editor the name parts of the accessorString at MetaParts are seperated by spaces and the last character is not a space. Only name parts which begins with a lower case letter are interpret by the framework.

Changing this collection will result in the dynamic Notebook Pages being closed and replaced by the newly declared ones.

The following side effects have to be considered for pages declaring child process access:

Any transaction declared for the child process will be started. When removing a page, the transaction for the child process will NOT be ended, but remain open until it is committed or aborted by a command or the view is closed.

4.8.3.7. Dynamic Container Details View

The content of the widget AbtContainerDetailsView are colums. These columns are represented within the framework by instances of the   class. Therefore the content accessor of a connected container details view returns a metapart collection containing meta Aspects.

`MicFwMetaAspect>>aspect: aValue`

`MicFwMetaAspect>>columnWidth: aValue`

`MicFwMetaAspect>>positionHint: aValue`

`MicFwMetaAspect>>title: aValue`

The type and number of colums in a details view can be changed dynamically. A
    , which is obtained from a subaspect, will control the columns. The name of the subaspect is comprised of the content accessor of the details view and the string 'ItemAspects'. The subaspect is implemented in the connection viewport of the connection that returns the content for the details view.

```
ContentViewPort >> <detailsViewContentAspectName>ItemAspects
```

Example:

```
contentListItemAspects
|result|
result := self
  getMetaControlListForAspect:#columnsA
    ifNil:[ self newMetaControlListForAspect:#columnsA;
  add: (MicFwMetaAspect new
    accessorString: 'defaultProcess col4';
    title: 'ColumnTitel';
    columnWidth: 80;
    yourself)].
^result
```

The MetaPartCollection can be changed. Any changes will be displayed in the details view.

4.8.3.8. Direct editing in Container Details Views

When a container details view is direct-editted, a control in the cell of the container details view that has the focus is opened. The value of the cell can be defined with this control.

In order to use direct editting in the container details view, the edittable setting for the detail view and the column in the settings pane of the Composition Editor must be set to true.

Define the partName property for the column. Do not define the aspectName or attributeName properties.

There are 6 types of standard embedded controls:

- ##TEXT -> Entry field
- ##DROPDOWNLIST -> Drop down list
- ##COMBOBOX -> Combo box
- ##RADIOBUTTONS -> Radio button set
- ##TOGGLEBUTTON -> Toggle button
- ##NULL -> it opens no embedded part (for a not ediable cell)

The DomainModel ViewPort (   ) that is displayed in the details view cell is the connection that specifies the type of direct editting used. An aspect is defined in the property partName column (eg, column1).

Content for the details view is obtained via the viewport for the connection (   ). The method that returns the aspect for the column is implmented in this viewport. The column displays the object using this aspect.

```
ViewPortA >> column1
  ^#aspectXY
```

Subaspects for enabling and editting can be implemented in   .

```
ViewPort >> <aColumnAspect>Editable
ViewPort >> <aColumnAspect>Label
```

Example

```
ViewPortA >> column1Editable
  ^true
ViewPortA >> column1Label
  ^'ColumnLabel'
```

A column is edittable if:

- the subaspect (eg, <column1Editable>) returns true

or:

- the edittable subaspect is not implemented.

The column requests the object in the focused cell for the embedded part. The desired embedded part for the cell will returned from another subaspect. This name of the subaspect is <aColumnAspect>Embedded-Part (e.g. column1EmbeddedPart). The subsaspect is implemented in   . If no subaspect is defined, the default embedded part ##TEXT will be used.

```
ViewPortB >> column1EmbeddedPart
  ^MicFwEmbeddedMetaPart new
    accessorString: 'defaultItem name'
    type: ##TEXT
    initiallySelected: true;
  yourself
```

The embedded part is connected to the cell only while the cell has the focus. If the cell looses the focus, the part is disconnected.

The construction of the accessors is similar to that of "normal" controls, with the exception of a reserved accessor part **#defaultItem** (#defaultItem is reserved like #defaultProcess). The accessor part #default-Item corresponds to the item of the selected row. Thus, the selection is already made, and must not be obtained from the model. In the example code for the ##TEXT embedded part, the name of the selected item is displayed in the entry field (-> 'defaultItem name').

Example

A list with objects, each of which uses the ViewPortB. The column partName is 'defaultProcess_column1_'. ViewPortA of the default process implements:

```
ViewPortA>>column1
  ^#aspectXY
ViewPortB>>column1EmbeddedPart
  ^MicFwEmbeddedMetaPart new
    accessorString: 'defaultItem name'
    type: ##TEXT
    initiallySelected: true;
  yourself
```

The entry field ('defaultItem name') displays the name of the selected item. This name can be editted.

```
ViewPortB>>column1EmbeddedPart
```

```
   ^MicFwEmbeddedMetaPart new
     accessorString: 'defaultItem entireList defaultItem theSelection VOID'
     type: ##DROPDOWNLIST;
   yourself

ViewPortB>>column1EmbeddedPart
   ^MicFwEmbeddedMetaPart new
     accessorString: 'baseList theEntireList baseSelection theSelection first
name VOID VOID'
     type: ##COMBOBOX;
   yourself

ViewPortB>>column1EmbeddedPart
   ^MicFwEmbeddedMetaPart new
     accessorString: 'defaultItem entireList defaultItem theSelection VOID'
     type: ##RADIOBUTTONS;
   yourself

ViewPortB>>column1EmbeddedPart
   ^MicFwEmbeddedMetaPart new
     accessorString: 'defaultItem toggleButton'
     type: ##TOGGLEBUTTON;
   yourself

ViewPortB>>column1EmbeddedPart
   ^MicFwEmbeddedMetaPart new
     accessorString: ''
     type: ##NULL
     initiallySelected: true;
   yourself
```

4.9. Collections

A special but frequent scenario is the use of instance collections, which can be displayed in all List-like Controls. Normally, a collection of Domain Objects will be connected by means of some kind of Collection Holder which maintains the collection; this must be implemented like any other Domain Model to be connected to a view via Application Framework.

This class - or the corresponding viewport class - must understand the and possibly . These are called under the following circumstances:

- Any time the user selects a List item, the message will be called; the parameter passed to this method is the selected object (if the control's abstract type is) or a collection with the selected objects (for).
- Each time the List is to be updated - this is the case immediately after view creation or when the control gets the focus, or when the user selects an item displayed in the List - the message is sent.
- The message can also be used by the application programmer to implement a default selection when the List appears the first time.

As mentioned earlier, it is not necessary in the first place to write all Domain Model Accessor methods if the collection holder keeps instance variables for maintaining both the collection and selection.

Example:

In order to prepare for some editing and deleting, the members of the fitness club shall be displayed in a List box. It will be possible to select a specific member, which can then be processed by pushing a button.

To support this scenario, the following things must be implemented:

- A List Control named club_members_default_selectedMember_.
- Please note that the members themselves will be retrieved through the 'club' Connection; whereas the selected member will be managed by the Domain Process which is accessible through the 'default' Connection.
- Accessor methods in the Domain Process. In the example, you´ll see that there is a Connection Names 'currentMember' which holds an instance of . The methods are used to convert between a single member and a collection holding that single member, as List Controls always require a collection for their selection (there are List Controls which can have multiple selected items)

4.10. List Connections

Each single instance in the collection to be displayed must be a Domain Object. For administration of these Domain Objects the Model View Connector internally maintains a List Connection (instance of the framework class MicFwListModelConnection).

This Connection has an internal technical name, however, it is possible to manually create an empty List Connection at Connector initialization with the contents selector as Connection Name. This preset List Connection will be used by the List Control and can be accessed through its Connection Name.

The List Connection holds a pseudo model which responds to the methods   (the List Control contents),   (the current selection), and   (the currently chosen, i.e. doubleclicked model).

An example for processing the double-click in the default Domain Process:

```
modelChosen: aListConnection
| models |
models := aListConnection model selection.
"more processing..."
```

After executing the sequence above, models holds a collection with the selected Domain Object.

4.11. Cached Views

The present release provide a   class, which is meant as a simple base class for creating windowing policy classes which meet the special needs (e.g. creation of complex views in background, removal of all Cached Views when the application is closed).

```
MicFwDomainProcess class >> defaultWindowingPolicyClass
```

Alternate windowing policy classes are activated overriding the method as described above.

For detail of using windowing policies refer to section Windowing Policy Windowing Policy

4.12. Drag & Drop

Drag & Drop within the framework enable the developer to abstract from the implementation details of Drag & Drop and lead him to a more technical use of the mechanism.

The complete process control of Drag & Drop is done by implementing some methods in the viewports of the models connected to the widgets on the screen. These methods cover all dynamic Aspects of drag drop (allow / deny actions, etc.)

4.12.1. Concept of Drag & Drop

In the internal layers a Drag & Drop session is represented as following: If a drag session start from VA, a Control and a Drag & Drop Connection will be created in the specific Connector the source model is connected. The Drag & Drop Connection will connect all models which are involved in the Drag & Drop process. The Drag & Drop Control is the framework interface to the VA Drag & Drop session (via its implementation).

The Drag & Drop Connection also contains some additional information of the current session, so if more information about the current Drag & Drop state is needed, you can look in this Connection (named 'dragdrop').

4.12.2. How to Use Drag & Drop

To use Drag & Drop within your application the following protocol must be implemented in the specific viewports of the involved Domain Objects.

Notations:

- < aString > : always mean a String, for example an Aspect name like 'contents.'
- <anOperation> : is one of the following Strings 'Copy' , 'Move' or 'Link'

In concatenated methods, like <anAspect><anOperation><anotherAspect>, each Part in <> starts with an uppercase letter, except the first one.

Example

```
personsMovePersonList
```

4.12.2.1. Enabling Drag & Drop

To enable Drag & Drop from a view widget one of the following methods must be implemented in the Viewport of the connected model:

```
<anAspect>AcceptableOperations
"valid drop operations performed on the receiver "
  ^MicFwOperations enableNone enableCopy enableLink enableMove

<anAspect>ProvidedOperations
"valid drag operations could be started from the receiver"
  ^MicFwOperations enableNone enableCopy enableLink enableMove
```

The operation named   defines the valid drop operations on the receiver and the method   names the possible operations which could be started. Each method must return an instance of the class   which contains the possible operations (see above). The Aspect should be the   Aspect on the view.

The following example methods show how to describe the behavior of a the model with the Aspect contents. The model can start move operations and accept copy and move operations from other models.

```
contentsAcceptableOperations
  ^MivFwOperations enableCopy enableMove

contentsProvidedOperations
  ^MicFwOperations enableMove
```

Or you may reimplement the default methods to enable Drag & Drop for various models.

```
defaultAcceptedOperationsValueOf: anAspect
"Answer the default acceptable operations for the receiver.
 The default is no operations"
  ^MicFwOperations enableNone

defaultProvidedOperationsValueOf: anAspect
"Answer the default operations could be started from the receiver"
```

```
   ^MicFwOperations enableNone.
```

The default operations will be set to no operations if we don't specify. This means Drag & Drop is disabled by default.

4.12.2.2. Dynamic Permit / Deny of Operations

If a Drag & Drop session is started the involved models will be asked each time necessary if the current operation is permitted or not. To control this behavior one must implement the following methods within the involved viewports:

The protocols tested in this order. After the first protocol with return a value not equal nil the test ended.

```
<aSourceAspect>Can<anOperation><aTargetAspect>: aListOfViewPorts
   ^true/false

<aSourceAspect>Can<anOperation>: aListOfViewPorts
   ^true/false

defaultCan<anOperation>: aListOfViewPorts valueOf: anAspect
   ^false
```

Example

```
personsCanCopyPersonList: aListOfDraggedViewPorts
   ^aListOfDraggedViewPorts size < 4
```

Copying from persons to personlist is allowed if we only drag less than 4 viewports of persons.

If we drag over a item in a List (for example in a icon area), first the cursored item is asked for the permission. If the item (i.e. his viewport) don't answer true or false the whole List is asked.

4.12.2.3. Performing Drop Actions

If a operation is permitted by the receivers model you must implement one of the following protocols to perform the correct actions done by the drop operation (the default behavior by the framework is to do nothing). Each method should return true or false to indicate the correct finish of the drop action.

The first parameter is a List of the dragged viewports, the second is the cursored Viewport or nil.

```
<anSourceAspect><anOperation><aTargetAspect>: aListOfViewPorts onto: aViewPor-
tOrNil
   ^true/false

<aSourceAspect><anOperation>: aListOfViewPorts onto: aViewPortOrNil
   ^true/false

default<anOperation>: aListOfViewPorts onto: aViewPortOrNil valueOf: anAspect
   ^nil
```

Example:

```
personsCopyPersonList: aList onto: aViewPortOrNil
   self personList addAll: (aList collect: [ :e | e model]
   ^true
```

4.12.2.4. Highlighting the Cursored Item

If you wish to highlight the item under cursor the target Viewport of the Drag & Drop process must implement the following protocol:

```
<aSourceAspect>ShowTouched
   ^true/false
```

Example

```
personsShowTouched
   ^true
```

In the personlist should be every person highlighted if some object's are dragged from persons to personlist.

Or you reimplement the default behavior which highlights every object if some object's are dragged over.

```
defaultShowTouchedValueOf: anAspect
   ^true "by default"
```

5

Tools

5.1. Overview

This section provides detailed information about the following Application Framework tools.

- The **Domain Processes Browser** provides complete functionality for working with domain processes.
- The **Connectors Browser** provides static and dynamic information for any existing connections. Such information is important for analysis and debugging.
- The **Framework Logger** can log any types of messages and thus aid in debugging.
- The **Online Debugger** provides detailed information for debugging.
- The **Visual Age Composition Editor** is not an Application Framwork tool; however, how the CE is used to enter connection information for parts is described.

The following tools are not actually part of the Application Framework; however, they are used quite often when working with Application Framework tools:

- The **Object Model Browser** supports analysis and modification of object models. The Transaction Browser is a part of the Object Behavior Framework and is desribed in the the ***Object Behavior Framework User's Guide***.
- The **Transaction Browser** supports analysis and modification of transactions. The Transaction Browser is a part of the Object Behavior Framework and is desribed in the the ***Object Behavior Framework User's Guide***.

5.2. Domain Processes Browser

5.2.1. Overview

Domain Processes play a central role in determining the structure and control flow of an application.

The Domain Processes Browser supports creation and maintenance of:

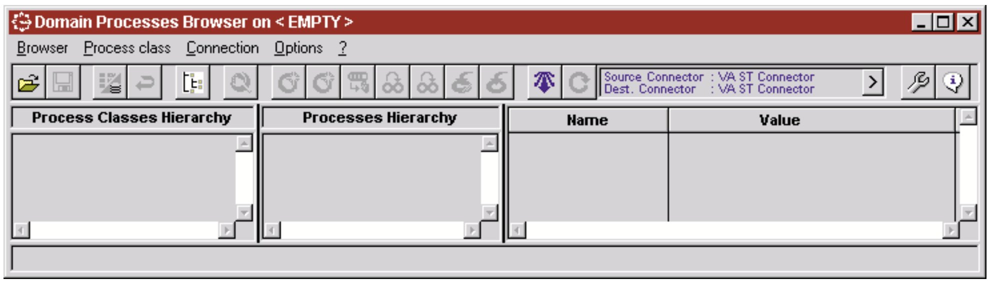
- Relationships between Domain Processes and Domain Models.
- Parent-child relationships between Domain Processes.

During design time, such relationships are implemented with Connections. These Connections are implemented by special accessor methods for processes and models that are created by the Domain Processes Browser.

During runtime, these special accessor methods create connections between the models and processes. Such connections provide a static aspect of the interaction between processes and models.

5.2.2. Main dialog

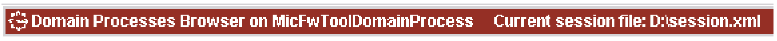
To open the DPB main dialog: From the **System Transcript** main menu: Select **micFrameworks / Open Domain Processes Browser...**. The following dialog appears:


Domain Processes Browser on < EMPTY >
Browser
Process class
Connection
Options
?
Source Connector : VA ST Connector
Dest. Connector : VA ST Connector
Process Classes Hierarchy
Processes Hierarchy
Name
Value

5.2.2.1. Title bar

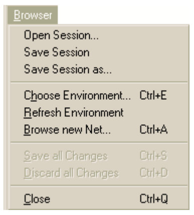
The title bar displays:

- Name of the process that the DPB was opened on
- Name of the session file (if any)



Domain Processes Browser on MicFwToolDomainProcess Current session file: D:\session.xml

5.2.2.2. Submenu Browser



5.2.2.2.1. Open session...

Opens the **Load from file** dialog. Double-click on a **.xml** file in this dialog to open a session that was previously saved to the .xml file.

5.2.2.2.2. Save session

Saves the session to the .xml file. This allows a DPB session to be saved without actually implementing the changes in VA. Thus, incomplete changes can be saved without throwing exceptions in the VA environment.

Note: If no session file has been selected: Same functionality as **Save session as...**.

5.2.2.2.3. Save session as...

Opens the **Save to file** dialog. Enter the name of the **.xml** file and click on **Save** to save the current session to the .xml file.

NOTE: Enter the extension **.xml** when entering the filename. The extension is NOT added to the filename automatically.

5.2.2.2.4. Choose environment... Ctrl+E

Opens the **Connector manager** dialog (see ‘Connector manager dialog’ (page 205)).

5.2.2.2.5. Refresh environment

Reloads and initializes the complete environment.

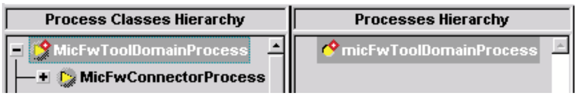
5.2.2.2.6. Browse new net... Ctrl+A

Opens a dialog for selecting a new root class.

5.2.2.2.7. Save all changes Ctrl+S

Saves all changes that have been made since the last save.

When changes are made in the DPB, the modified items are marked with a red circle. The changes that have not been saved can be discarded.



The red circles are removed after the changes have been saved.

5.2.2.2.7.1. Storage and Initialization

When saving changes, a class method named is generated for each modified Domain Process class, that contains all declarations made so far. For each Connection, a read accessor will be generated, and for the Base Connections an additional write accessor will be generated. This gives access to the models in the Connections.

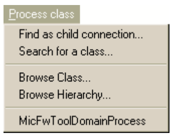
5.2.2.2.8. At runtime, all declarations will be stored in a class instance variable named . Discard all changes Ctrl+D

Discards any unsaved (ie, marked with red circles) changes.

5.2.2.2.9. Close

Closes the DPB.

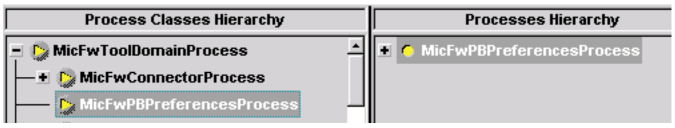
5.2.2.3. Submenu Process Class



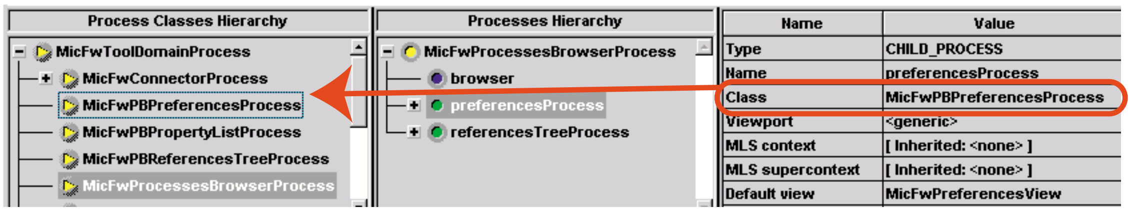
5.2.2.3.1. Find as child connection

Searches for a class in the **Process Classes Hierarchy** column that has a child connection to the class currently selected in the **Process Classes Hierarchy** column. If such a class exists, then the class become the selected class in the **Process Classes Hierarchy** column and the connection is selected in the **Processes Hierarchy** column.

For example, assume that the following class is selected:


Process Classes Hierarchy
MicFwToolDomainProcess
MicFwConnectorProcess
MicFwPBPreferencesProcess
Processes Hierarchy
MicFwPBPreferencesProcess

Selecting **Find as child connection** will locate the process class that has a child connection to the selected class:


Process Classes Hierarchy
MicFwToolDomainProcess
MicFwConnectorProcess
MicFwPBPreferencesProcess
MicFwPBPropertyListProcess
MicFwPBReferencesTreeProcess
MicFwProcessesBrowserProcess
Processes Hierarchy
MicFwProcessesBrowserProcess
browser
preferencesProcess
referencesTreeProcess

Name	Value
Type	CHILD_PROCESS
Name	preferencesProcess
Class	MicFwPBPreferencesProcess
Viewport	<generic>
MLS context	[Inherited: <none>]
MLS supercontext	[Inherited: <none>]
Default view	MicFwPreferencesView

5.2.2.3.2. Search for a class...

Opens a dialog for entering the name of the class to search for in the processes displayed in the Process Classes Hierarchy column.

5.2.2.3.3. Browse class...

Opens the VA Class Browser on the class selected in the Process Classes Hierarchy column.

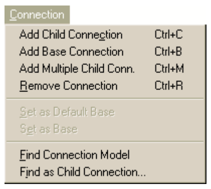
5.2.2.3.4. Browse class...

Opens the VA Hierarchy Browser on the class selected in the Process Classes Hierarchy column.

5.2.2.3.5. [list of classes]

Contains a list of all classes that have been found during this session with the **Search for a class...** menu item.

5.2.2.4. Submenu Connection


Connection
Add Child Connection Ctrl+C
Add Base Connection Ctrl+B
Add Multiple Child Conn. Ctrl+M
Remove Connection Ctrl+R
Set as Default Base
Set as Base
Find Connection Model
Find as Child Connection...

5.2.2.4.1. Add child connection Ctrl+C

Adds a child process connection for the Domain Process class currently selected in the column **Process Hierarchy**. The target class is selected from a list of all availabe Domain Process classes.

5.2.2.4.2. Add base connection Ctrl+B

Adds a base connection for the Domain Process class currently selected in the column **Process Hierarchy**. The target class is selected from a list of all availabe Domain Object classes.

5.2.2.4.3. Add Multiple Child Conn. Ctrl+M

Adds a multiple child connection for the Domain Process class currently selected in the column **Process Hierarchy**.

5.2.2.4.4. Set as default base

Available if a non-default base connection is selected in the column **Process Hierarchy**. Selecting causes the selected non-default base connection to become the default base connection.

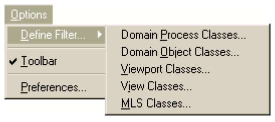
5.2.2.4.5. Find connection model

Selects in the **Process Classes Hierarchy** column the class that is referenced by the currently selected connection in the **Processes Hierarchy** column.

5.2.2.4.6. Find as child connection...

For the currently selected process connection: Finds other child connections to the same target class.

5.2.2.5. Submenu Options



5.2.2.5.1. Define filter...

Selecting a submenu brings up a dialog for selecting the filter for one of the following:

- Domain Process Classes...
- Domain Object Classes...
- Viewport Classes...
- View Classes...
- MLS Classes...

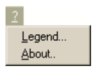
5.2.2.5.2. Toolbar

Displays/hides toolbar.

5.2.2.5.3. Preferences...

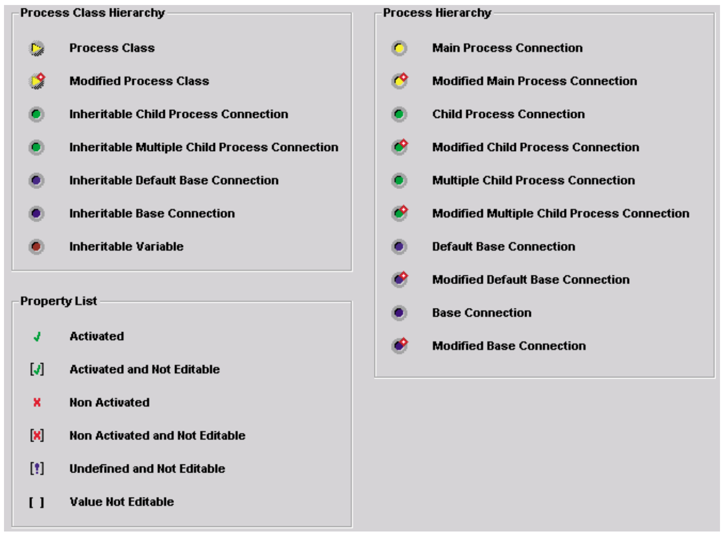
Opens the **Preferences** dialog (see ‘Preferences dialog’ (page 206)).

5.2.2.6. Submenu ?



5.2.2.6.1. Legend...

Displays the following description of the various symbols used in the DPB:



Process Class Hierarchy

- ● Process Class
- ● Modified Process Class
- ● Inheritable Child Process Connection
- ● Inheritable Multiple Child Process Connection
- ● Inheritable Default Base Connection
- ● Inheritable Base Connection
- ● Inheritable Variable

Property List

- ✓ Activated
- [✓] Activated and Not Editable
- × Non Activated
- [×] Non Activated and Not Editable
- [!] Undefined and Not Editable
- [ ] Value Not Editable

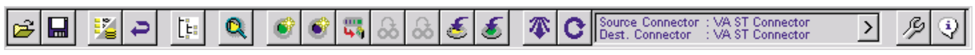
Process Hierarchy

- ● Main Process Connection
- ● Modified Main Process Connection
- ● Child Process Connection
- ● Modified Child Process Connection
- ● Multiple Child Process Connection
- ● Modified Multiple Child Process Connection
- ● Default Base Connection
- ● Modified Default Base Connection
- ● Base Connection
- ● Modified Base Connection

5.2.2.6.2. About..

Displays contact information for PMS Micado.

5.2.2.7. Toolbar



Source Connector : VA ST Connector
Dest. Connector : VA ST Connector

5.2.2.7.1. Load a session from disc

Same as **Browser / Open session....**

5.2.2.7.2. Save current session to disc

Same as **Browser / Save session**.

5.2.2.7.3. Save all changes via destination connector

Same as **Browser / Save all changes Ctrl+S**.

5.2.2.7.4. Discard all changes

Same as **Browser / Discard all changes Ctrl+D**.

5.2.2.7.5. Browse new object net

Same as **Browser / Browse new Net... Ctrl+A**.

5.2.2.7.6. Search for a process class in object net

Same as **Process class / Search for a class...**.

5.2.2.7.7. Add a new child process connection

Same as **Connection / Add Child Connection Ctrl+C**.

5.2.2.7.8. Add a new base connection

Same as **Connection / Add Base Connection Ctrl+B**.

5.2.2.7.9. Remove selected connection

Same as **Connection / Remove Connection Ctrl+R**.

5.2.2.7.10. Make selected base to default base

Same as **Connection / Set as default base**.

5.2.2.7.11. Make selected default base to base

Same as **Connection / Set as base**.

5.2.2.7.12. Find process class of selected model

Same as **Process class / Search for a class**.

5.2.2.7.13. Find child process connections with selected model

Same as **Connection / Find as child connection**.

5.2.2.7.14. Choose a new environment connector

Same as **Browser / Choose Environment... Ctrl+E**.

5.2.2.7.15. Reload all classes from current environment

Same as **Browser / Refresh Environment**

5.2.2.7.16. Connection manager settings

```
Source Connector  : VA ST Connector
Dest. Connector   : VA ST Connector
```

Click to display the current settings in the Connection Manager.

5.2.2.7.17. Open preferences window

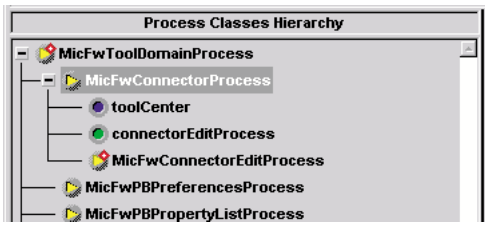
Same as **Options / Preferences...**.

5.2.2.7.18. About PMS Micado

Same as **? / About...**.

5.2.2.8. Process Classes Hierarchy column

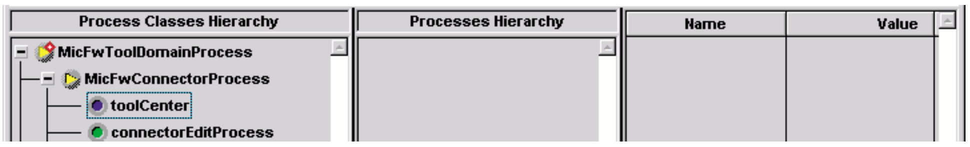
5.2.2.8.1. Domain processes in the Process Hierarchy column


Process Classes Hierarchy
MicFwToolDomainProcess
MicFwConnectorProcess
toolCenter
connectorEditProcess
MicFwConnectorEditProcess
MicFwPBPreferencesProcess
MicFwPBPropertyListProcess

The domain process class that was opened in the DPB and all other domain process classes that belong to that class's tree are displayed in the Process Classes Hierarchy column.

5.2.2.8.2. Connections and variables in the Process Hierarchy column

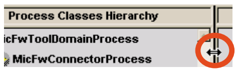
The column also displays connections and variables. However, this is for information only. Selecting a connection will result in empty Process Hierarchy, Name and Value columns.



Note also that only the immediate connections for a class are displayed (subconnections for a connection are not inherited).

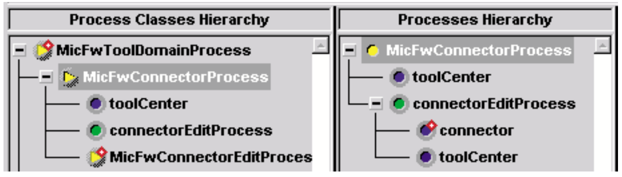
5.2.2.8.3. Resizing the Process Hierarchy column

The column can be resized:



5.2.2.9. Processes Hierarchy column

The connections and any subconnections for the Domain Process class selected in the Process Classes Hierarch column are displayed in the Processes Hierarchy column, as shown in the following diagram:



5.2.2.10. Name/Value columns

The Name/Value columns displays the **name** and **value** of **properties** for the selection in the Processes Hierarchy column.

5.2.2.10.1. Non-transaction properties displayed in column Name

The non-transaction properties displayed in the Name column depend on the type of connection selected in the Processes Hierarchy column:

PROPERTY	MAIN_ PROCESS	CHILD_ PROCESS	MULTIPLE_ CHILD_ PROCESS	DEFAULT_ BASE	BASE
Type	X	X	X	X	X
Name	X	X	X	X	X
Prefix			X		
Class	X	X	X	X	X

PROPERTY	MAIN_ PROCESS	CHILD_ PROCESS	MULTIPLE_ CHILD_ PROCESS	DEFAULT_ BASE	BASE
Viewport	X	X	X	X	X
Base class			X		
MLS context	X	X	X	X	X
MLS supercontext	X	X	X	X	X
Default view	X	X	X		
Max. instances			X		

5.2.2.10.2. Transaction properties displayed in column Name

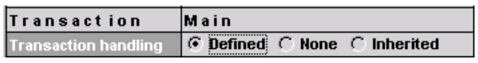
Transaction properties can be defined for the following types of transactions:

- **Transaction Main**
- **Transaction Connection**
- **Transaction Child**

The transaction properties are only displayed if they are defined. Transaction properties can only be defined as shown in the table below::

PROPERTY	MAIN_ PROCESS	CHILD_ PROCESS	DEFAULT_ BASE	BASE
Transaction Main properties	If defined			
Transaction connection properties		If defined		
Transaction child properties	If defined	If defined		

Transaction properties are defined by setting the property **Transaction handling** to **defined**:



Transaction	Main
Transaction handling	[✓ Defined] None Inherited

Available transaction properties are the same for Main, Connection, and Child. The available properties are show in the following table:

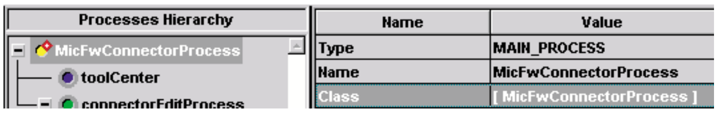
TRANSACTION PROPERTY (if Transaction handling = Defined)
Autostart mode
Use own context
Use parent context
Isolate mode
Persistence context
Transaction context
Hierarchical mode

5.2.2.10.3. Property descriptions

5.2.2.10.3.1. Class

Within the framework, type checking is implemented ensuring that only Domain Models of the specified Domain Model class, its subclasses, or may be connected to a Base Connection during runtime.

The Domain Process itself is attached to the Default Process Connection, which cannot be changed or removed, neither by the browser nor by API methods.



Processes Hierarchy

- MicFwConnectorProcess
  - toolCenter
  - connectorEditProcess

Name	Value
Type	MAIN_PROCESS
Name	MicFwConnectorProcess
Class	[MicFwConnectorProcess]

The default Connection receives the class name of the Domain Process as name by default. This name is sufficient if you don’t want to attach platform part like entry fields or lists to this Process Connection.

In order to be able to access this Connection explicitly by name, it has to be renamed like any other base or Process Connection declared. Because a Connection Name beginning with an upper case letter are interpreted as a joker by the framework and replaced using (e.g. for entry fields and lists) or (e.g. for Command Controls) Connection.

5.2.2.10.3.2. Viewport

Every connection defines its own Viewport. The Framework uses this information to start the search for the right `portName` which returns the class name of the model instance that will write to the Connection at runtime. If no Viewport of this hierarchy answers the right model class, then the Viewport class that is defined at the Connection will be taken.

5.2.2.10.3.3. MLS context / supercontext

To activate MLS for your application you have to set an MLS Context at your Main Process at least. This could be done within the property list.

5.2.2.10.3.4. Defining Transactional Behavior

Domain processes are the ideal place for transaction control. Thus, the class is able to maintain a Transaction Context, and its API supports all of the messages necessary to manipulate the context.

5.2.2.10.3.5. Transaction Attributes

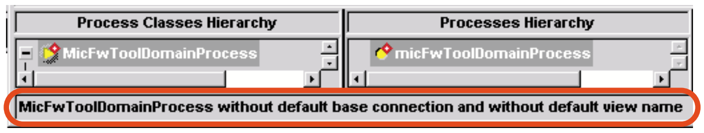
You can influence the transactional behavior of a Domain Process by defining the following items:

- Transaction Handling can be set to , or .
- Isolate mode means that a transaction won’t be affected by changes caused by concurrent transactions.
- Begin transaction means that a new transaction level will be started when the Connector is opened. Attention if you set this feature to false!
- You have to ensure the transaction behavior by your own. This is only applicable for special cases, where the provided behavior is not sufficient.
- Transaction context / Persistence context indicates which type of transaction context will be used (image / persistence)
- **Use own context** / **Use parent context** indicates which transaction context will be used (concurrent or nested).

Note that when using a persistence context, a method answering the Persistence Manager must be implemented by the Domain Process.

5.2.2.11. Status bar

The status bar displays information depending on what has been selected in the DPB dialog:



Process Classes Hierarchy: MicFwToolDomainProcess

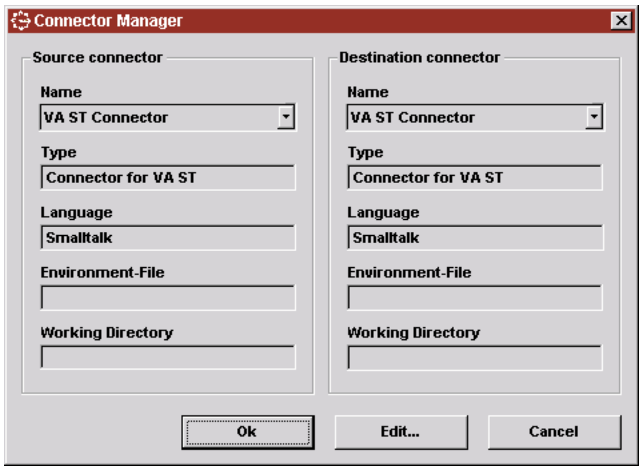
Processes Hierarchy: micFwToolDomainProcess

MicFwToolDomainProcess without default base connection and without default view name

5.2.3. Connector manager dialog

The Connector Manager dialog is opened from the main menu by selecting **Browser / Choose environ-**

ment:



5.2.3.1. Source / Destination connectors

From the drop-down list: Choose the source and destination connectors.

5.2.3.2. Connector properties

5.2.3.2.1. Name

Name of the connector.

5.2.3.2.2. Type

Type of connector (for example: **Connector for Smalltalk**).

5.2.3.2.3. Language

Programming language of the connector (for example: Smalltalk).

5.2.3.2.4. Environment-File

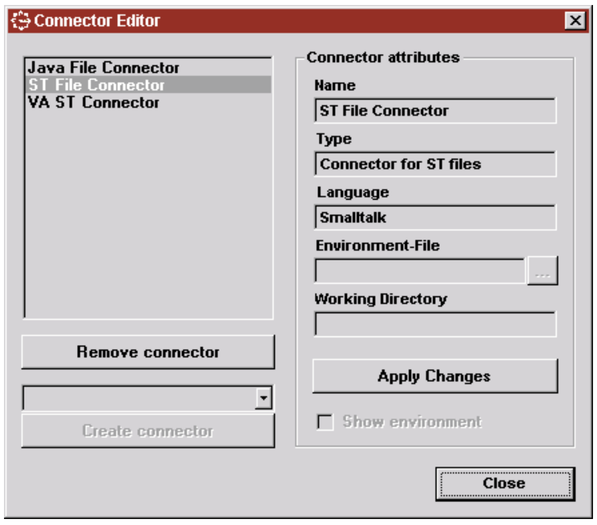
Name of the environment file.

5.2.3.2.5. Working directory

Name of working directory (normally vast).

5.2.3.3. Modifying connector properties

To modify the connector properties: In the **Connector Manager** dialog: Click on **Edit...**. The **Connector Editor** dialog appears:



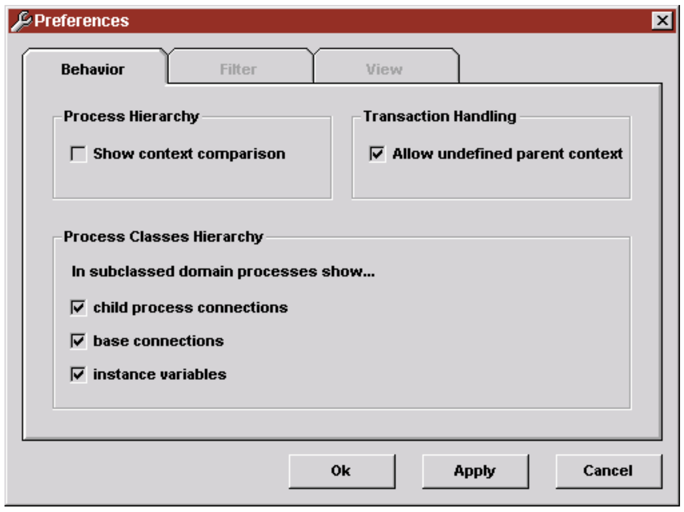
Click **Apply Changes** to apply any changes (simply clicking **Close** will not apply the changes).

5.2.4. Preferences dialog

To open the preferences dialog: From the main menu: Select **Options / Preferences...**.

5.2.4.1. Tab Behavior

Tab behavior settings define general behavior.



Preferences

Behavior | Filter | View

Process Hierarchy
- [ ] Show context comparison

Transaction Handling
- [✓] Allow undefined parent context

Process Classes Hierarchy

In subclassed domain processes show...
- [✓] child process connections
- [✓] base connections
- [✓] instance variables

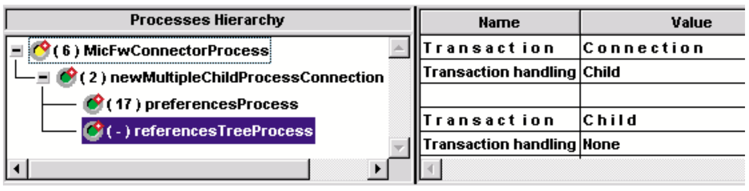
5.2.4.1.1. Processes Hierarchy settings

5.2.4.1.1.1. Show context comparison

If checked: Each connection will be marked with a context number. The context number is unique for each context. The set of contexts numbers are consecutive integers starting with 1.

Example

In the following diagram: For the connecttion referencesTreeProcess: No Transaction connection or Transaction child is defined. Therefore, a separate context does not exist, and the context is marked with a hypen ("-").



Processes Hierarchy
- (6) MicFwConnectorProcess
  - (2) newMultipleChildProcessConnection
    - (17) preferencesProcess
    - (-) referencesTreeProcess

Name	Value
Transaction	Connection
Transaction handling	Child
Transaction	Child
Transaction handling	None

5.2.4.1.2. Process Classes Hierarchy settings

5.2.4.1.2.1. child process connections

If checked: Child process connections will be shown in the Process Classes Hierarchy column.

5.2.4.1.2.2. base connections

If checked: Base connections will be shown in the Process Classes Hierarchy column.

5.2.4.1.2.3. instance variables

If checked: Instance variables will be shown in the Process Classes Hierarchy column.

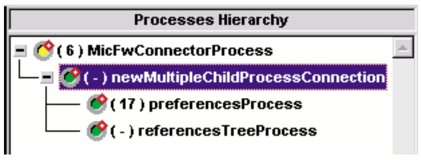
5.2.4.1.3. Transaction handling

5.2.4.1.3.1. Allow undefined parent context

If checked: A child connection can have a defined context without the parent connection having a defined context.

Example

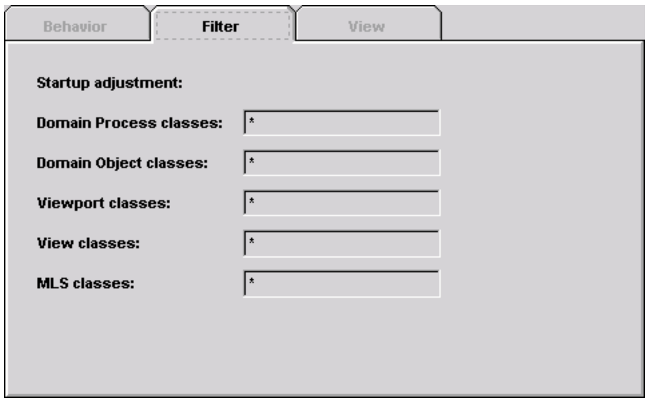
In the following diagram: preferencesProcess has a defined context, although it is a child connection whose parent has no defined context.



Processes Hierarchy
- (6) MicFwConnectorProcess
  - (-) newMultipleChildProcessConnection
    - (17) preferencesProcess
    - (-) referencesTreeProcess

5.2.4.2. Tab Filter

Tab Filter settings define the startup adjustments for the DPB. Startup adjustments define the options that are available when adding connections, viewports, default views, and MLS classes. Startup adjustments only take effect after the DPB has been restarted.

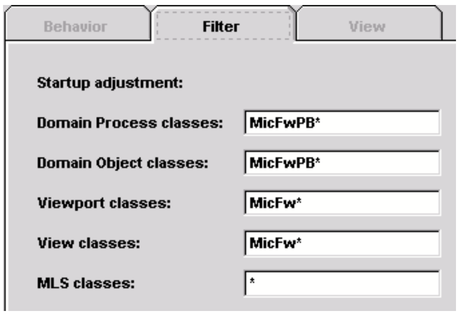


Startup adjustment:

Domain Process classes:	*
Domain Object classes:	*
Viewport classes:	*
View classes:	*
MLS classes:	*

5.2.4.2.1. Startup adjustments

The following diagram shows example startup settings.



Startup adjustment:

Domain Process classes:	MicFwPB*
Domain Object classes:	MicFwPB*
Viewport classes:	MicFw*
View classes:	MicFw*
MLS classes:	*

5.2.4.2.1.1. Domain Process classes:

Specifies the subset of Domain Process classes available for selection.

5.2.4.2.1.2. Domain Object classes:

Specifies the subset of Domain Object classes available for selection.

5.2.4.2.1.3. Viewport classes:

Specifies the subset of Viewport classes available for selection.

5.2.4.2.1.4. View classes:

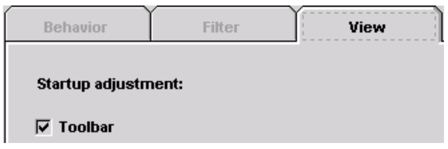
Specifies the subset of Default View classes available for selection.

5.2.4.2.1.5. MLS classes:

Specifies the subset of MLS classes available for selection.

5.2.4.3. Tab View

Tab View settings define the startup adjustments for the the toolbar.



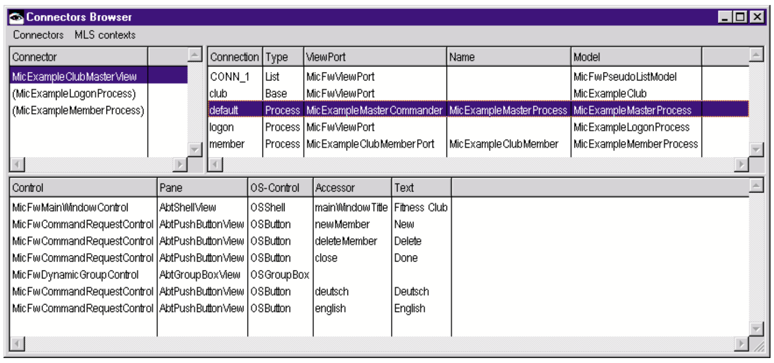
Startup adjustment:

☑ Toolbar

5.3. View Connectors Browser

The View Connectors Browser grabs information from all the elements surrounding the Model View Connectors: Connections, Viewports, Domain Objects and Domain Processes on the one hand, and Views and Controls on the other.

The View Connectors Browser can be used to display information on important kinds of instances created by the application at runtime - Connectors, Connections, Viewports etc. Only opened Connectors which have corresponding opened views are displayed.



Connector
MicExampleClubMasterView
(MicExampleLogonProcess)
(MicExampleMemberProcess)

Connection	Type	ViewPort	Name	Model
CONN_1	List	MicFwViewPort		MicFwPseudoListModel
club	Base	MicFwViewPort		MicExampleClub
default	Process	MicExampleMasterCommander	MicExampleMasterProcess	MicExampleMasterProcess
logon	Process	MicFwViewPort		MicExampleLogonProcess
member	Process	MicExampleClubMemberPort	MicExampleClubMember	MicExampleMemberProcess

Control	Pane	OS-Control	Accessor	Text
MicFwMainWindowControl	AbtShellView	OSShell	mainWindowTitle	Fitness Club
MicFwCommandRequestControl	AbtPushButtonView	OSButton	newMember	New
MicFwCommandRequestControl	AbtPushButtonView	OSButton	deleteMember	Delete
MicFwCommandRequestControl	AbtPushButtonView	OSButton	close	Done
MicFwDynamicGroupControl	AbtGroupBoxView	OSGroupBox		
MicFwCommandRequestControl	AbtPushButtonView	OSButton	deutsch	Deutsch
MicFwCommandRequestControl	AbtPushButtonView	OSButton	english	English

The browser is best inspected from left to right, first the top part and then the lower part for Control details. Starting with a Connector selected from the top left List, its Connections and attached Viewports and models appear in the top right. Unnamed Connectors are displayed by the names of their view classes or – if there is no view opened for the Connector – displayed in brackets.

Connections are displayed by their assigned names. The Default Process Connection and all other unnamed Connections inherit the (class) names of their models.

Moreover, the following pieces of information will be shown for each Connection:

- **Type:** Base, Process, List or Choice
- **ViewPort:**class name of Viewport
- **Name:** class name of the model, which is associated with the Viewport
- **Model:** Domain Model or Domain Process (class name)

Properties of the Controls appearing on the view are shown in the lower part of the browser. Each Control realized on the view appears and is displayed by its Abstract Control type.

Moreover, the following information will be displayed for each Control:

- **Pane:** real Control (class name)
- **OS-Control:** the operating system control, which is used by the real Control (class name)
- **Accessor:** the name of the real Control, if it's used as a selector within the framework
- **Text:** textual representation of the real Control

To get more details you can double click at any row to get an ordinary inspector of it.

5.4. Framework Logger

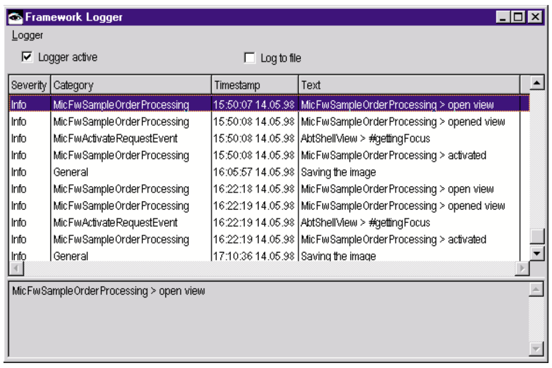
The Framework Logger is more of a general purpose tool, collecting and displaying diagnostic messages generated by one of the frameworks, or even by the application itself. Because of the event-driven nature of the Application Framework it is sometimes difficult to debug code by setting breakpoints and so it is more convenient to log state information to the logger for later inspection.

5.4.1. Operational Area

The Framework Logger is a general purpose tool used to display diagnostic messages from a running application. It is very useful in conjunction to the Application Framework if you rely on a defined sequence of messages and message-flow when a Debugger would interrupt or change those sequences (e.g. popping up the Debugger will generate a Losing Focus Event again causing some operation within the framework).

The Application Framework contains logging facilities on many important areas (e.g. I/O between models and parts, opening and closing of views), providing useful diagnostic information to the logger (if activated by the user).

Information in the logger is presented line by line in an independent window as shown below:



Severity	Category	Timestamp	Text
Info	MicFwSampleOrderProcessing	15:50:07 14.05.98	MicFwSampleOrderProcessing > open view
Info	MicFwSampleOrderProcessing	15:50:08 14.05.98	MicFwSampleOrderProcessing > opened view
Info	MicFwActivateRequestEvent	15:50:08 14.05.98	AbtShellView > #gettingFocus
Info	MicFwSampleOrderProcessing	15:50:08 14.05.98	MicFwSampleOrderProcessing > activated
Info	General	16:05:57 14.05.98	Saving the image
Info	MicFwSampleOrderProcessing	16:22:18 14.05.98	MicFwSampleOrderProcessing > open view
Info	MicFwSampleOrderProcessing	16:22:19 14.05.98	MicFwSampleOrderProcessing > opened view
Info	MicFwActivateRequestEvent	16:22:19 14.05.98	AbtShellView > #gettingFocus
Info	MicFwSampleOrderProcessing	16:22:19 14.05.98	MicFwSampleOrderProcessing > activated
Info	General	17:10:36 14.05.98	Saving the image

MicFwSampleOrderProcessing > open view

Severities are used to give hints on the impacts of a displayed message:

- **Primitive:** for monitoring low level Application Framework activities
- **Info:** for monitoring purposes
- **Warning:** an unexpected situation or result has been encountered
- **Error:** an error has occurred
- **FatalError**: a fatal error has occurred

An arbitrary number of categories may be used for classification - see next section Protocol Protocol. The display of messages can be filtered by activating or deactivating severities and categories (done from the Menu).

One special category is   which collects information on missing accessor methods (accesses relying on the   ) or similar things.

It is recommended that you turn this category off until performance tuning becomes an issue.

5.4.2. Protocol

Every object in the system understands the following message:

```
Object>>log:severity:withString:
```

The arguments are:

- **category** (arbitrary symbol)
- **severity** (one of the symbols described in section Operational Area Operational Area)
- **text** (string to be displayed)

The short form

```
self log: aString
```

is equivalent to

```
self log: self class name asSymbol severity: #Info withString: aString
```

To avoid creating huge log messages being ignored by an inactive logger, another protocol is available to support a trace block evaluated later:

```
Object >> micTraceBlock: aZeroArgumentBlock
```

The passed block will be evaluated only if the logger is active. For example:

```
self micTraceBlock: [
self log: string1, string2, string3, string4, string5.].
```

5.5. Online Debugger

If you want to debug a Framework problem, e.g. a button doesn't get disabled, even though it should be, you also have to know about internal structures built up by the Framework. In order to get these additional information about the model you should use the Online Debugger. It will help you to inspect the Framework structure while your application is running and allows you to inspect the usually invisible information.

When working with Application Framework, one designs Domain Objects and Domain Processes and connects them with the help of DomainProcessesBrowser. Besides these user defined structures, Application Framework builds up internal structures and provides additional information about the model.

If you want to debug a Framework problem, e.g. a button doesn't get disabled although it should, you have to know which process this button is currently connected to and what the state of the enabled method is. Because of the dynamic of the Framework, this can be a rather complex procedure. The Online Debugger helps you to inspect the Framework structure while your application is running and allows you to inspect otherwise invisible information.

To start the online debugger select from the Transcript menu **micFrameworks -> Open online-debugger.** In the online-debugger's main menu select **'Hover manager on'**. Go with the mouse cursor over the witget that you interested and press the scroll-button on the keyport. After the blink of the cursor press the scroll-button. Now you can inspect all interested things ot these widget.

The Online Debugger always works on a single widget and collects all important information available for it. The tool has 7 notebook pages, which may contain or may not contain information, depending on the widget you inspect. All non List widgets have information on the first three pages.

5.5.1. Online debugger pages

5.5.1.1. Page "Basic":

The basic page contains information about the Domain Object and its Viewport, the widget is connected to.

- ViewPort accessor name:
- The Aspect name the Framework uses to get the content of the widget or the method which is called for a button. This name is defined by the widget accessor name entered inside the composition editor.
- You can inspect the accessor Object used by the Framework.
- ViewPort:
- The Viewport which is used for the model this widget is connected to.
- You can inspect the Viewport class and it currently used instance.
- Model:
- The Domain Object or process the widget is connected to.
- Domain Processes are displayed hierarchical and shows their parent child relations, as they exists at this point of time.
- You can inspect the model instance. If the model is a Domain Process, you can open the Domain Processes Browser on it. If the process has a transaction, you can browse the Transaction Context instance.
- Connector:
- The Connector of the model, this widget is connected to.
- The List contains all process and Base Connections. The default process and Default Base Connection are always the first Connections of its group.
- The arrow displays the Connection, of the model the widget is connected to.
- You can inspect the selected model Connection object.
- Type:
- If the widget is an entry field, and you have set a type in the object net browser, you will see it displayed here.
- You can change the type in the object net browser if you like.

5.5.1.2. Page "Basic Vp"

You get a table of all the objects which were touched by the Framework while retrieving the result for the basic Aspect or its Subaspects.

If you want to understand why some update mechanism didn't work, you should first check, if the Framework registered all your dependent models.

You can inspect the selected Domain Object and browse the object net of its class.

The two Lists "active" and "inactive" display the user implemented Subaspect methods, which are currently used by the framework. If a Subaspect is marked inactive, a method written by the user is not used, even it might exist. In such case, you should reset all Framework Caches.

At the bottom of the view, you can inspect the Transaction Context which is used.

5.5.1.3. Page "MLS"

The table displays the Locale Entities from the MLS system, which may be used to localize data. The Debugger displays the data in all Locales which are available.

This page only contains data if the MLS system is currently loaded into your image.

5.5.1.4. Page "List contents"

This page contains basically all the information of the "Basic" page, but for the content Aspect of a List widget. Mainly the page shows the model and its Viewport, which are used to get the content of a List.

5.5.1.5. Page "List Vp"

The page contains a table, as it exists on the "Basic Vp" page, only this time the table contains the models which were touched by the Framework while retrieving the content of the List or its Subaspects.

5.5.1.6. Page "List items"

This page displays all the list items and offers additional information about them.

Each Framework List contains Viewport objects and because of this, the tool allows you to inspect the Viewport instance, the Viewport class and the model belonging to each Viewport.

When working with Container Details, you can choose the attribute to be displayed in the main item list from the "Available attributes" list. After you have selected an item, you can choose an Aspect in the lower Drop-down List, which is used either to get the String which appears as list entry or to handle the children of an object inside a hierarchy list.

You can get the information about all touched models for each of these list item Aspects in the table called "Models touched while touching selected item".

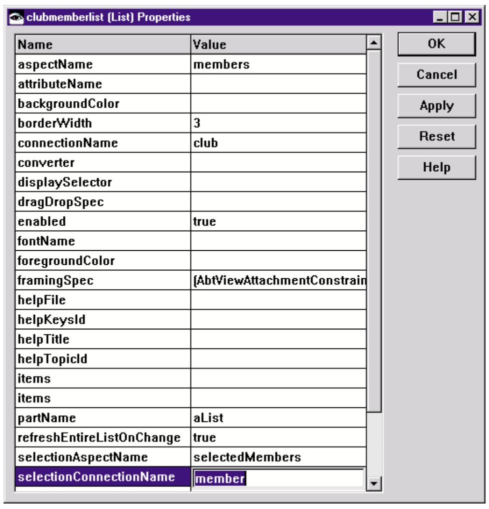
5.6. VA Composition Editor

GUI Controls can be connected to Application Frameworks-based models by using a specific naming convention. We recommend this method especially for complex or dynamic views.

The VisualAge Composition Editor provides two additional hooks for connecting parts to the model world: One is to use the settings table and with the help of the other feature you can use VisualAge Connections to connect to features of models and Viewports. The last feature is not completely supported by now, but very helpful when you begin to design a GUI.

5.6.1. Connect to Visual Parts with Part Settings

The Composition Editor has been extended to provide another means of specifying Aspect Connections. In the 'Settings' table for GUI Controls, there is now some table rows for entering this information.



Name	Value
aspectName	members
attributeName	
backgroundColor	
borderWidth	3
connectionName	club
converter	
displaySelector	
dragDropSpec	
enabled	true
fontName	
foregroundColor	
framingSpec	(AbtViewAttachmentConstrain
helpFile	
helpKeysId	
helpTitle	
helpTopicId	
items	
items	
partName	aList
refreshEntireListOnChange	true
selectionAspectName	selectedMembers
selectionConnectionName	member

The Connection settings has been added to the table view for manipulating the settings of a part.

5.6.1.1. Content/Command Settings

Command Control and Controls, which get any sort of content, are connected to a model by using the same two settings attribute names:

- The is like a path to the corresponding model.
- The is an Action or an Attribute at this model.
- The has the same task as the 1st name part of the name convention. The corresponds to the 2nd name part.

5.6.1.2. List Selection Settings

With List and Choice Controls (e.g. Spin Button, Radio Button Set, IconTree, Container Details...) you can connect the selection to a model:

- The is like a path to the corresponding model and
- The is an Action or an Attribute at this model.
- The has the same task as the 3rd name part of the name convention of a List Control. The corresponds to the 4th name part of the name convention of a List Control.

In addition to this you want to set the display method of the list items, for that you can use the , which corresponds to the 5th name part of the name convention of a List Control. This additional setting is the selector used for obtaining a printable representation of the objects displayed in the List. The default value (used if nothing is entered in this field) is .

5.6.1.3. Additional Text Aspect Settings

With Controls like Combo Box or Spin Button you can connect the text editor part to a model:

- The is like a path to the corresponding model and
- The is an Action or an Attribute at this model.
- The has the same task as the 3rd name part of the name convention of a List Control.

The corresponds to the 4th name part of the name convention of a List Control.

5.6.1.4. Additional Group Aspect Settings

For grouping Controls (Group Box, Notebook Page, Form...) you can connect set two additional settings:

- The , which replace the Default Base Connection for child parts of the group control where the Connection setting is not set.
- The replace the Connector of itself and its child controls by the Connector of a child process of the origin Connector.
- To do this you set the child Process Connection name here.

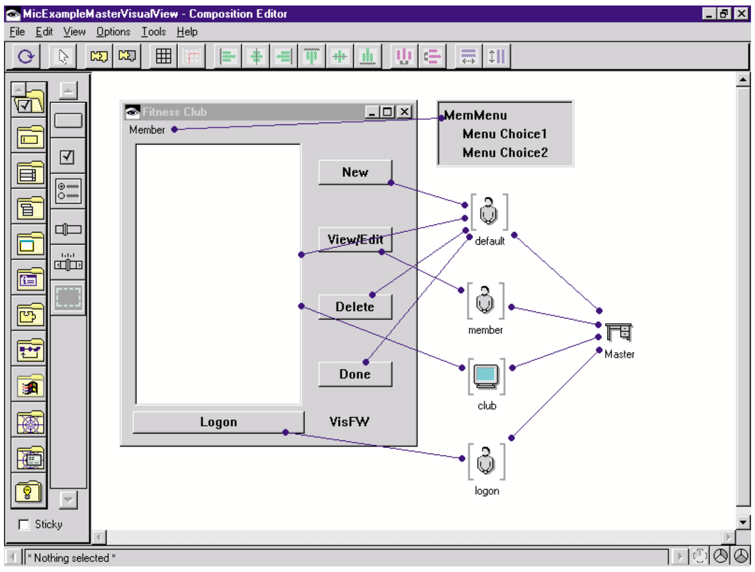
The Drop-down List labeled ‘Connection’ allows you to select a Connection from all the available Connections in the context of the part (i.e. from those that can be reached from the Domain Process associated with the domain manager part on the canvas). The Drop-down List labeled ‘Aspect’ allows you to choose from the Aspects of the selected Connection. The same rules apply to the selection Connection settings.

For List Controls (i.e. Lists, Multiple Select Lists, Drop-down Lists etc.) the settings page contains one more field.

5.6.2. Connect to models using Non-Visual Parts

To further facilitate the process of connecting views and models as well as to provide a method of visualizing structural elements of an application, the Frameworks also extend the VisualAge Composition Editor.

Let’s first look at an example of what this means in practice. The example is based on the Fitness Club application that is included as a coded example. After that, the extended capabilities of the Composition Editor will be explained in detail.



shows how the extended Composition Editor helps to visualize important Aspects of an application. These are :

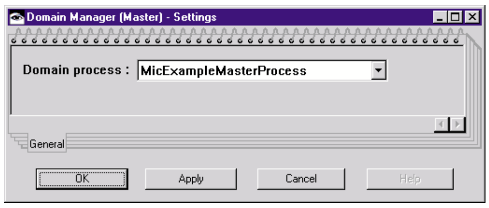
- : the icon labeled ‘ represents a Domain Process (in this case)
- the icons labeled ‘ , ‘ , ‘ and ‘ represent Viewports

Process hierarchy : the wires between the Viewports of Domain Processes and the master process ( ) indicate a child process relationship (e.g. ‘logon’ is a child process of the master process)

- the links between Viewports and GUI controls indicate a link between an Aspect of the respective Viewport and the GUI element. For example, the link from the ‘Logon’ button to the ‘ means that the button is linked to the Aspect of that (functionally equivalent to naming it ‘)

5.6.2.1. Domain Processes

Domain Processes can be used as non-visual parts in the Composition Editor. To do so, select the section ‘micado Frameworks’ section in the parts catalog on the far left of the Composition Editor. Then select the symbol (Domain Manager) and place it on the canvas. Double-clicking on it will open the settings dialog, which simply consists of a Listbox allowing you to select a Domain Process from those currently defined in your image:

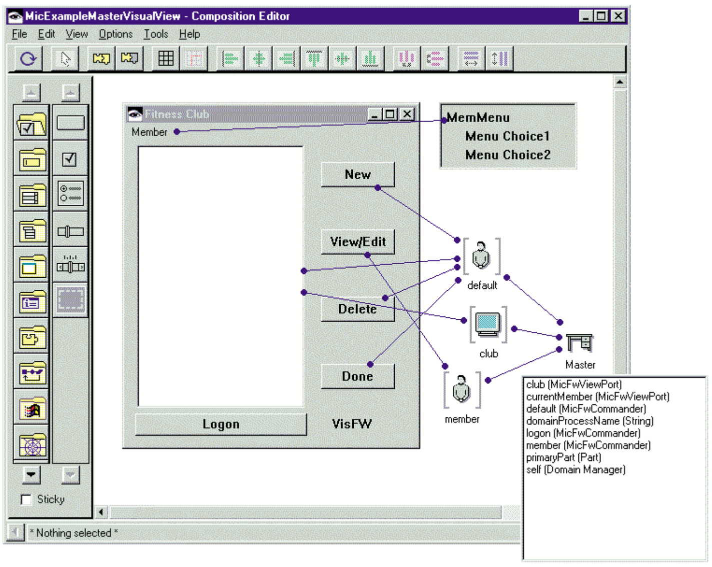


Domain Manager (Master) - Settings

Domain process : MicExampleMasterProcess

5.6.2.2. Viewports

The s associated with a Domain Process (i.e. the child processes and the involved Base Connections defined in the DomainProcessesBrowser) are accessible as tear-off attributes of the domain manager:



MicExampleMasterVisualView - Composition Editor

club (MicFwViewPort)
currentMember (MicFwViewPort)
default (MicFwCommander)
domainProcessName (String)
logon (MicFwCommander)
member (MicFwCommander)
primaryPart (Part)
self (Domain Manager)

* Nothing selected *

5.6.2.3. Aspect Connections

The Aspects of s are represented as connectable features of the corresponding edit parts. Depending on their type, the GUI parts also have one or two extra features. The first, which is called 'micContents' represents the contents of the part. The other, named 'micSelection', represents the selection of the part. In the example in the above figure, the 'members' Aspect of the 'club' is connected to the 'micContents' feature of the Listbox, indicating that the Listbox will be filled with the members of the club. A detailed description of a complete example will follow in the next chapter.

6

Customizing

6.1. Overview

This chapter describes the possibilities for changing some of the Application Framework's default behavior or enlarging its functionality in some way.

The Framework consists of several subsystems and major components that are interacting through their public interface. To be independent from their specific implementation, especially class names or inheritance structure, they are coupled as loosely as possible. Each plugable component must register itself with centralized mapping objects, responsible for maintaining the path from a requester to the appropriate service provider.

Application programmers have the opportunity to substitute entire subsystems with their own implementation if this is required. This may be the case when a well known, complete and mature subsystem already exists in your company and you want to reuse it with your new application. The amount of effort it takes to accomplish such modifications depends widely on the complexity of the particular component. Let's look at some examples.

Assume you have created a new composite visual Control with a specific behavior and a public interface. All the details of managing the embedded Controls are hidden and it's prepared for reuse in the entire application (or even more). The task now is to introduce your Control to the Framework. This is one of the simple things to do and requires the choice of a set of (existing) abstract components from the abstract Model View Controller and the implementation of one Adapter class with about 5 - 10 methods transforming the abstract protocol to your real Control's interface. This task is explained in detail in one of the following sections.

Externalization is a pattern to allow easy access to and influence the behavior of specific application components. The Framework offers an interface to define authorization and validation of all Domain Objects based on externally maintained rules. These rules may be implemented by Smalltalk blocks or messages or even complex objects that you implemented as a part of a security system. They just have to be compatible with a very narrow and well defined interface which is explained in the chapters covering authorization and validation.

It will take more effort to substitute an entire subsystem. The Framework uses specialized Mappers to let the developer set and get interface classes that are used to fulfill a requested responsibility. Under certain circumstances you'll find a simplified default behavior which doesn't meet your requirements. One of the Framework's default assumptions is that s are uniquely related to their model objects (Domain Objects and Domain Processes). In other words, the which is selected for presentation of a Domain Object is chosen without any context or state of your application. The following sections show how Brokers are used to implement this simple relation and how you can develop and introduce your own Broker in order to choose dynamically the correct Viewport for a given Domain Object.

The entire Application Framework has been developed to offer as many reusable components and as much reusable behavior as possible. But, on the other hand, the framework uses several mechanisms to let developers plug in new or different subsystems to be as flexible as necessary. Beyond this manual, as an experienced Smalltalk programmer you'll find these hooks by browsing the code, looking for micAPI categorized methods and exploring the examples. You are provided with the entire source code, so you'll always find a way to create performance critical pieces or components that are difficult to implement in a generic manner without being dependent on the support of the Application Framework.

6.2. Internal Design

6.2.1. Abstract Model View Controller

The Application Framework implements an enhanced and refined Model View Controller (MVC) approach which provides for strict separation of the various aspects of application development.

6.2.1.1. Architecture

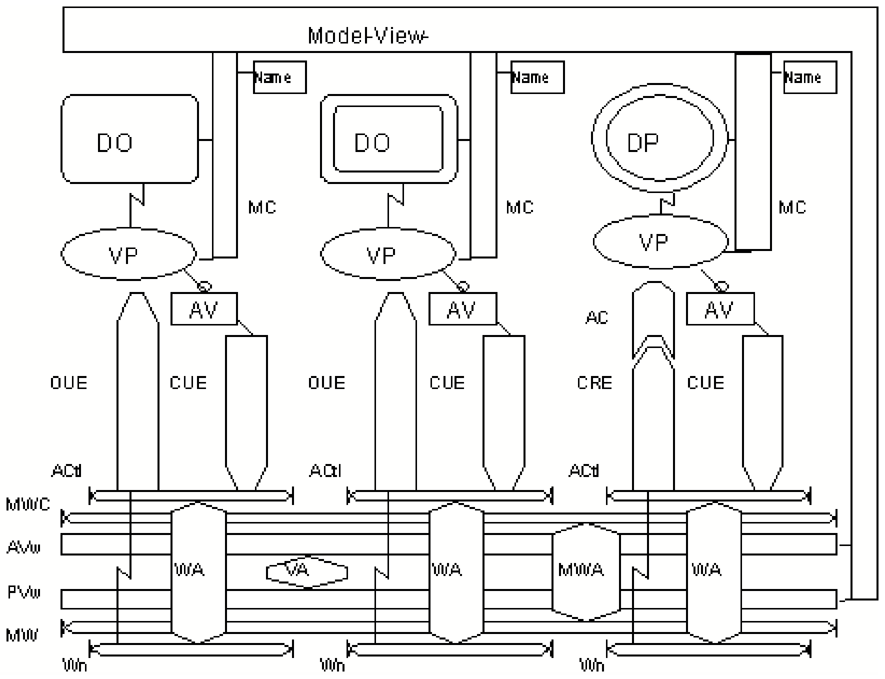
On the one hand there is a separation between the problem domain and the technical requirements of the window system like object representation and user interaction. The problem domain is reflected by the model objects (implemented as subclasses of   for objects managing Domain Processes and subclasses of   for actors and entities in the process). Viewports (instances of a subclass of   ) serve as interfaces for the technical requirements of object presentation in the view and transmission of user requests to the corresponding models. With respect to the window system (view and controller side of the MVC), Viewports communicate with Abstract Controls via Abstract Events.

On the other hand the window system (the view-controller-layer) is divided into an abstract platform-independent side and the Platform View. The platform-independent Abstract Window System is formed by the Abstract View (instance of   ), which holds the Abstract Controls (instances of subclasses of   ), and the Abstract Events (instances of subclasses of   ) which communicate with the Viewports. The technical interfaces to native windows and Controls in the "host" Smalltalk or Java environment are completely encapsulated in a set of classes which form the Platform Adapter. Communication between models, Viewports and Connectors on the one side and the Platform Windows and Controls (via their corresponding Adapters) on the other side, is accomplished in an unique way by the Abstract Window System, and is equivalent on all platforms.

The main functions of the Abstract Window System layers are:

- Abstract Controls:
- Shadowing real (native) controls. The variety of details brought in by real Controls is reduced considerably by a smaller hierarchy of Abstract Controls. Real controls with similar functionality in an abstract sense are mapped to the same Abstract Control class (e.g. single Aspect Controls, Choice Controls, List Controls, ...).
- Abstract Events:
- The Abstract Events manage the communication between the window system and model world in two directions:
- Real Events triggered by the Platform Windows are transformed into abstract instantiated event objects owned by the respective control which generates them. Different real events may be mapped to a single Abstract Event (class) if the nature of the event with respect to the model is the same. On the other hand, it is possible for the same real event in different real Controls to be mapped onto different Abstract Events.
- In the other direction, model changes caused by write access to instance variables or by command execution trigger model events. These events are propagated by Abstract Events resulting in the update of shadowed real Control.
- Abstract Commands:
- Certain events are targeted at actions in Domain Processes. Typically, actions like pressing Push Buttons, selecting Menu items or toolbar buttons as well as double-clicking on List Controls are mapped into Abstract Events of a special type (Command Request Events) which hold and activate Abstract

Commands to propagate the request to the corresponding Domain Process Viewport.



6.2.1.1.1. Models

- **DO: Domain Object** (instance of a subclass, acts in a Domain Process)
- **DP: Domain Process** (instance of a subclass, controls a Domain Process)

6.2.1.1.2. Delegation Models

- **VP: Viewport** (instance of a subclass, transforms Domain Object Aspects for presentation and passes commands to the model)

6.2.1.1.3. Model-View-Connector and Connections

- **Model-View-Connector** (instance of MicFwModelViewConnector, performs routing between models and view controls)
- **MC: Model Connection** (instance of , plugged into the Connector, connects models and corresponding delegation models)

6.2.1.1.4. Internal Data Transfer

- **AV: Abstract Value** (instance of , packs Model Aspect and status information)

6.2.1.1.5. Abstract Events and Commands

- **AC: Abstract Command** (instance of a subclass, performs command execution in the corresponding Viewport)
- **CRE: Command Request Event** (instance of a subclass, holds an Abstract Command, forwards a command request to the Viewport)
- **CUE: Control Update Event** (instance of , updates a Window upon model changes)
- **OUE: Object Update Event** (instance of , propagates modifications of an Object Aspect to the model)

6.2.1.1.6. Abstract Windows and View

- **ACtl: Abstract Control** (instance of a subclass, shadows a platform widget, holds Abstract Events)
- **AVw: Abstract View** (instance of , shadows the Platform View)
- **MWC: Main Window Control** (instance of , an Abstract Control for the Platform Main Window, shell or top pane)

6.2.1.1.7. Platform Adapter Classes

- **MWA: Main Window Adapter** (instance of a subclass)
- **VA: View Adapter** (instance of a subclass)
- **WA: Window (Widget, Pane or Part) Adapter** (instance of a subclass)

6.2.1.1.8. Platform Windows and View

- **MW: Main Window** (platform-specific: top pane, shell, ...)
- **PVw: Platform View** (the object which holds and coordinates the view on the platform)
- **Wn: Window** (**Widget**, **Pane or View Part**)

6.2.1.2. Abstract Controls incl. Internal Interfaces

Focus change, command issuance and transfer of data - including a range of state information like validation or authorization - will be performed in this abstract layer. The real Controls are completely decoupled from all Domain Process actions and Domain Model Aspect data; all technical details of external interfaces (GUI, DDE, ...) are completely hidden in the Abstract Control implementation.

In contrast to the large real Control class hierarchy (wrapping or representing the whole group of emulated or native Controls), there are only few Abstract Control classes, whose instances are managed by their Abstract Views, which themselves are held in a Model View Connector. Each real Control class has a unique but changeable mapping to the hierarchy of Abstract Controls. A simple, fixed protocol is implemented in the Abstract Control to perform changes in the corresponding real Control. Because of this, different windowing systems can be supported uniformly by the framework (Visual Age / Composition Editor).

6.2.1.2.1. Abstract Control Classes

- CommandRequestControl:Push Button, MenuItem, SmartIcon
- ObjectAspectControl:ContainerItem, Slider, RadioButton, SpinButton
- ObjectAspectEditorControl:Entry Field, Text Control
- GroupControl:GroupPane, Composite
- DynamicControl:(Controls which allow components to be added dynamically at runtime)
- DynamicMenuControl:Menu
- ObjectBookControl:Notebook, other book-like ones
- PageControl:Notebook page
- MainWindowControl:ApplicationWindow, TopPane
- ObjectCollectionControl:ListBox, Table, ColumnarListBox, Container
- ObjectChoiceControl:DropDownList
- ObjectRangeControl:Slider, SpinButton

6.2.1.2.2. Technical Details of Abstract Controls

All Abstract Controls are instances of subclasses of . At initialization, an Abstract Window is created for each platform component window together with its corresponding Platform Adapter and connected. Afterwards, the Abstract Window gets the component name of the Platform Window from the Platform Adapter. This name is checked for syntactical accordance to a Model Accessor name, and if it matches an accessor name, i.e. the Window is wanted to communicate with the framework models, it is converted into a Model Accessor object. Moreover, the Abstract Window creates Abstract Events for the Platform Events which are to be hooked to ApplicationFramework and for the model (resp. Viewport) events which are to cause view updates. After creation and registration of the framework structures an initial update of the Window with the corresponding model contents is performed. Finally the initialization method recourses into the child platform panes to create framework structures for them and connect them as explained.

At runtime, the Abstract Windows hold their corresponding Abstract Events. They are called by the Abstract Events which are responsible for the view update upon changes in the model world. These calls are forwarded to the corresponding Platform Adapters mainly by the method , which takes an instance of class . The Abstract Value packs the value to be put into the widget with state information about visibility, editability, sensitivity for user inputs, color and other Window-specific information. The Abstract Window unpacks the Abstract Value and sets the corresponding properties in the view window by calls to the Platform Adapter.

is an Abstract Window which shadows Push Buttons, Menu items and similar widgets which can be selected to send a command to the model world. It holds an Abstract Event of the class (or a subclass) which uses an Abstract Command to send the request to the Viewport of the corresponding default Domain Process object.

corresponds to a Menu which can access the Domain Process (if its platform Menu has a name which installs model access) to add and remove Menu items at runtime.

is the Abstract Window for all Windows capable of showing some Aspect or representation of a model. The Object Aspect Control itself is used for read only Aspects (like static texts). There is a subclass, the , which supports editable Aspects (like the text of an entry field). The Abstract Window class is the same for a variety of Controls, because the abstract protocol is equal. Communication with the model is performed by the Abstract Events, and the translation to the protocol of the real window is done by the Platform Adapter. A few Object Aspect Controls require special treatment, therefore their Abstract Windows are subclasses of .

has other Controls as children. The Group Control can have a title which can be set according to one of the Model Aspects. Some Group Controls allow dynamic setting of their internal components at runtime, their abstract superclass is a subclass of named .

is a specialized Group Control to handle the Main Window and, respectively, the shell. It contains some special Abstract Events to activate and to close a view.

and are the Abstract Controls for Notebooks and Notebook Pages.

is the Abstract Window for all kinds of Lists. Its protocol is somewhat more complicated than other Object Aspect Controls because it has to care for the list of objects to be represented, the list of selected objects, and the representation of the objects shown in the List. Therefore, it creates a special type of Connection at runtime, a List Model Connection (Instance of class), where Viewports are created for the objects shown in the List. These Viewports are passed to the Platform Adapter of the List, which either passes them to the platform List, if the List can handle objects, or it extracts the representation string from each Viewport and passes a string list. Moreover, it creates a special Abstract Event ( ) to pass a selected model to the default Domain Process in a command which, for example, may open some child process upon the chosen model object.

subclasses to add some enhancements to facilitate treatment of single selections as required by Combo Boxes and Radio Button Sets.

is the Abstract Window which shadows List Controls like Scrollbars or Sliders where the list is a numeric interval and the selection is a value from this interval. In comparison to object Collection Controls, it has a simplified protocol which does not create special Connections at runtime, but manages the update of the characteristic parameters for a Range Control.

6.2.1.3. Abstract View incl. Internal Interface

When a real View is created, an Abstract View begins to exist in its shadow. The lifetime of the Abstract View is determined exactly by the lifetime of the real View. The Abstract View's purpose is to manage the Abstract Controls corresponding one-to-one to the real Controls in the view. Moreover, the Abstract View performs coordination between the abstract and platform layer when opening, activating and closing the view. It communicates with the real Platform View using a Platform Adapter.

Important Methods of the Abstract View:

- class has a method which is called at initialization from the Domain Process via the Model View Connector:
- asks the windowing policy object passed by the Domain Process for a new Platform View and creates an initialized Abstract View.
- The instance delegates platform-specific tasks to its corresponding :
- uses the Platform Adapter to initialize and open the Platform View, and to retrieve the component windows of the Platform View in their hierarchy. Additionally, in this method the Abstract Windows are created and initialized for the component windows which leads to creation of the corresponding Window Adapter between each Abstract Window and Platform Window. The open method also initializes the interaction with the model world, opening and activating the Viewports.
- sends a request to the Platform View to get the focus.
- closes and destroys the real View using the Platform Adapter.
- transmits an abort close request to the Platform View when required.

6.2.1.4. Abstract Event incl. Internal Interface

Abstract Window Events, instances of subclasses of , are the objects that actually perform the communication between the view system and the model world, whereas Abstract Windows are merely containers for Abstract Events which may provide some additional services for them. Abstract events can be divided into two functional types:

Abstract events which propagate changes in the model world to the view and Abstract Events which propagate changes or requests from the user interface to the model world. This difference is not entirely reflected by the class hierarchy, but it is important for the Platform Adapters. Abstract events which dispatch user requests to the models are hooked to Platform Events (events triggered by the Platform Windows) at initialization time. They may have platform-specific extensions, methods which conform to the syntax needed by the Platform Windows for callbacks or event handlers or which may transform the way the event is triggered by the platform into the way the event is needed in the Application Framework. Abstract events which cause an update of the view upon changes in the model world are entirely platform-independent, they call their corresponding Abstract Window for the update, which uses its Platform Adapter.

The following list shows part of the event hierarchy:

```
MicFwAbstractWindowEvent
| MicFwAbstractControlEvent
| | MicFwCommandRequestEvent:Button-click, Menu-select,...
| | | MicFwCloseRequestEvent:System Menu-close,...
| | | MicFwActivateRequestEvent:Window activation
| | MicFwGenericAbstractEvent:(not configured)

MicFwObjectAspectEvent
| | | MicFwControlUpdateEvent:Update Instance variable
| | | MicFwObjectUpdateEvent:Update single Aspect Control
| MicFwObjectSelectEvent:List-select,...
```

6.2.1.4.1. Description of Abstract Window Events

- is an event triggered when a Push Button or Menu item in the view is clicked. At initialization ( ), an Abstract Command (instance of a subclass of) is created according to the accessor name of its window, either a generic one which transforms the accessor name into a selector and propagates it to the default Viewport, or a specific one which has a predefined execution method and performs some standard request of the window system (e.g. 'close' → to close a Domain Process and its corresponding view).
- The method , which is called by the Platform Event handler, first enforces a model update if there is a widget (e.g. entry field) which holds changes not yet propagated to the corresponding model (method). Afterwards, it executes its Abstract Command.
- is a special Command Request Event bound to a Main Window and triggered when the Window is activated (gets the input focus). It holds a
- is a special Command Request Event bound to a Collection Control and triggered when there is a request for the default Domain Process to perform some action upon the selected item (e.g. double-click on a list item). It holds a
- is a special Command Request Event bound to a Main Window and triggered when the Window is to be closed. It holds a which checks to see if the corresponding Domain Process and its children can be closed, and if so performs the close; if not it aborts the close.
- is a special Command Request Event bound to a Notebook Page Control and is triggered when the page is exposed and gets the input focus. It holds a .
- is the abstract superclass for all events which are concerned with the synchronization of model and view.
- is the major event for propagating changes in the model world to the view. Its method filters the incoming model events to the symbol anAspect, retrieves the updated Aspect data and state information to control visibility, sensitivity, editability etc. of the corresponding view widget, and invokes the correct update services of the corresponding Abstract Window.
- is an event which is triggered by the models if the validation mechanism is active to convert the result of validation into visual signals in the view.
- and hook Platform Events of the view and retrieve help information from the Viewport which mainly communicates with the widget for which the help is requested.
- is the major event for propagating any editing of Object Aspects performed by the user in the view to the corresponding model. It is hooked to a Platform Event of the corresponding widget. Its trigger method retrieves the actual data from the widget via the Abstract Window and Platform Adapter and calls the Viewport to set the corresponding Aspect to the retrieved value.
- is similar to the Object Update Event; but it does not necessarily perform an

update. Instead, the Abstract View is informed about the change and memorizes that the widget holds modified data. This will cause any Command Request Event to enforce an update of the model with the modified data kept in the widget. The behavior of the object change event can be configured in the Viewport by overriding the method This event is used by entry fields to hook the event triggered upon each key stroke.

- is an Abstract Event for communicating the selection made in a List or similar widget to the models. It performs an update of the Model Aspect which holds the selection and additionally updates the selection in the List Connection, the Connection dynamically created by the abstract Collection Control.
- is an Abstract Event which can hold a platform-specific method to retrieve a graphics representation of a model object (an icon) returned by the Viewport method .

6.2.1.5. Abstract Command incl. Internal Interface

All of the Command Request Events launch the execution of Abstract Commands. The behavior provided by the framework's generic Class is to pass on the given selector without change to the connected Domain Process or its Viewport.

This behavior can be refined or extended by providing and registering specific command classes. Some examples of functionality which can be implemented this way:

- Macros (chaining of commands)
- Crash recovery
- Queuing (caching of commands for later execution)
- Transfer (propagating a command to another process)
- Undo / Redo (rollback / rollforward of commands in arbitrary depth)

6.2.1.6. Abstract Value

Each Abstract control owns an Abstract Value object (instance of or a subclass) which it sends to the corresponding Viewport upon view update to retrieve the Aspect value to be shown by the control together with all necessary Subaspect information like visibility, editability, enabled state etc. of the control.

6.2.1.7. Aspect Feature

The Aspect Feature is like a Viewport counterpart to the Abstract Value. Aspect features are owned by each Viewport class for each Aspect the Viewport may be asked by a view component. The Aspect Feature keeps the information how to retrieve its Aspect and Subaspect values and how to set the Aspect value. Aspect features are generated automatically using reflectivity, but as a mean of optimization Viewport subclasses can create specific "hard-coded" Aspect Feature objects. The transmission of commands is handled in a similar way by the Viewport using Action Feature objects.

6.2.1.8. Evaluators

- Evaluators are encapsulated message sends or blocks, which have a common protocol for evaluation given by the methods
  - ,
  - :,
  - :,
  - :.
- Each of these methods can be called. Missing arguments are substituted by default values, superfluous arguments are ignored. Basically, there are two types of Evaluators, Directed Evaluators, which act like directed messages, and Anonymous Evaluators, where the receiver has to be passed as first argument. From the technical point of view, there are generic Evaluators which use reflectivity for the method call, and hard-coded Evaluators.
- The application framework uses Evaluators for two purposes:
- The Aspect and Action Feature objects have Evaluators to get and set the specific Aspect resp. perform the action. Model object classes also generate Evaluators for their user-implemented features, which are requested by the Aspect / Action Feature object for direct communication with the model, i.e. if there is no complementary method implemented in the Viewport.
- ApplicationFramework event handling is based on a special Evaluator, the , which can hold a linked list of Chainable Evaluators (subclass of) behind itself which are called in sequence when the Method of the chain head is called. There are and methods to add / remove Chainable Evaluators, and a method to remove all Evaluators for a given receiver. The Evaluator Chain Head has a well-defined

behavior when an Evaluator is added or removed during evaluation, i.e. iteration through the list: Added Evaluators will not be evaluated during the current iteration; removed Evaluators will no longer be evaluated, even if they are still in the stack of the current iteration.

6.2.2. Adapters

6.2.2.1. Platform Dependent and Independent Components

The Application Framework divides the window system (the view-controller-layer) into an abstract platform-independent side and the Platform View. The platform-independent window system (Abstract View, Abstract Controls, Abstract Events and commands) provides an interface to the view according to the architecture and the protocol of ApplicationFramework. The Platform View, however, forms part of the "host" Smalltalk system (e.g. IBM Visual Age), and therefore, it presents its own architecture and protocol. Because of this, a Platform Adapter system is used as an extension of the framework to perform the translation of the protocol and overcome the architectural differences between the host Smalltalk system architecture and ApplicationFramework architecture. It should be noted that the ApplicationFramework does not intend to take complete control over the platform window system, but take advantage of the host system capabilities and interface only those system features which are relevant for interacting with ApplicationFramework.

The Window System Adapter of ApplicationFramework consists of the View Adapter (instance of class), the Window Adapters (instances of subclasses of), and some platform-specific extensions to Abstract Events and Viewports. The View Adapter interfaces the platform object which holds and controls the Platform View and the Abstract View. Additionally, for each component window of the Platform View (View Part, widget, pane) there is a Window Adapter which interfaces to its corresponding Abstract Window.

6.2.2.2. Window and Control Adapters

The Platform Adapters which connect the Platform View with the Abstract View are instances of , which has a set of methods implemented for each platform supported. The Platform Windows (widgets, views, parts) are connected with their corresponding Abstract Controls by instances of platform-specific subclasses of .

6.2.2.3. View Adapter

Each Abstract View (instance of) has a corresponding object interfacing the real View. Access to the view, retrieval of its top level window (method) and component windows (method) and commands like open, activate, close, abortClose are implemented as platform dependent instance methods in .

There is a unique Platform Adapter class for each Platform Window class. The Platform Adapter class name conforms to the convention MicFw<platform widget class name>Implementation.

6.2.2.4. Tasks of the Window Adapter

The Platform Adapter maps the Platform Events ApplicationFramework is interested in to corresponding Abstract Events. The class method #implementedEvents, which is implemented for each Window Adapter, returns a dictionary with the Platform Event symbols as keys and the corresponding Abstract Event classes as values. The instance method performs the creation and platform-dependent registration of the view observing Abstract Events for their corresponding real events and returns the collection of Abstract Events. The instance method can be overridden by Window Adapter classes to perform specific event registrations (e.g., if platform and Abstract Events cannot be mapped in a clear one-to-one relationship). The method has to do the corresponding deregistrations when the view is closed.

Furthermore the Platform Adapter must translate the Abstract Control protocol of "getting" and "setting" contents and properties to the platform widget protocol.

Later sections describe how an Adapter for a GUI is created and where the hooks are for plugging in a new Widget Adapter.

6.2.2.5. Abstract Window Adapter Superclasses Common to All Platforms

Each Abstract Control can interface its real Control by a Window Implementation wrapping access to values, events, children etc. The hierarchy of the abstract implementation classes follows:

```
MicFwWindowImplementation:(root class for all Widget Adapters)
| MicFwControlImplementation:(root class for visual widgets)
| | MicFwAspectControlImplementation: (Object Aspects, visual parts)
| | | MicFwLazyControlImplementation:(for controls which have no relevant
            behavior)
   | | | MicFwMultipleAspectControlImplementation: (for all controls which
            contain ...)
   | | | | MicFwBookControlImplementation: ("Notebooks", showing more than
              one Domain Process)
   | | | | MicFwCollectionControlImplementation: (object collections, showing
              a list of objects)
   | | | | | MicFwTextCollectionControlImplementation: (object collections,
                but the platform widget accepts strings only)
   | | | | MicFwGroupControlImplementation: (groups of "child" platform wid-
              gets)
   | | | | | MicFwMainWindowControlImplementation: (Adapter for the shell
                widget, Main Window, top pane)
   | | | | MicFwPageControlImplementation: (note book pages)
   | | | MicFwSingleAspectControlImplementation: (representing one Aspect of
            one or more objects)
   | | | | MicFwChoiceControlImplementation: (List widgets with a mandatory
              single selection shown in a special way)
   | | | | MicFwRangeControlImplementation: (widgets similar to List widgets
              showing a value out of a numeric interval)
   | | | | MicFwStaticControlImplementation: (showing one Object Aspect read
              only)
   | | | | MicFwTextControlImplementation: (showing an Object Aspect as edit-
              able string)
   | | | | MicFwToggleControlImplementation: (an object represented by a
              boolean typed Aspect - allows true or false only)
   | | MicFwCommandCollectionControlImplementation: (list or group of command
          requests, e.g. Menus)
   | | MicFwCommandRequestControlImplementation: (command request controls
          like Push Buttons and Menu items)
   | MicFwVisualProgrammingImplementation: (non visual parts, VA integration
        of the Application Framework)
```

Each Window Adapter class is a subclass of one of the classes above. Class
implements most of the methods common to all windows.

6.2.2.6. Platform Methods

With the exception of integration methods (see below) most of the methods which require re-implementation according to the necessities of the specific widget have abstract or default implementations in class

The methods are

6.2.2.6.1. Instance Methods

- [:] (Get or set the background color of the widget),
- [:] (Get or set the foreground color of the widget),
- [:] (Get or set the real View's checked state),
- [:] (Get or set the real View's editable state),
- [:] (Get or set the real View's enabled state),
- [:] (Get or set the real View's visible state),
- [:] (Get or set the real View's text. This may be a label, the first part of the content or anything that can be used as a hint in a textual representation of it, e.g. a browser, log, etc...),
- [:] (Get or set value),
- [:] (Get or set selection (collections)),
- activate (Set the focus),
- childPlatformPanes (Retrieve all child controls),

6.2.2.6.2. Class Methods

- implementedEvents
- (Answers a dictionary with supported real events as keys and Abstract Events as values. It is implemented in and has to be re-implemented in subclasses which must map Platform Events to Abstract Events.)
- abstractWindowClass

- (must be implemented in every Adapter class and returns the class of its Abstract Control.)

6.2.2.7. Mapping of Platform Events

In communication between the models and the user interface, events are important in both directions.

- A Graphical User Interface (GUI) triggers events upon actions of the user.
- In the opposite direction, events are triggered by the models causing GUI widgets to be updated when some has changed in the model object net.
- ApplicationFramework is designed to be platform independent. Therefore, it maps events triggered by the platform GUI to its own Abstract Events which have platform-specific extensions to transform the Platform Event protocol into the protocol of the framework. These Abstract Events are instances of:
- classes named (used for modifications in s)
- the class and its subclasses (used for command requests like pressing a Push Button).

The Abstract Events responsible for passing changes from the model world to the window system are instances of classes named .

As already mentioned, the classes are responsible for the mapping (method).

Example:

VA composition editor - Class hierarchy

The following hierarchy shows the Platform Adapters which connect the VA ABT parts with the Abstract Controls of ApplicationFramework.

```
MicFwWindowImplementation
| MicFwControlImplementation
| | MicFwAspectControlImplementation
| | | MicFwLazyControlImplementation
| | | | MicFwMultipleAspectControlImplementation
| | | | | MicFwBookControlImplementation
| | | | | | MicFwAbtNoteBookViewImplementation
| | | | | | | MicFwAbtPortableNotebookImplementation
| | | | | MicFwCollectionControlImplementation
| | | | | | MicFwAbtExtendedListImplementation
| | | | | | MicFwAbtContainerDetailsViewImplementation
| | | | | | | MicFwAbtContainerIconAreaViewImplementation
| | | | | | | MicFwAbtContainerIconListViewImplementation
| | | | | | MicFwAbtListViewImplementation
| | | | | | MicFwAbtMultipleSelectListViewImplementation
| | | | | | MicFwTextCollectionControlImplementation
| | | | | MicFwGroupControlImplementation
| | | | | | MicFwAbtCompositeViewImplementation
| | | | | | MicFwAbtGroupBoxViewImplementation
| | | | | | MicFwMainWindowControlImplementation
| | | | | | | MicFwAbtShellViewImplementation
| | | | | MicFwPageControlImplementation
| | | | | | MicFwAbtNoteBookPageViewImplementation
| | | | | | MicFwAbtPortableNotebookPageViewImplementation
| | | | MicFwSingleAspectControlImplementation
| | | | | MicFwAbtContainerDetailsColumnImplementation
| | | | | MicFwChoiceControlImplementation
| | | | | | MicFwAbtComboBoxViewImplementation
| | | | | | MicFwAbtDropDownListComboBoxImplementation
| | | | | | MicFwAbtRadioButtonSetImplementation
| | | | | | MicFwAbtSpinButtonViewImplementation
| | | | | MicFwRangeControlImplementation
| | | | | | MicFwAbtScaleViewImplementation
| | | | | | MicFwAbtSliderViewImplementation
| | | | | MicFwStaticControlImplementation
| | | | | | MicFwAbtLabelViewImplementation
| | | | | | MicFwAbtSeparatorViewImplementation
| | | | | MicFwTextControlImplementation
| | | | | | MicFwAbtTextViewImplementation
| | | | | MicFwToggleControlImplementation
| | | | | | MicFwAbtToggleButtonViewImplementation
| | | | MicFwCommandCollectionControlImplementation
| | | | | MicFwAbtCwMenuViewImplementation
| | | | MicFwCommandRequestControlImplementation
| | | | | MicFwAbtHotSpotViewImplementation
| | | | | MicFwAbtPushButtonViewImplementation
| | | MicFwVisualProgrammingImplementation
| | | | MicFwAbtVariableImplementation
| | | | MicFwMicFwDomainManagerPartImplementation
```

6.2.2.8. Example of an Adapter for an Entry Field

This section describes how a Platform Adapter is created. As our example, we will use an entry field in Visual Age (VA). In VA the class of the entry field widget is named AbtTextView. The class of the Adapter for the entry field is . Below are methods which are re-implemented in a manner which differs from their implementation in the root classes. One of these methods is editable:, which is used to change the editable flag of an entry field.

Therefore, the method editable: is re-implemented in the Adapter class named . The method is executed when an event in the object net triggers a change concerning the corresponding entry field.

For the textual representation of the widget, the method text is re-implemented. An entry field in VA is represented by the user input.

6.2.2.9. Mapping of Platform Events

The entry field requires a different event mapping than that of the root class. The re-implemented method maps the Platform Event losingFocus to the framework event

,

- stringChanged to and
- helpRequested to .

losingFocus is triggered if the user leaves the entry field. The framework checks to see if it is necessary to update the object net.

stringChanged is triggered if a key stroke occurs. The user has the chance to activate this event if needed. So it should be mapped.

helpRequest is needed for the help system. The user can activate help for every attribute in the Viewport.

6.2.3. Feature Objects

The Aspect Feature is like a Viewport counterpart to the Abstract Value. Aspect features are owned by each Viewport class for each Aspect the Viewport may be asked by a view component. The Aspect Feature keeps the information how to retrieve its Aspect and Subaspect values and how to set the Aspect value. Aspect features are generated automatically using reflectivity, but as a mean of optimization Viewport subclasses can create specific "hard-coded" Aspect Feature objects. The transmission of commands is handled in a similar way by the Viewport using Action Feature objects.

6.2.4. Exception Handling

To allow an easy way of handling conditions that may cause the framework not to operate correctly due to developer's mistakes, missing Adapters and other circumstances, the framework signals specialized exceptions that provide a default behavior for them. To override the behavior of exceptions please refer to the chapter explaining the implementation and use of the Exception Handling System.

Application Framework flags the following exceptions:

- **MicFwInvalidModel**: During development this exception signals an invalid definition of a Domain Object class as an involved Domain Object within a Domain Process description. During runtime it signals that the application wants to set a Domain Object into a Connection that is not compatible with the defined involved Domain Object class.
- **MicFwInvalidView**: Signaled when the class hint passed for the view class in the #openView... method cannot be resolved to a valid view.
- **MicFwMissingAccessor**: Signaled when an Aspect requested from a Viewport cannot find a corresponding accessor neither in the Viewport nor in the model.
- **VetoAction**: Silent exception internally used to interrupt model access.

6.3. Substituting Subsystems

To offer as much flexibility as possible the Application Framework uses some well known patterns to decouple requests from their implementation. This is especially useful for developers in extending or even modifying the Framework's default behavior where this is required to meet the application's needs. In this section we give you a brief overview of the substituteable subsystems and components and describe how they are connected to the Framework kernel. The following sections describe some details and examples on how to implement and register your own components.

From the programmer's point of view the Framework must support all steps of an evolutionary or spiral development approach. The programmer starts implementing essential domain classes, their relationships and views. As the project proceeds, Domain Processes, Viewports and other components may follow.

Collections of hard coded primitives (Symbols, Strings, ...) are substituted with support tables read from databases, the user interface gets more and more elaborated and a growing number of reusable pieces are detected and isolated.

The Framework's architecture is prepared to support huge applications. This leads to a highly decentralized location of responsibilities. To enable Framework beginners as well as project developers to fill in concrete implementations at the time they are needed, generic Objects are used as much as possible. This also enabled us to keep the Framework's code free of errorprone flow control statements and makes it easier to maintain and extend. However, generic Objects have to be very common and sometimes the assumptions that were made in deciding on their behavior may not meet your project's requirements. For that reason, most of them are exchangeable. Exchangeable generic classes are:

- is the place where each user action is handled that is not implemented by a framework or application specific Abstract Command. It implements the defaults for passing the request to the ModelViewConnector, the associated Viewport, execution details and logging.

Additionally there are some generic classes that are intentionally not exchangeable. They offer mechanisms and behavior that are embedded so deeply within the Kernel of the framework that modifications can only be done by subclassing them with concrete classes and registering these in a normal, defined way.

- is always taken as generic Dispatcher if you didn't implement your own subclasses. It provides defaults for mapping the Domain Model Aspects to its corresponding and aggregating View Aspects, automated update between Domain Objects and views, mapping authorization and validation rules of the Domain Objects to state control of view elements (editable, visible,...), formatting and a lot more.

The Framework's Mappers and Brokers provide an easy to use mechanism to allow the exchange of existing subsystems. Each exchangeable subsystem of the framework is coupled via a Broker which must be registered with the Broker Mapper after loading the subsystem. The Broker Mapper is one of several Mappers that will be explained later on in this chapter. It maintains a collection of subsystem names (which are unique and constant for all future releases) and their associated Brokers. During the following sections you will have a closer look at the currently implemented subsystem Brokers and a brief overview of their responsibilities. We also describe in a detailed example how to implement and plug in a Broker.

6.3.1. Brokers

A subsystem implements its public interface in several ways. It may offer a public class with some methods presenting services that are requested and handled at once, returning a result which can be used without further actions. Additionally it may define classes that are used more interactively, the application has to know a lot about the communication with them. The most integrated subsystems implement frameworks, the developer must create subclasses and override methods to specialize their behavior to meet the application's requirements.

To allow subsystems or specific requests to them to be exchanged with an implementation, the framework uses Broker classes that offer a thin public interface with the internal knowledge how to delegate the call to the subsystem. Only the Broker knows class names or other details of the subsystem so it decouples the request from its implementation. If returned elements of the subsystem are then used within the application, they have to implement the expected public interface which usually is called a type based coupling in contrast to a class based coupling. This is a lot more flexible since clients don't have to hardwire these class names.

A common Broker concept enables the developer to modify request algorithms or subsystem behavior quite easily. To avoid even the hardcoding of calls to the Broker classes, the framework maintains each Broker within one single Broker Mapper that provides the developer with a simple interface to register, deregister and use the services. To register the new Broker   that answers an appropriate Viewport for a passed Domain Object call:

```
self broker use: MyViewPortRequestBroker for: #viewPort.
```

To remove the Broker you call:

```
self broker remove: #viewPort.
```

The framework will use the registered Broker automatically to retrieve Viewports instead of using the default mechanism provided by the original product release. This sample shows how Brokers are used by the framework to allow modifications of implemented algorithms or the behavior of entire subsystems even at a high abstraction level and in central places within its design.

6.3.1.1. Implemented Brokers

The current release of the framework implements five Brokers to allow modifications to default behavior or subsystem usage:

6.3.1.1.1. MicFwAbstractCommandRequestBroker

takes requests by the framework to handle the mapping between concrete commands sent by the user interface and the corresponding Abstract Commands supported by the abstract Model View Controller or the application. The underlying subsystem is .

Its interface consists of:

```
abstractCommandFor: aCommandName
"Public - Answer the AbstractCommand that corresponds to <aCommandName> "
```

6.3.1.1.2. MicFwAbstractControlRequestBroker

takes requests by the framework to handle the mapping between real Controls (panes or fields embedded in views) by the user interface and the corresponding Abstract Control supported by the abstract Model View Controller. The underlying subsystem is .

Its interface consists of:

```
getImplementationOf: aRealPaneClass
"Public - Answer the implementing class of <aRealPaneClass>. To allow inherit-
ance reuse, <aRealPaneClass> may be implemented like any of its superclasses. "

abstractControlFor: aRealPane
"Public - Answer an abstract control derived from the class of <aRealPane>. Use
<aRealPane> to connect the abstract control to the rest of the world. "
```

6.3.1.1.3. MicFwAbstractEventRequestBroker

takes requests by the framework to handle the mapping between concrete events triggered by the user interface and the corresponding Abstract Event supported by the abstract Model View Controller. The underlying subsystem is .

Its interface consists of:

```
abstractEventFor: aRealEvent in: aRealWindow
"Public - Answer the corresponding abstract event. "
```

6.3.1.1.4. MicFwViewPortRequestBroker

takes requests by the framework to handle the mapping between Domain Objects resp. Domain Processes and their Viewports. The underlying subsystem is .

Its basic interface methods are:

`MicFwViewPortRequestBroker >> openViewPortForModel: aModel inConnection: aConnection`

`MicFwViewPortRequestBroker >> openViewPortForModelClassNamed: aSymbol inConnection: aConnection`

`MicFwViewPortRequestBroker >> viewPortClassForConnection: aModelConnection`

6.3.1.1.5. MicFwCachingViewPortRequestBroker

is provided as subclass of .

6.3.2. Mappers

To maintain registration of loosely coupled elements within the framework, Mappers are used to set, get and remove associations between unique, constant names and their corresponding classes, which themselves may change or be reimplemented in your project. Mappers are implemented as singletons and can be reached through an easy to use interface as they extend Object with one method per Mapper.

Each Mapper has its own working field. You will find a complete list of them together with a brief explanation of their responsibilities in the next section. In addition, an example will show you how to use the to exchange the default generic Domain Process in order to modify the application-wide handling of views that are not opened and controlled via a specific subclass of Domain Process.

6.3.2.1. Implemented Mappers

The current release of the framework implements four Mappers to allow modifications to default behavior or subsystem usage:

6.3.2.1.1. MicFwMapper

serves as an abstract superclass for all Mappers. Its interface consists of :

```
MicFwMapper >> use: aResponsibleObject for: aCategorieSymbol

MicFwMapper >> remove: aCategorieSymbol


MicFwMapper >> for: aCategorieSymbol
```

6.3.2.1.2. MicFwBrokerMapper

maintains the mapping to Brokers. Its interface consists of the inherited methods of plus:

- no additional methods.

It is called via:

```
self broker for: #theBrokerName
```

Its current mappings are:

abstractControl	MicFwAbstractControlRequestBroker
abstractEvent	MicFwAbstractEventRequestBroker
abstractCommand	MicFwAbstractCommandRequestBroker
viewPort	MicFwViewPortRequestBroker

6.3.2.1.3. MicFwBrowserMapper

maintains the mapping to the framework tools. Its interface consists of the inherited methods of plus:

- no additional methods.

It is called via:

```
self micFwBrowser for: #theBrowserNumber
```

Its current mapping is:

1	MicFwLogBrowser

6.3.2.1.4. MicFwGenusMapper

maintains the mapping to generic classes. Its interface consists of the inherited methods of plus:

`MicFwGenusMapper >> use: aResponsibleObject as: aCategorieSymbol`

It is called via:
`self genus for: #theGenusClassName`
Its current mappings are:

domainProcess	MicFwGenericDomainProcess
command	MicFwGenericAbstractCommand

6.3.2.1.5. MicFwInterfaceMapper

maintains the mapping to used subsystem interfaces. Its interface consists of the inherited methods of plus:

- no additional methods.

It is called via:
`self micFwInterface for: #theInterfaceName`
Its current mappings are:

authorization	MicFwAuthorizationManager
transactionManager	a MicFwTransactionManager
validation	IdentityDictionary

6.3.2.2. An Example of Implementing Generic Behavior Using a Mapper

As an example of modifying the default behavior of views without an explicit Domain Process we will exchange the framework's with an implementation which will be a subclass named . Let's assume we want to run our views in an isolated transaction context in contrast to the framework's default, which is not to support any transaction behavior with generic Domain Processes at all.

- Create a subclass of named located in your subsystem.
- Use the Domain Processes Browser to declare and store the requested transaction behavior with the new .
- Use your loaded (VA) or initialize (VSE) mechanism to implement the registration of as the generic Domain Process the Framework will use in the future. This will take place by calling:
- self genus use: as: #domainProcess.
- Version and release your code, reload your subsystem or execute the registration code to activate the exchange now.
- Now test your modification by executing one of the framework examples that uses a view without an explicit Domain Process. You find all provided examples in the ExamplesBrowser that can be reached through the micFrameworks in your Transcript window. You can examine the new behavior with the Logger, ViewConnectorsBrowser and TransactionsBrowser.
- If necessary you may now implement additional methods in to meet further requirements.

6.4. Adding Your Own Widget

6.4.1. How to Create a Platform Adapter for a New Widget

- Check the behavior and the protocol of the widget to be adapted:
- If the widget is passive, but contains other possibly active widgets which are to communicate with ApplicationFramework, then its Abstract Window must be a and the Platform Adapter a subclass of .
- If the widget behaves like a Push Button, the Abstract Window is a and the Adapter a subclass of .
- If the widget is some kind of specialized entry field, the Abstract Window is a and the Platform Adapter a subclass of .
- If the widget behaves like a List, the Abstract Window is a and the Platform Adapter a subclass of . Furthermore, it must be determined whether or not the widget can accept a list of objects which are queried for the Aspect to be presented by the widget itself, or if the widget needs the presented Aspect to be extracted by the Adapter and passed as a list of strings or numbers (So the Adapter may be subclass of).
- Other types of widgets like Combo Boxes or Radio Button Sets (with a subclass of), Sliders or scrollbars (with a subclass of) etc. are treated in an similar manner. It must be considered how these widgets are adapted to locate the right place of the new Platform Adapter within the class hierarchy, and what kind of Abstract Window corresponds to them.
- After having found the right place in the class hierarchy for the Adapter class, the class is created with the name where <WidgetClassName> is the name of the platform widget class. Next, it must be determined whether the inherited class method returns the desired Abstract Window class for the new widget. If this is not the case, then this method must be overridden. Moreover, a class method must be created which returns an IdentityDictionary with the Platform Event selectors as keys and the corresponding Abstract Event classes as values. It is not necessary to map all Platform Events to Abstract Events, only those which need to be communicated to ApplicationFramework.
- Next, you must determine whether the instance methods inherited by the Platform Adapter are implemented as needed for the widget. They should have the effects described above or do nothing if they do not apply to the widget. Finally, the specific services of the widget must be adapted, especially the methods for getting and setting the content of the widget, for getting and setting the selection in a List widget (Further information concerning List Widget Adapters is included in the next chapter).

6.4.1.1. Platform-Specific Methods in Viewports

can have platform-specific extensions because, in the case of , Viewports are created and handed over to the Platform Adapter, which can pass them to the platform widget if it handles objects. As the platform widget addresses them in a platform-specific way, Platform Adapter extensions may be necessary to intercept the calls from the platform widget and perform ApplicationFramework-specific tasks like event registration for propagation of model changes. As a Visual Age example, Viewports have a platform-specific method
which intercepts the Visual Age mechanism for getting a representation of an Object Aspect. Therefore, Viewports can be passed to the platform list widgets as they are.

On platforms where the Viewports are called directly by sending them an unary message, the Platform Adapter must wrap them with an instance of in order to have the framework-specific event registration switched on.

7

Multi Language Support (MLS)

7.1. Language Support Base System

7.1.1. Overview

The MLS system enables all Locale-(language and territory combination) specific data of an application to be externalized in a simple manner.This can be all forms of Labels or other text, as well as formats for date and time, and all forms of numeric values.

The data can be stored in various storage media. At present, we support ENVY classes, sequential files as well as relational databases containers for externalization.

All data will be stored in unicode, a platform-independent 16 bit character set.

Apart from correctly displaying model information on the view, the format strings are used for transforming input from the view (e.g. strings) into valid model objects (e.g. time, date).

Input errors relating to defined format are identified, and only input which complies with the format is transformed into valid objects and placed into the model.

We provide a simple API for enabling specific parts of an application to be switched over to the required Locale.

The MLS system is fully integrated in Application Framework, and a generic Adapter provides a comprehensive default reaction, thus minimizing the amount of work involved in connecting the MLS system to the user’s own applications.

It is possible to access the MLS system independently of a view in order to obtain MLS Data relating to the model, and to visualize these data in a form which depends on the status of the model. This permits a high level of flexibility for using the system.

As a result of MLS being integrated in Application Framework, not only VA Views but also HTML pages and Java Applets are capable of supporting several languages without any model change.

The MLS is connected to the transaction handling facility of Object Behavior Frameworks. This also means that run time changes to the MLS Data are also transaction-capable.

We provide a sophisticated editor for maintaining externalized data in a simple manner.

7.1.2. Overview

The micado Multi Language System (MLS) is an extension of the existing National Language System (NLS) such as the one provided by IBM. The micado MLS is intended to achieve the following targets:

- POSIX Locale model
- Several languages in one system
- Independence from character set being used
- Independence from storage forms used for localized data

This introduction describes the basics and the technical basis of the micado MLS.

7.1.2.1. Terms

- **Locale Category**: A template for Locale-related data. A Locale Category for a concrete Locale is a Locale Entity.
- **Locale Entity**: A concrete object for a Locale Category. A Locale Entity has a Locale and information concerning the localization of its data, e.g. a Locale Entity for a date with the Locale of Germany; german supports representation of a date with this Locale.
- **Locale**: A combination of territory and language which specifies the localization of externalized data, e.g. time formatting in Germany.
- **Localization**: Process in which externalized data is adapted to a Locale.
- **Externalization**: Process of identification and separation of Locale-related data from a system.

7.1.2.2. Micado Locale Model

Based on the POSIX Locale model, the Locale Categories shown in the table below have been defined.

All Locale Categories inherit a uniform protocol from the class, and a facility for storing to external sources.

Category

Task	Context
Manages Locale Category objects	Date

Task	Context
Date formatting	Messages
Localization of a number-indexed message	Monetary
Formatting of a currency	Numeric
Formatting of a number	Strings
Localization of a character-string-indexed message	Time
Formatting of a time	

7.1.2.3. MLS

In order to realize an MLS, the Locale Category was implemented. This category manages objects from other Locale Categories and can be linked in a tree-like structure, i.e. such a context can have a parent context and child contexts.

A context also has a Locale. If the Locale of a context changes, the following happens:

- all connected child contexts receive the new Locale
- all Locale Entities which are linked to the context (and its child contexts) receive the new Locale

If the Locale of a Locale Entity is changed, the data for this Locale is loaded from an external source and the Locale Entity is initialized with this data. If a localization is now carried out using the Locale Entity, the objects to be localized (date, time, messages, ...) are now localized for the new Locale.

7.1.2.4. Unicode

Unicode is a standardized 16-bit character set and the basis for the new ISO 32-bit character set. The first 256 characters of Unicode are compatible with the ANSI character set, as used by Windows 3.x and Windows 95, for example.

In order to allow tests produced on one platform to be correctly displayed on another platform in a multi-platform environment (e.g. ü, Ö, ß have different codes in IBM code page 437 and ANSI), Unicode has been selected as the character set.

All externalized data is stored in Unicode and converted into the used character set of a target platform when required. For this purpose, there are some files which contain tables for converting native character sets to Unicode.

Storage in Unicode takes place automatically, just like conversion into a native character set.

It should be noted that Unicode is a character set and not a font. This means that no characters will be lost during conversion, but the display thereof depends on the font used on a platform.

7.1.2.5. Storage of Localized Data

Localized data (messages, texts, date and time formats, ...) can be stored into various sources. We provide support for flat files, Smalltalk classes and (optional) storage a relational data base (“micApplication Service - Multi Language Persitence Interface” is required for this storage model, A POM for SQL anywhere is included).

The responsible layer for managing localized data is generic, i.e. independent of the storage form which is used. It is possible to switch between various forms of storage, i.e. you can store your localized data from the Smalltalk classes in files and vice-versa.

7.1.2.6. Locale Names

Not all territory and language identifiers can be used. The and classes check the validity of a Locale.

7.1.2.7. Maintaining Localized Data

The MLS editor is a tool which can be used for maintaining the externalised data in a simple manner. An export / import option for CSV files (Comma Separated Values) enables the data to be interchanged with external parties, for instance translators, by means of spreadsheet programs.

In order to be able to work with the micado MLS, it must first be initialized. The configuration map "micApplication Service – Multi language" must be loaded in order to do this.

If you want to see the initialization results in the Transcript, you should set the commentProgress flag to true.

```
MicMlsSystemInitializer commentProgress: true
```

Now you can read off the status of the initialization in the transcript along with any error messages which have been produced.

7.1.2.8. File-Based Initialization

In order to store the localized data in files, the file-based initialization procedure must be carried out.

You need a text file called "mls.mls" in your VA directory which contains the path to your mls directory (e.g. "c:\mls").

The mls directory must contain a subdirectory called "Locale".

In this directory ("c:\mls\locale" in this case) the MLS system will then store its data.

You can now initialize and create the system from the Frameworks Transcript Menu or by executing the following code in a workspace:

```
MicMlsSystemInitializer createFileBasedSystem
```

If you got old data in your file system, it will be deleted entirely.

If you want to open and access an existing file system you should later execute:

```
MicMlsSystemInitializer initializeFileBasedSystem
```

7.1.2.9. ENVY-Based Initialization

In order to store the localized data in Smalltalk classes into the ENVY library, you can initialize the system from the Framworks Transcript Menu or by executing the following code in a workspace:

```
MicMlsSystemInitializer initializeLibraryBasedSystem.
```

If you wish to save data in development and do not use the mls data editor, you must set the application to store your data in:

```
MicMlsLocaleLibraryManager
defaultControllerClass: <anApplicationClass>.
```

<anApplicationClass> is the class of the ENVY application in which the localized data is stored in classes. For this purpose, a new application must first be created by hand: e.g. .

The controller class can also be set inside the MLS Data Editor.

7.1.2.10. MLS Persitence Interface-”Based Initialization

In order to store the localized data through the ”Multi Language Persitence Interface”, you must have a POM, which is correctly initialized and connected to the database.

Set the POM instance into your MlsDatabaseManager:

```
MicMlsLocaleDBManager pom: aPom
```

You can initialize the system from the Framworks Transcript Menu or by executing the following code in a workspace:

```
MicMlsSystemInitializer initializeDBBasedSystem
```

7.1.3. Methods and Formats

7.1.3.1. MicMlsLocaleCategory

All Locale Entities are identified by their territory, their language and a string. The string is the name of the Locale Entity (Locale name). It is required for unique identification, since several Locale Entities in a category can have the same Locale. Any name can be used, but may not contain any blanks or underscores.

The following methods are the basis for all Locale Categories, but they are only needed when creating Locale Entities manually and not inside the MLS Data Editor :

`MicMlsLocaleCategory >> locale: aLocale`

<aLocale> is which specifies the territory and the language: e.g. language: ‘english’ territory: ‘united kingdom’ or language: ‘german’ territory: ‘germany’ .

If you use a Locale Entity within a MLS context, this Locale Entity will be set to the Locale of the context and should not be modified manually .

`MicMlsLocaleCategory >> localeName: aLocaleName`

A Locale Entity within a context already got its name, because its needed to add it.

`MicMlsLocaleCategory >> localize: anObject`

`MicMlsLocaleCategory >> localize: anObject withKey: aKeyName`

7.1.3.2. MicMlsLocaleContext

`MicMlsLocaleContext >> localize: aLocale`

`MicMlsLocaleContext >> connectSubcontext: aLocaleContext`

`MicMlsLocaleContext >> connectSupercontext: aLocaleContext`

`MicMlsLocaleContext >> delegateLocalizationOf: anObject to: aLocale-Class`

`MicMlsLocaleContext >> delegateLocalizationOf: anObject to: aLocale-Class withKey: aKeyName`

`MicMlsLocaleContext >> delegateLocalizationOf: anObject to: aLocale-Class named: aLocaleName`

`MicMlsLocaleContext >> delegateLocalizationOf: anObject to: aLocale-Class named: aLocaleName withKey: aKeyName`

`MicMlsLocaleContext >> connectLocaleFor: aLocaleClass named: aLocale-Name`

7.1.3.3. Formats

The various format placeholders are described in the appendix. The relevant current value is entered at this position in the format string when localizing. All format variables start with a percent sign (%). Any other characters may be used in the format string as well as the format variables. The percent sign can be displayed as %%.

7.2. MLS and Application Framework

7.2.1. Overview

The Application Framework provides a generic Adapter for accessing the underlying MLS base system which is described in Language Support Base System.

The generic Adapter helps the developer to fetch data from the MLS base system into his views without great effort. The framework will change its behavior only if the developer has included additional MLS Data. In all other cases, the framework will show its default behavior. The Adapter can thus be seen as an additional way to access the MLS base system, which will make some things easier, but it doesn't take away any of the flexibility which the framework offers by default.

7.2.2. Basics

All MLS Data is controlled by the underlying MLS base system. A view which needs this data must access the MLS base system in some way. When using the Application Framework, views are linked with a Connector. This Connector permits access to the functionality of its Domain Obects and Prcesses via their Viewports.

When using MLS, a developer wants to display some kind of text in different languages, to format the output of some kind of object in a specific way or to set a specific mnemonic.

In the Application Framework the source of this MLS Data can be a Viewport or a Domain Object.

7.2.2.1. Roadmap

In order to understand the nature and structure of data which can be supported by an MLS system, you can use the following categories:

- Static strings with no editing
  - Data independent of a model
    - textual decorations e.g.: window title.
      - The key is static and can be translated.
      - This data is just for orientation purposes
    - non textual interaction references e.g.: mnemonics.
      - The key is static and can be translated.
      - This data is just for usability
  - Data related to a model
    - textual interaction references e.g.: Menu item label & accelerator text, button label (radio button label, Toggle Button label)
      - The key is static and can be translated.
      - These labels display functionality which lies in a Domain Process.
    - textual structuring references e.g.: static label for an entry field, group label
      - The key is static and can be translated.
      - These labels show a part of the structure which lies in the Domain Objects (instance variables or relationships to other objects)
    - Hover Help texts or help texts for static texts, text entry fields or buttons
      - The key is static and can be translated.
      - This data explains the structure of a model.
    - logical information e.g.: sorting order in a List
      - The order is static and can be translated
      - Model information is formatted in some way.
- Data which is from the model
    - indirect textual model contents (via external key) e.g.: error message
      - The key is static and can be translated.
      - An error message xxx must be displayed.
    - direct textual model contents : a static text which displays a model content
      - Is not static and can't be translated and the model must offer Locale information (e.g. code page of original data) to display the data in a correct way.
    - appearance of model content (Boolean or rule result)
      - Is static and can be translated.

      - A static string which represents a model true or false return should be displayed.
- Model-internal data
  - textual model defaults e.g.: Default content of variable which contains a string of some kind
    - is static and can be translated.
    - The model itself needs this data to set its default. In this case it is a dynamic default string.
- Dynamic strings with no editing
  - Data which is from the model
    - direct / indirect textual model contents (via external key + parameter) e.g.: dynamic help text
    - The key part is static and can be translated.
    - An error message xxx ('File : ', yyy,' not found!') must be displayed
- Dynamic strings with editing
  - Data which is from the model
    - direct textual model contents e.g.: text entry, Combo Box, List
      - Is non-static and can't be translated and the model must offer Locale information (e.g. code page of original data) to display the data in a correct way.
      - Information might be lost, depending on the storage model (see storage model)
    - formatted non-textual (Date, Time objects) model contents e.g.: text entry, Combo Box, List
      - Is non-static and can be translated. An object must be displayed in a specific format.
- Appearance of model information
  - Data related to a model
    - logical information e.g.: sorting order in a List
    - The order is static and can be translated
    - Model information is formatted in some way.

7.2.2.2. Storage

Storage of non-translatable model data is another problem:

With a storage model for data which can't be translated, there are 3 main possibilities:

- The database uses Unicode and the Mls system uses this Unicode to display the data
- Strings are saved as a collection of integers in which each integer represents a Unicode character
- The database knows the code page in which the data was created. Each string is then transformed into Unicode and from Unicode into the code page of the current system and vice versa. In this case data could be lost if it doesn't exist in the code page of the database.

7.2.2.3. Categories

As mentioned in earlier chapters, the MLS base system offers a set of Locale Categories to handle all this:

- , which can format a Date object in a specified format
- , which can format a Time object in a specified format
- , which contains one message string for each integer MLS key.
- , which can format a Float or Integer object in a specified format
- , which can format a Float or Integer object in a specified monetary format.
- , which contains one string for each string MLS key
- and to display strings which are assembled from static string parts , non-localized object parts and localized object parts

All Locale Categories contain formatting information or keys and Unicode strings, which are used to display static texts or display data in a desired format. They can be saved in classes or files.

In a view, each Control may need a different Locale Category to display its label or fill its content. In most cases you will have groups, like a group of static texts and entry fields which represent a logical group in the view (an address for example).

As described in section Language Support Base System Language Support Base System, the MLS base systems offers a structure called in which you can collect Locale Entities (instances of Locale Categories). Each MLS Context then offers a set of integer or string keys and formatting rules for Date, Time, Float and Integer objects.

All Locale Entities inside a MLS Context are localized to one Locale (e.g. germany, german)

When using the Application Framework MLS Adapter, you bring both things together.

Each Control of a view or a group of Controls is connected to a Model Connection which carries a model and a viewport. This Model Connection can be specified directly or default behavior is used.

Now each Model Connection gets its MLS Context. This can be done by selecting one inside the Domain

Processes Browser, or if none is specified the Model Connection will try to get a MLS Context along a default path.

Now the generic Adapter will try to use the MLS Data inside the Model Connection s to convert keys into strings or values into strings with the formatted content of the value.

For the generic Adapter, an Aspect name for a Control can be a key in the Locale Entities of a MLS Context and a value returned from the model is a candidate for the use of formatting rules.

7.2.3. Integration of the Adapter into the Structure of the Framework

The Adapter is fully integrated into the structure of the Application Framework as described in the chapters above.

Each Control of a view or group of Controls is connected to a Model Connection which carries a model and a viewport. The Model Connection is the organizational unit which brings view, Viewport and model together and, because of this central position, the Framework and the MLS base system come together in the Model Connection .

When defining a Model Connection inside the Domain Processes Browser, the developer can choose from a list of MLS Contexts. Each Control which uses this Model Connection to get its label, content, help text and so on will access the MLS Context which it carries to search for keys or formatting rules.

While testing the application, a developer can view the MLS Context of a Model Connection inside the Connectors Browser. If no MLS Context is specified within the Model Connection , it will try to get an MLS Context from the Default Domain Process Connection of the currently valid Connector or from a Default Domain Process Connection of its parent process Connector. Doing this, all Controls of a whole hierarchy of views can access an MLS Context which belongs to the Connection of the Domain Process whose view is opened first. This would turn the MLS system into an NLS system and switching the MLS Context to a new Locale would then change the appearance of the entire view hierarchy.

Each instance of an MLS Context can be switched independently from one Locale to another, but there is one default Locale for each MLS Context it is switched to when being created.

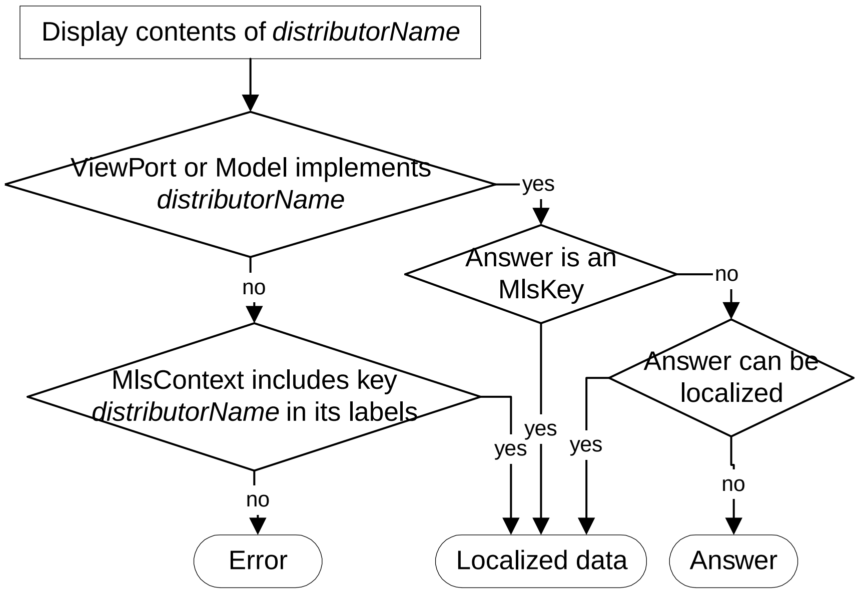
Because of this you have to make sure that an MLS Context is switched to the Locale you want - which may be not the default one - before you finally open the view.

You may reimplement method initializeMLS Context in your Viewport in order to do this before the view is opened.

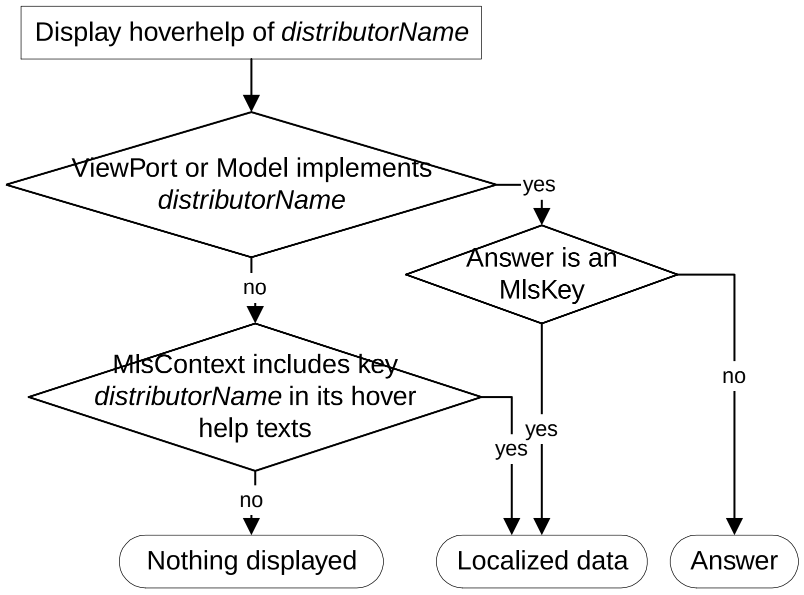
As we have said, the Adapter tries to convert data which comes from a model or looks for MLS keys if no other data can be found for a Control.

The order for getting data for a Control is illustrated with the following three examples:

You want a Text Control (static or editable) to display the Aspect named distributorName.

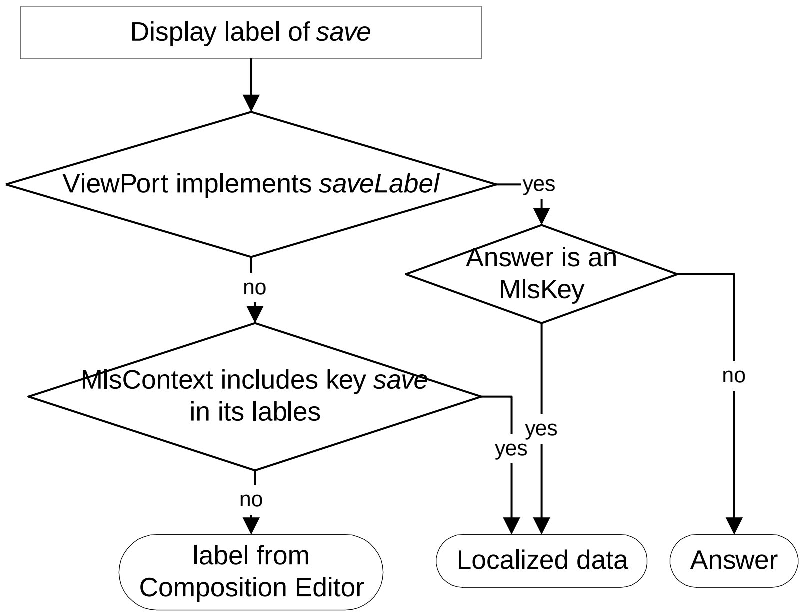

Display contents of *distributorName*
ViewPort or Model implements *distributorName*
yes
no
Answer is an MlsKey
no
MlsContext includes key *distributorName* in its labels
Answer can be localized
yes
yes
yes
no
no
Error
Localized data
Answer

You want the Text Control distributorName to display a HoverHelp.



Display hoverhelp of *distributorName*
ViewPort or Model implements *distributorName*
yes
Answer is an MlsKey
no
MlsContext includes key *distributorName* in its hover help texts
yes
yes
no
no
Nothing displayed
Localized data
Answer

You want the Button Control save to display a Label.



Display label of *save*
ViewPort implements *saveLabel*
yes
Answer is an MlsKey
no
MlsContext includes key *save* in its lables
yes
yes
no
no
label from Composition Editor
Localized data
Answer

To make this generic mechanism work properly, you have to follow some conventions:
The key in the MLS Data must be named like the Aspect name of the Control it offers data for. It makes no difference to the Adapter if a Label, a Hover Help text or a help text has to be shown. (Remember that the general mechanism of the framework looks for special extensions to the Aspect name, like ~Labels, ~HoverHelpText and ~HelpText when searching a method which is responsible).

- The naming convention for format keys which specify a special numeric, monetary, time or date format is as follows:
  - The key is built from the aspect name concatenated with the name of the class the returned object belongs to.
  - Doing this, you can specify multiple formats for one control depending on the model answer.

Example:

```
Key:Format:
#baseNumeric "% .........." (used of localize and delocalize)
#baseNumeric1"% .........." (used for delocalize)
#baseNumeric2"%..........." (used for delocalize)
An object which can be transformed by a Locale Entity of a context (e.g. a Date
object) is always shown in the formatted form unless the Viewport method over-
```

```
rules this process.
All successful or unsuccessful conversions which the Adapter carries out are
written to the Framework Logger at severity level #primitive.
This data can be used to debug the actions of the MLS Adapter in difficult sit-
uations.
```

7.2.4. Transcript Menu Items

The Framework Adapter adds new Menu items to the Transcript Menu. They can be used as follows:

- **Open MLS data editor** opens an editor, which helps to view and edit MLS Data.
- **Multi language tools - initialize library-based system** initializes an MLS storage system based on classes. No Files are required in this case.
- **Multi language tools - initialize file based system** initializes an MLS storage system based on files. The pathname is taken from 'micado.mls'.
- **Multi language tools – remove MLS system** removes the entire MLS systems and clears all caches.
- **Multi language tools - convert *.csv source into the current storage system** is used to convert data which is stored in a *.csv file into the current storage form. This can be either a file or a class.
- **Multi language tools - convert all library data into a new file based system** is used to convert data, which is stored in the currently loaded subclasses of MicMlsDATA, into file based data. During this procedure all old file based data in this path will be deleted, because data of a new file based system is written.

7.2.5. Extension of the Domain Processes Browser

The Domain Processes Browser is the location where a developer adds an MLS Context to a Model Connection .

When defining a Model Connection inside the Domain Processes Browser, the developer can choose from a list of MLS Contexts.

This list contains all subclasses of   . Choosing an MLS Context for a Model Connection means creating a new instance of the MLS Context class inside this Model Connection .Each instance can be localized independently from one Locale to another. If the developer chooses no MLS Context for a Model Connection , this doesn't mean that the Model Connection will not contain any MLS Context. In this case a MLS Context which belongs to the Model Connection of the default Domain Process or the default Domain Process of the Connector of the parent process of the default Domain Process will be filled in.

To undo a situation in which the Model Connection gets its own MLS Context instance, edit the Model Connection and choose 'none' from the list. The Model Connection is then assigned an MLS Context higher up in the hierarchy.

7.2.6. Extension of the Connectors Browser

While testing an application a developer can look at the MLS Context of a Model Connection inside the Connectors Browser. After selecting a Model Connection , the user user can view an MLS Context at different levels:

- View mls context - inspects the context object
- View mls Locale Entities - inspects a collection of all Locale Entities which are connected to the Mls-Context

This facility should only be used to have a quick look at the current keys which are available in a MLS Context at one point in time and to view changes during runtime. You can get more information using the Online Debugger, which displays all Locale Entities, which where used for one Control.

7.2.7. Using the MicMlsKey

The   is a class which can be used to modify or avoid the generic behavior of the Adapter.

When you want to access the MLS base system directly or when you want to realize special conversion behavior for a Control, this   offers an interface to do this.

The   has 6 instance variables and a corresponding API which should be used as follows:

- **value**: When using   or   it contains the key which can be found in the MLS Data. When using   ,   , or   it contains the object which should be transformed (eg a date object). This value must be set in any case.

- **localeType**: Contains the Locale class as symbol (e.g.) which should be used when converting this key.
- **localeName**: Contains the name of the Locale Entity as string which should be used when converting this key.
- **nameExtension**: Alternatively to LocaleName: you can specify the Locale Entity name extension (Labels, HoverHelp,...) and let the mls system search in all corresponding Locale Entities when converting this key
- **connectionName**: Contains the name of the Model Connection of which the MlsContext should be used as string
- **formatKey**: Contains the key which sets the format string when using , , or .

localeType and localeName can be left empty, but you can't fill just one of them. connectionName can be left empty. formatKey can be left empty. In this case a key named ‘default’ is assumed.

If localeType is left empty, it is filled according to the value you choose. If the value is a string, the localeType is set to . If the value is an integer, the localeType is set to .

If localeName is left empty, it is constructed using the value and the context in which this key should be converted. If the value is a string and it is a model or Dispatcher return value, the localeName is set to 'context name' + 'Labels'. If the value is an integer and it is a model or Dispatcher return value, the localeName is set to 'context name'.

If the value is a string and it is a return value from a Hover Help method, the localeName is set to 'context name' + 'HoverHelpText'.
If the value is an integer and it is a return value from a Hover Help method, the localeName is set to 'context name'.

If the value is a string and it is a return value from a help method, the localeName is set to 'context name' + 'HelpText'. If the value is an integer and it is a return value from a help method, the localeName is set to 'context name'.

Examples:

```
aMicMlsKey value: 1
```

Result: The MLS system looks for a message number 1 in the currently valid context

```
aMicMlsKey value: 'model'
```

Result: The MLS system looks for a string with the key 'model' in the currently valid context. The MLS system will use label strings when the is returned from a ~Label method in a viewport or it is a model return. The MLS system will use help text strings when the is returned from a ~HelpText method in a viewport. The MLS system will use Hover Help text strings when the is returned from a ~HoverHelpText method in a viewport.

```
aMicMlsKey value: 1;
localeType: #MicMlsLocaleMessages;
localeName: 'Context1'
```

Result: The MLS system looks for a message number 1 in a Locale Entity named 'Context1' in the currently valid context

```
aMicMlsKey value: 1;
connectionName: 'computer'
```

Result: The MLS system looks for a message number 1 in the context of a Model Connection named 'computer' inside the current Connector. If you want to access a Locale Entity outside a MLS Context, which is needed when setting model defaults for example, you should also use this . In this case you must specify all the information which is required: value, localeType and a localeName. Now you can use the API ‘localizeFor: aLocale’ to convert this key.

```
(aMicMlsKey value: 'defaultName';
localeType: #MicMlsLocaleStrings ;
localeName: 'ComputerDefaults')
localizeFor: aGermanLocaleDescription
```

Result: The MLS system looks for a string with the key 'defaultName' in a Locale Entity named 'ComputerDefaults' which has the Locale specified by aGermanLocaleDescription.

```
(aMicMlsKey value: 123456.78 ;
localeType: #MicMlsLocaleNumeric ;
localeName: 'LocaleNumerics' ;
formatKey: 'modelNumberFloat')
localizeFor: anEnglishLocaleDescription
```

Result: The MLS system looks for a format string called ‘modelNumberFloat’ in a Locale Entity named: ‘LocaleNumerics’ which has the Locale specified by anEnglishLocaleDescription. This format string is then used to transform the Float 123456.78.

7.2.8. Creating an MlsContext

The default procedure for creating a new MLS Context is to add a new subclass of . This can be done from the MLS Data Editor. If the default behavior is acceptable for you, you don't need to do more. In this case the MLS Context gets its default name from its class name and following Locale Entities can be created and added if you want:

```
MicMlsLocaleNumeric named: MlsContext name
MicMlsLocaleMonetary named: MlsContext name
MicMlsLocaleDate named: MlsContext name
MicMlsLocaleTime named: MlsContext name
MicMlsLocaleMessages named: MlsContext name
MicMlsLocaleStrings named: MlsContext name + 'Label'
MicMlsLocaleStrings named: MlsContext name + 'HelpText'
MicMlsLocaleStrings named: MlsContext name + 'HoverHelpText'
```

The default Locale is set to 'united kingdom' 'english'.

7.2.9. The MLS API

This API should be used when accessing MLS Contexts or the MLS system directly (micAPI-mls):

`MicFwViewPort >> contextAtConnectionNamed: aSymbol`

`MicFwViewPort >> localize: aLocale`

`MicFwViewPort >> localizeAllConnectionsTo: aLocale`

`MicFwViewPort >> localizeConnectionNamed: aSymbol To: aLocale`

`MicFwViewPort >> mlsContext`

`MicFwViewPort >> returnContextNamed: aContextName`

`MicFwViewPort >> getLocaleNamed: name ofType: MicMlsLocaleSymbol fromContextAtConnectionNamed: aSymbol`

`MicFwViewPort >> converter`

`MicMlsKey >> localizeTo: aLocale`

7.3. The MLS Data Editor

7.3.1. Overview

The MLS Data Editor is accessible via the Menu item **Open MLS data editor** in the **Frameworks** Menu.

7.3.1.1. Explanation of Lists and Their Functionality

7.3.1.1.1. MLS Contexts

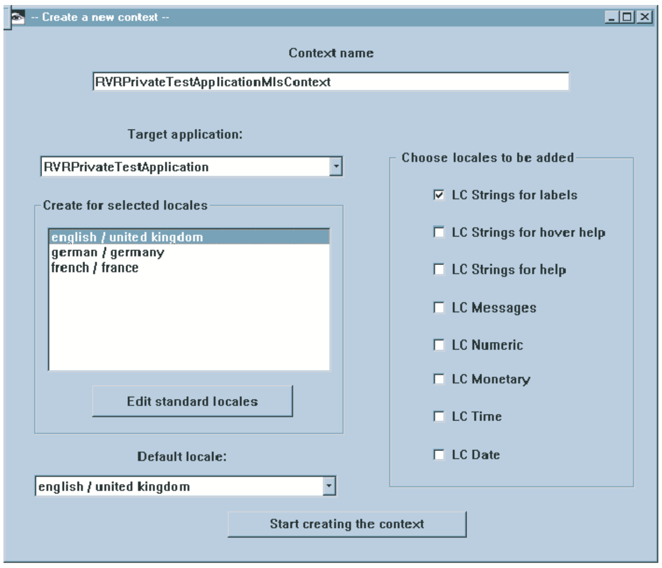
This list contains MLS contexts which store and organize Locale Entities. It includes all MLS context classes to be found under . These classes are broken down into Master Contexts and “normal” contexts.

The context entitled "<all MLS data available>" is not a genuine context; it stands for a view of the MLS Data pool irrespective of contexts.

The Menu items:

7.3.1.1.2. New context

This is used for creating a new context class in the required application, and the marked Locale Entities are generated in the required languages and added to the context.


-- Create a new context --

Context name: RVRPrivateTestApplicationMlsContext

Target application: RVRPrivateTestApplication

Create for selected locales:
- english / united kingdom
- german / germany
- french / france

Default locale: english / united kingdom

Choose locales to be added:
- [✓] LC Strings for labels
- [ ] LC Strings for hover help
- [ ] LC Strings for help
- [ ] LC Messages
- [ ] LC Numeric
- [ ] LC Monetary
- [ ] LC Time
- [ ] LC Date

The individual fields of the view Create a new context have the following meaning:

Context name : The name of the context - a default name is proposed

Target application : The application in which this class is to be defined - the most recently created version of an application is proposed

Choose Locales to be added : The user is able to specify the categories for which new Locale Entities are to be created.

These Locale Entities are given standard names based on the name of the context.

Create for selected Locales : The user is able to specify the Locales (language territory combinations) for which the selected Locale Entities are to be created.

Start creating new context is used to start the process of creating a context. When this process has been completed, this view is closed, and a new context exists in the data editor.

7.3.1.1.3. New master context

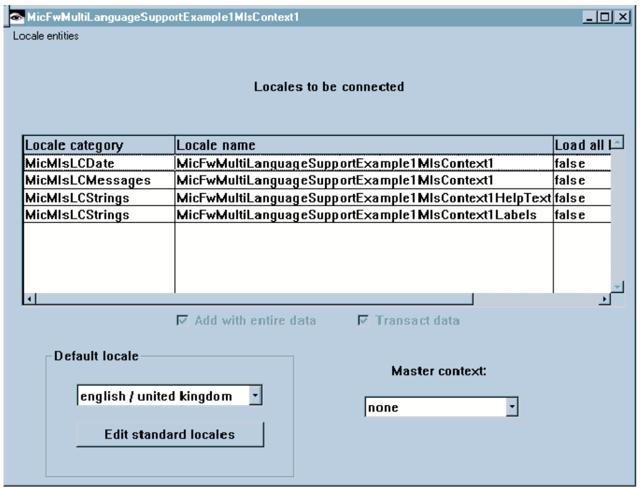
A new Master Context is created as described above; the new Master Context is instanced and stored in LocaleManager when the system is next initialized.

7.3.1.1.4. Delete context

The selected context class is deleted, although the Locale Entities belonging to this context are retained. These have to be deleted individually elsewhere.

7.3.1.1.5. Edit context settings

Settings for the selected context can be made in the following view.


MicFwMultiLanguageSupportExample1MlsContext1

Locale entities

Locales to be connected

Locale category	Locale name	Load all
MicMlsLCDate	MicFwMultiLanguageSupportExample1MlsContext1	false
MicMlsLCMessages	MicFwMultiLanguageSupportExample1MlsContext1	false
MicMlsLCStrings	MicFwMultiLanguageSupportExample1MlsContext1HelpText	false
MicMlsLCStrings	MicFwMultiLanguageSupportExample1MlsContext1Labels	false

[✓] Add with entire data [✓] Transact data

Default locale: english / united kingdom

Edit standard locales

Master context: none

It is possible to define the default Locale which is automatically set whenever the context is instanced.

In addition, the list Locales to be connected contains the Locale Entities which are added to the context during the instancing process if these are present in the data pool.

The entry Load all keys into cache specifies whether, when the context is instanced, all data of its Locale Entities are to be loaded or whether the keys which have already been requested are retained in memory.

This option should be set to false particularly for large data volumes where only a few keys are actually required.

The entry Transacted specifies whether the MLS Data are to be incorporated in the transaction mechanism of Object Behavior Framework.

The Toggle Button below the List of all Locale Entities enables changes to be made for the selected line.

Delete entity and **add entity** and **rename entity** can be used for deleting, creating or renaming the entries for the context. However, changes to these entries only change the reaction of the context during initialization. This means that no Locale Entities are deleted or created, and that only the potential addition of such entities to this context is affected.

The Combo Box **master context** can be used for specifying that a specific Master Context is set for this context when it is initialized as a super context. This is only relevant for working without Application Framework or for specific cases, as the super context can also be set in the Domain Process browser. It must also be borne in mind that any Master Context set at this point overrides all other settings in the Domain Process browser.

A (*) symbol before the name of the context indicates an inconsistency in the Locale Entities which are to be added to this context. In this case, Locale Entities are not available in the data pool for all language territory combinations, and it would not be possible for this context to be switched perfectly to all existing Locales. This inconsistency is not noticed as long as the specific Locale is not used.

7.3.1.1.6. Technical comment:

The instance in the LocaleManager is used for processing a Master Context. However, if a “normal” context is processed, a new instance is created for the editor.

7.3.1.1.7. MLS Categories

This list indicates which categories of Locales are present in the particular context.

With "<All MLS data available> ", the system displays all categories which exist in the current data pool.

7.3.1.1.8. MLS Locale Entities

This list indicates which Locale Entities of the selected category exist in the particular context.

More precisely, the system displays all Locale Entities which could theoretically exist in this context as a result of the entries in Edit context settings, depending on the Locale to which this context is currently connected.

With "<All MLS data available>", the system displays all Locale Entities of the selected category which exist in the data pool.

The Menu items:

7.3.1.1.8.1. Add locale entity

One or more new Locale Entities are created in the current storage system.

The system initially asks for the Locale Category to which the new Locale Entity is to belong; it then prompts a name.The name can either be selected from the list, or a new name is entered via <other>.

When the name has been entered, the system asks for the Locales for which this Locale Entity is to be created. The list displays the standard Locales which can be processed via **Extra -> Edit standard locales**.

And finally, the system asks whether the user wishes to create a new csv file if the system does not find a suitable file in which the data would fit in the csv-search-path.

7.3.1.1.8.2. Delete locale entity permanently

This can be used for permanently deleting a Locale Entity of a specific language territory combination from the current memory system. All data are lost, and these data can only be retrieved in the form of an old version of the class in a library-based system.

7.3.1.1.8.3. Remove locale entity from context

The selected Locale Entity is no longer added to the selected context when this is instanced.

7.3.1.1.8.4. Make locale entity visible in context

A Locale Entity is added to the selected context when the latter is instanced. The data of the Locale are then available in this context.

7.3.1.1.8.5. Store entity in csv

A Locale Entity of a specific language territory combination can be stored in an existing csv file or in a new csv file to be created. If the system does not find an appropriate csv file, the system asks whether it should create a new file and then creates a new file when a new file name has been entered.

7.3.1.1.8.6. Show all csv files for entity

The user is able to view which Locale Entities are stored in which csv files.If the system does not find any csv file, it displays the message no file visible.

7.3.1.1.9. MLS Keys

This list displays all keys which exist in the selected Locale Entity.These keys are integer values for Locale and strings for all other Locales.

A key of a Locale which contains formats, e.g. , identifies a format which is used for displaying an object, e.g. an integer or data in formatted form.

The Menu items:

7.3.1.1.9.1. Add key

Adds a new key to the selected Locale Entity in the memory.

If a key name for a Label ends with 'ACC', this key describes the accelerator for this Label or for the designated button.

7.3.1.1.9.2. Delete key

Deletes the selected key from the Locale Entity in memory.

7.3.1.1.9.3. Rename key

This can be used for renaming a selected key. All data of the various Locales are retained, and are subsequently available under the changed name.

7.3.1.1.9.4. Copy key

The key with all related data is copied to another Locale Entity. The system copies all local data which can be copied in any way. If the target Locale Entity has fewer Locales available, the data for these Locales cannot be copied. However, if the target Locale Entity has all Locales, the copied keys for these Locales are created with empty contents.

This is necessary because a key must always exist for all Locales of a Locale Entity. It is not necessary to enter data for all Locales, but a key exists either for all Locales or for no Locales.

7.3.1.1.9.5. Insert keys from view

Data together with their Aspect already entered in a view can be inserted as keys into the selected Locale Entity.

It’s assumed that the Aspect name follows Application Framework naming convention.

Depending on the Locale Category, only those widget Aspects for which it is normally meaningful to enter MLS Data in such a category are entered as keys. These data may have to be processed further, but represent a good starting point.

7.3.1.1.10. Values of the Chosen Key

This list displays the MLS Data which have been entered for the selected key. These data can be either strings or formats.

The Menu items:

7.3.1.1.10.1. Edit format

This Menu item is accessible only for those Locales which include formats.

An exception in this respect are accelerator keys which end with ‘ACC’. An editor also exists for this case.

In this case, the system opens a mask in which the information for the format can be entered. With accept changes, the format is copied to the selected local entity, and the image of this format is displayed in the testing box.

7.3.1.1.10.2. Add locale

A further Locale is created for the selected Locale Entity. The selected category with the selected name is thus applicable.

This Menu item is used for simply adding a further language territory combination to existing combinations.

7.3.1.1.10.3. Delete locale

This is used for permanently deleting the selected Locale of the selected Locale Entity from the current memory system.

All data of this Locale are lost.

7.3.1.1.11. Browse version changes of locale

In a library based systems you can check the changes you made compared to another version of the selected Locale.

The lower entry field always contains the MLS Data of the Locale which is in <Values of the chosen key>.

This the place for entering and changing the actual MLS Data.

7.3.1.1.12. Further Menu Items

General:

7.3.1.1.12.1. Save all changes

This ensures that all changes to MLS keys and their data which have been made in memory and which have not been saved are permanently written to the current storage system. In general, data which have not yet been saved will not exist as all data are immediately saved when the system changes from input field to another field.

7.3.1.1.12.2. Clear mls cache

All MLS Data stored in memory are deleted and are retrieved from the permanent medium when subsequently requested.

The keys in the Master Contexts are reloaded directly after the cache is deleted. The value of the cached keys in this case is therefore not zero.

7.3.1.1.12.3. Refresh Browser

All browser data are re-initialized in this way.

7.3.1.1.12.4. Convert csv files

Data in csv files can be transferred to the current unicode-based storage system.

7.3.1.1.12.5. Initialize file based system

A file-based system is opened, and data can now be read or written. No data are changed when the system is opened. (For initialization of a file based system a file “mls.mls” in the current VA directory is required, which points to the directory, in which the MLS directory structure exists)

7.3.1.1.12.6. Initialize library based system

An ENVY class based system is opened

7.3.1.1.12.7. Initialize db based system

A database-based system is opened. This means that an attempt is made to establish a Connection with an appropriately prepared database.

7.3.1.1.12.8. View keys with empty values <---> View all keys

This setting defines whether all keys (data) are to be generally displayed, or whether a mask is to be imposed over the data displaying only objects containing empty entries. This option is designed for searching for translation omissions where a key for a Locale has been defined but no data have been entered.

7.3.1.1.13. Options

7.3.1.1.13.1. Set target for locales

In a class-based system, it is possible to specify the application in which the new classes are to be defined.

In a file- or database-based system, the targets are fixed and cannot be changed during run times.

7.3.1.1.13.2. Show csv info <--> Hide csv info

This setting specifies when the system should display for the Locale Entities, and whether (and if so, how many) Locales exist in currently visible csv files. If the csv information is displayed, there are three states in which a Locale Entity may exist:

(all) means that all Locales exist in a csv file.

(some) means that some but not all Locales exist in a csv file.

(none) means that no Locales exist in a csv file.

7.3.1.1.13.3. Update csv files automatically <--> Don't update csv files

This setting specifies whether the csv files viewed by the system are to be automatically changed (or not) in the event of any changes to the unicode data.

7.3.1.1.13.4. Set csv path

The user is able to enter the path to the directory in which his csv files are held. Only the csv files in this directory are visible for the system.

7.3.1.1.14. Extra

7.3.1.1.14.1. Edit standard locales

The user is able to define a quantity of Locales which are suggested when he is working with the editor at those points where one or more Locales have to be specified. For instance, the quantity of all possible language territory combinations is very restricted and the choice is simplified (for storing data a VA dump file is created in the current VA directory).

7.4. Tables

7.4.1. Formats

7.4.1.1. MicMlsLCDate

%bAbbreviated name of month

%BFull name of month

%mMonth as number with leading zero (01-12)

%zMonth as number without leading zero (1-12)

%dWeekday as number with leading zero (01-31)

%eWeekday as number without leading zero. A space is put before single digit figures (1-31)

%fWeekday without leading zero (1-31)

%wDay of week (Sunday is 0).

%ALong name of weekday.

%aShort name of weekday.

%yYear without century (00-99).

%YYear with century

7.4.1.2. MicMlsLCNumeric

%IInteger part of a number (with leading blanks)

%i Integer part of a number (without leading blanks).

%S Decimal separator

%fFraction part

%aAlways signed (+ or -).

%nOnly signed if negative value (-).

7.4.1.3. MicMlsLCTime

%HHour in 24 hour format (00-23).

%IHour in 12 hour format (01-12).

%kHour in 24 hour format without leading zero (0-23).

%iHour in 12 hour format without leading zero (1-12).

%MMinutes (00-59)

%SSeconds (00-59)

%pLocalized AM / PM string.

7.4.1.4. MicMlsLCMonetary

For Monetary, the same format variables apply as for Numeric, plus the following:

%UFull name of currency.

%HShort name of currency.

%YCurrency symbol.

7.5. Examples

7.5.1. Externalize Data and Change Language During Runtime

7.5.1.1. Fundamental Requirements

A completely initialized library-based MLS system is specified (see Application Framework documentation - MLS).

If the user wishes to work with a file-based system, he should initially transfer all data to a new file-based system (Multi-language tools-> Convert all library data into a new file system) and should then initialize such a file-based system.

In a newly installed MLS system, an MLS context entitled StandardMasterContext normally exists as a sub-class of . This context contains some standard data and some prepared Locales, which however do not yet contain any data.

This standard context is used in this example in order to provide some standard Locales in a simple manner.

A new application entitled has been created.

The following classes have been created in this application:

- - a Domain Object. It contains the instance variables: name, firstName, street, city, phone, dayOfBirth
- - a Domain Process
- - a view drawn in the composition editor. In this view, not only all entry fields but also the button labels, the Group Box and all static texts concerning the widget names are connected to Application Framework.

When MLS is used, the data required for this reason are taken from the MLS system without methods having to be written for this purpose.

A new Default Base Connection bearing an object of class is created in the Domain Process browser.

The method is used in the Domain Process for placing a instance in this Default Base Connection.

7.5.1.2. Create an MLS Context

No MLS-specific step has yet been taken. The purpose of all previous steps has been only to connect to Application Framework.

The first step is to create a new MLS context in the example application so that data can be stored in the context.

For this purpose, start the MLS Data Editor (**micFrameworks -> Open MLS data editor**).

A new MLS context is created with the Menu item New context, which can be accessed via the context Menu of the MLS contexts list.

In the following mask, the user can define the name of the new context, the target application and the Locales which are to be added to this context.

The most recently created edition of an application is suggested with a standard name. In this case, the application is: and the name is .

Because we initially intend to add Labels in this example, the setting in Choose Locales to be added is correct.

As the view is to be translated into English and German, the user only needs to select the two Locales (english / united kingdom) and (german / germany) from the list Create for selected Locales and create a new context with Start creating new context.

If the new context is now selected in the data editor, a new Locale Entity entitled is found under the category Strings.

If this is selected, it can be seen that no key has yet been created in this new Locale Entity and that no MLS Data yet exist in this entity.

7.5.1.3. Connection to Application Framework

The purpose of the context which has just been created is to store MLS Data for our example and to subsequently supply Framework with the appropriate translation.

In order to ensure that Framework knows that it should use this context for our example, this is entered in the Domain Process browser.

The model Connection of the default Domain Process is accordingly edited in the Domain Process browser. There are two Combo Boxes which contain MLS-specific data in the mask in which a Connection is defined. The context - is selected in the Combo Box entitled **Mls context name**, and the above-mentioned StandardMasterContext is set as the MLS super-context name , and all changes are saved.

This ensures that the new context subsequently receives the StandardMasterContext as a super context.

By declaring the StandardMasterContext as a super context, we use the inheritance of the standard data format for the newly created context. We are thus not forced to define a separate data format.

7.5.1.4. Externalization of Data

The Labels entered in the view in English are now to be externalized with their corresponding Aspect names.

The newly created - is selected in the list entitled **Mls contexts**.

Strings is selected as the MLS Category, and -
is selected as the Locale Entity, i.e. the strings labels Locale which are generated as standard with the context.

We now intend to externalize the Label data into this Locale Entity.

The ending of the name of the strings Locale Entity (namely 'Labels') specifies for Application Framework that Label data are present.

For each Label, it is necessary to specify a key which is identical to the Aspect name of the corresponding widget, as defined in the view.

Data in various languages can then be entered in relation to every key.

This can be performed automatically for a Locale (language / territory combination) for the Label data which have already been entered.

For this purpose, select the context Menu via the list of MLS keys and select Insert keys from view.

Now select the required view class (in this instance).

The system then asks for the language of the data in the view, i.e. to which Locale this data is to be added. Because the data in the view are in English, we select (english / united kingdom).

When this procedure has been completed, MLS keys have been created for all possible widgets and the data entered for the view have been entered for the Locale (english / united kingdom).

We can now start the process and observe the view in English:

Enter

```
MicMlsExampleProcess1 new openView: MicMlsExampleView1
```

If the data had not been previously externalized, Framework would now search for contents of the connected static texts, which represent a Model Aspect, and would not find these contents; this would result in an error message.

7.5.1.5. Translate the MLS Keys into Another Language

The second language (in this instance: German) is now to be entered:

Every key of the Locale Entity in which we have just inserted the English translations is selected, and the Locale (german / germany) is selected in the List Values of the chosen mls key.

The German translation is now entered in the lower entry field.

All entries are saved immediately, and methods are compiled.

7.5.1.6. Implementing the Language Switch

Functionality is now placed behind the buttons:

For this purpose, four methods are written in the :

germanLocale

```
^ MicMlsLocale language: 'german' territory: 'germany'
```

This method returns a Locale description object which is used in the MLS system for describing a Locale.

englishLocale

```
^ MicMlsLocale language: 'english' territory: 'united kingdom'
```

Supplies the corresponding English Locale.

switchToEnglish

```
self localize: self englishLocale.
```

This method changes the Locale of the context hierarchy to which the button is connected to English.

In this case, this is and the StandardMasteContext.

switchToGerman

`self localize: self germanLocale.`

As above, but all data are subsequently in German.

The process can now be started, and the view can be switched between German and English and vice versa using the two buttons under “Language control".

The format of the date of birth is determined by the format in the StandardMasterContext, as we have not defined a separate format in our new context.

7.5.2. Advanced Exercises

7.5.2.1. Externalized Help Text

A help text is to be entered for the entry field firstName.

For this purpose, we must initially establish a facility in our new context for enabling help texts to be stored.

This new Locale Entity must be able to hold data for our two languages and it is created as follows:

Our new context is selected, and the Menu item **Add locale entity** is reached via the context Menu on the list **MLS locale entities**.

When this Menu item has been selected, the user is asked which Locale Category he would like to create. Help texts are stored in a Locale, and the name must end with help text.

We select the following standard name .

We select the German and the English Locales.

When the Locale Category is being created, the user is asked whether he would also like to save the data in a csv file (in order for instance to permit a simple data interchange process) which contains “readable” non-unicode data or in order to send MLS keys to the translator.

In our example, we answer “no” to the question for creating a csv file and continue.

The editor then contains a Locale Entity of the category Strings names -
.

This Locale Entity is now selected and, using the context Menu on the list MLS keys, we select add key in order to enter a key for our help text.

As the help text is to be applicable for the first name and as the Aspect name of this entry field is first-Name, we create a new key firstName.

When text has been entered under this key for both languages, this part is concluded, and the text is displayed in the first name field when F1 is pressed while the application is running.

7.5.2.2. New Data Format

If the user would like a different format for the date of birth of the person, he could change the date format in the standard Master Context if the standard itself is to be changed.

In this case however, we only wish to change the date format for our view, and thus require a Locale Entity which is able to handle date formats.

Using **Add locale entity**, we create a Locale Entity for English and German under the category
and the name .

This new Locale Entity is now selected, and a new key is added. This can be entitled "default"; in this case, this will be the new default for our view, which is used if no key has been entered for a specific Aspect.

In this case, we wish to specify a separate format for the Aspect dayOfBirth . The name of the new key is accordingly dayOfBirth.

When the new key has been created, select the required language in the list "values of the chosen MLS key" and go to a mask in which the format can be edited via the context Menu -> editFormat.

We choose : "%B %e. %Y" for the English format, and "%d.%m.%Y" for the German format.

The meaning of the format symbols, which all commence with a percent symbol (%), is explained in this mask.

Accept the format with accept changes, and the format is then immediately available in the edited Locale Entity.

When our application is restarted, this new format is used to format the date object which comes from the model. The new format overrides the default format of the StandardMasterContext.

Advanced exercises - use of a *MicMlsKey*

An even more advanced technique will now be used.

The purpose of the model is to specify whether the person is over 18 years of age or not, and an appropriate message should be displayed as help text.

In order to display this variant as well, these texts should be messages instead of help texts.

For this purpose, we initially create Locale Entities for the category for English and German.

Only figures are accepted as a key for this Locale, and we create the message (over 18) under key 1, and the message (not over 18) under key 2.

If these texts are to be capable of being displayed as help texts, it is necessary to write two methods in Viewport and in the model; these are as follows:

In Viewport ( ):

```
DayOfBirthHelpText
"Public - return a MLS key which contains a text to visualize if the person is
over 18 years   of age"
  ^ self model returnAgeKey
```

In the model ( ):

```
ReturnAgeKey
" Public - return a MLS key which informs the reader if the person is over 18"
  ^ ((Date today subtractDate: self dayOfBirth) / 365 asFloat) >= 18
    ifTrue:[ (MicMlsKey new) value: 1 ]
     ifFalse:[ (MicMlsKey new) value: 2 ]
```

These methods ensure that Application Framework receives a object from the model in response to the help text. For Framework, this special object means that a text has to be retrieved from the MLS system. This then represents the actual help text.

This is only one example for the use of the class. This class can be used at various points in order to specifically access data of the MLS system which is not covered by the generic standard response of Application Framework.

7.6. Usage of Locale Categories CompoundMessage and CompoundString

These Locale Categories are used to display strings which are assembled from static string parts , non-localized object parts and localized object parts. The result can be strings like "Date today: 10/9/97" or "Do you want to delete 2 files ?".

In some cases you might want to use plural expressions depending on the value of a parameter. Because of this, it's possible to decide from the value of a parameter if you want to use part A or B, so you can build expressions like "one file" or "2 files" depending on the number of files.

The general data structure of and is as follows:

`MicMlsLCCompoundMessages:`

Integer key <---> Format string

`MicMlsLCCompoundStrings:`

String key <---> Format string

The format string consists of n elements, of which each element can be as follows:

Elements:

- " String part " or
- {Parameter part}. A parameter part can be defined as follows:
  - {n} or
  - {n,o} or
  - {n,choice,t,c[1], . . ,c[n]}

where :

n = parameter number n

lc,tr,ln,nm,ky

o =

lc

{Parts of o :

```
lc = locale category (Numeric, Date, Time)
tr = territory (e.g. "germany" or "*")
ln = language (e.g. "german" or "*")
nm = Name of a locale entity which lies in the context hierarchie the compound
locale is in.
The name can also be "*" which is the first Locale Entity of the Locale Cate-
gory found.
ky = A key which defines the format string used in the Locale Entity defined by
lc,tr,ln,nm.
A key named "*" means the default key.}
```

The Locale Entity defined by o is used to format the parameter n.

"choice" is a fixed key word.

t = Class of the paramter which should be compared (Currently available: "Integer")

c = operator,value,result

{Parts of c:

operator = The operator to compare with (">" , "<" , "<=", ">=" , "=", "<>")

value = A value which must be of type t

result = {e[1] . . e[n]}}

e = An element as defined above (recursive!)

Examples

Some examples for parameter parts and their possible results :

Parameter no. 2: Parameter part: Result:

```
1"Number "{2}"Number 1"
1{2,Numeric}"+ 1.00"
1{2,choice,Integer,=,1,{"A file"},>,1,{"Some files"}}"A file"
```

7.7. Packing the Frameworks using MLS?

Different strategies can be used with MLS packaging. There are basically 2 directions in which MLS packaging can be optimized:

- Minimize initialization time: MLS is initialized before packaging in order to minimize the initialization time.
- Minimize runtime image: MLS is not initialized before packaging in order to minimize the runtime image. In this case the following API methods will be used at runtime to initialize the MLS system:

```
MicMlsSystemInitializer initializeLibraryBasedSystem
MicMlsSystemInitializer initializeFileBasedSystem
```

For runtime initialization with a database system, an initialized POM should be assigned to the MLS Datamanager:

```
#MicMlsLocaleDBManager pom: aPom
```

After this the initialization methods can be used:

```
MicMlsSystemInitializer initializeDBBasedSystem
```

The storage classes for a library-based system are not required in a runtine image if the system is initialized and the cache is filled. If development is library-based and runtime is file based, then none of the storage classes should be packed in the runtime image.

It sometimes makes sense to pack the runtime with a filled viewPortRequestBroker-cache. This is accomplished by running the application in development and save the cache from "MicFwCachingViewPortRequestBroker viewPortCache" in a file. The viewPortCache instance is set again in the class instance in the runtime image and thus an initialized viewPort-cache is packed.

8

Creating Runtime Executables for Distribution

8.1. Introduction

The ultimate goal of any development effort is the creation of a runtime executable for installation on the user’s machines.

Unfortunately, due to the nature of the Smalltalk environment, which treats development and runtime code as one and the same, distributing Smalltalk applications, especially applications based on the framework, appears to be one of the most complicated tasks of developing applications.

Enclose we will offer you a brief description of the process of ”packaging” under Visual Age for Smalltalk™ (version > 4.00) regarding the special considerations to the Framework.

As a first step we will describe the structure of the framework’s, followed by a short description of the IBM Packaging Tools.

The next step will be a chapter on the special considerations to the Framework and the last step will be short overwiew of tips and tricks and common gotcha’s.

8.2. Organization of Application Framework

In this chapter we will offer a brief overview on the logical and technical (ENVY) organization of the Framework. Following we will show the important runtime applications.

8.2.1. Layout of Application Framework Components

Application Framework consists of several components. As a first, rather inexact classification, we can identify the following components:

- Framework Kernel: contains the base functionality, abstract classes, MVC, Controls
- Kernel Extensions: contains components and extensions as
  - Platform Adapter VA
  - Drag & Drop
  - Web
  - Visual Programming

Kernel Extensions will be denotated inside the Framework as Interaction components.

To create a runtime using the Framework the specialized kernel extensions for the target platform must be loaded.

- Framework Services: The term services is used in the means of the frameworks as all those components, thoat offer support for ”usual” purposes,e.g. validation.
- Services can be added to the application during the advanced steps of the implementation to keep track of the basis functionality and add sophisticated or enhanded services later. The Framework offers the following services:
  - Authorisation,
  - Validation,
  - Multi Language,
  - VA Enhancements.
- Development Tools: Tools like the DomainProcess Browser

This logical structure is mirrored in the technical structure of the ENVY Configuration Map. the table below shows the names and contents of the configuration maps (leading micApplication is dismissed).

The configuration maps are separated into Runtime and Development.

Besides this we can define a general rule for the separation of Runtime and Development applications :

Runtime Applications as part of the Application Frameworks are all applications, whose names do not end with Examle, D oder Tools.

In a Runtime Executable you only need applications from the Runtime configuration maps. When preparing your runtime image, you should only load the Fw services you really use within your application.

Most services have initialization parts within their "loaded" code and store new broker entries or other settings globally.

If you load a service you don't really need, you might package a global setting into your runtime image that might lead to a walkback during execution.

Map Name	**Contents**	**Runtime**	**Devel.**
Examples	Examples for the Framework		X
Framework Development	Framework development core		X
Framework Runtime	Framework Kernel	X	
Interaction – Drag & Drop Runtime	Image Drag & Drop Adapter	X	
Interaction - Platform GUI Enhancements Development	Visual programming, mnemonics, custom parts development extensions		X
Interaction - Platform GUI Enhancements Runtime	Visual programming, mnemonics, custom parts runtime parts	X	
Interaction – Platform GUI Runtime	VA Platform Adapter	X	

Map Name	Contents	Runtime	Devel.
Interaction –Web	Web Connection Adapter	X	
Obsolete	Downward-compatibilty-layer.	X	
Service – Authorization	Authorization	X	
Service - Multi Language Development	Mls data editor and development extensions		X
Service – Multi Language Runtime	Multi Language Support runtime parts	X	
Service – Multi Language Persistence Interface Development	Mls Interface persistence development extensions and POM classes		X
Service – Multi Language Persistence Interface	Mls Interface persistence- runtime parts	X	
Service – Validation	Validation-Adapter	X	
Tools Process Browser Development	Domain processes browser development extensions		X
Tools Process Browser Runtime	Domain processes browser runtime parts	X	
Tools Runtime	Fw tools runtime base map	X	

8.2.2. Prerequiste Applications

Component	Application	Usage
Kernel	MicFwViewPorts	Models & Viewports
Kernel	MicFwWindowSystemAdapter	Window Adapter
Kernel	MicVaWorkArounds	Some Fixes in Va
Interaction	MicFwDragAndDrop- FrameworkExtensions	Drag & Drop
Interaction	MicFwVisualProgrammingAdapter	CompEd for visual Framework programming
Interaction	MicFwPartsEsVa	if using some specialized parts
Interaction	MicVAMemonics	if using Memonics (lack in VA - micado extension)
Services	MicFwAuthorization	Authorization
Services	MicFwValidation	Validation
Services	MicMlsFrameworksAdapter	MLS

8.3. The VA Packager

An integrated tool of the Visual Age for Smalltalk™ environment is used for the creation of a runtime executable. This tool, called the packager can be accessed throug different Menu options, but offers under all conditions a similar functionality: creation of a runtime executable.

It can be reached through :

- **VA Organizer:** The way of starting the packager on the organizer is the way of creating a runtime executable mostly acccording to VA's philosophy of usage.First you highlight the application that contains the startup-class and then you have to select the Menu-item **Application -> Make Executable ....**
- You will see a dialog, where you have to define the StartView and the application's prerequisites. VA will start immediately trying to write the runtime-executable, depending on VA'S default-methods of starting a view.
- **Application Manager:** Similar to the procedure of packaging from the Organizer, from the Application Manager you have to highlight the application that contains the startup-class and then you have to select the Menu-item **Application -> Package**.In contrast to the first way you are offered the opinion of starting the packager tool in two ways:
- (Browse Problems then Package...) to see any problems before packaging or
- (Package then Browse the Problems...). to start packaging immediately.
- You can enter the definition of the StartView but the prerequisites have to be defined before.
- **System Transcript:** The transcript Menu tools offers the **item Browse Packed Images**. to start the tool directly. It is your resposibility to enter all necessary options and parameters.

By choosing one of the first two opportunities you can only create "Reduced Runtime Images". The advantage of these proceedings is the automatic generation of startup-methods and setting of parameters.

In general, according to the complexity of applications using the framework, these procedures will not completely fit the application's needs.Therefore you shoud prefer using the way of starting the packager throug the system-transcript.

HINT: The process of packaging should obtain the same results, not depending on from where it was started. In reality, the only chance of a successful completion of the packaging process' will be offered throug startgin from the transcript. All the other ways might end in unpredictable or undesired results.

8.3.1. Packaging Guide

8.3.1.1. Making your Frameworks-Based Application Packager-Compliant

The Packager bases a decision on whether or not to include a specific class in the runtime image on the existence of a reference to that class. Methods will only be included if there are senders of the corresponding message.

Due to the nature of the Frameworks, not all classes in your application will have explicit references, nor will all methods be explicitly invoked. For instance, there are usually no references to Viewport classes. Instead, Viewport classes implement a message named portName that associates them with a Domain Object class. This mechanism, being Framework-specific, is not transparent to the Packager.

The same holds true for certain categories of methods. In Viewports, you can implement methods that do not have explicit senders. Instead, their selectors are created dynamically at runtime; the methods are then invoked by a #perform:. Examples are selectors like #commitEnabled ('commit' being the name of a Domain Process Aspect) or #streetHoverHelpText ('street' being the name of a Domain Object Aspect).

The above indicates that the Packager needs some assistance in making the right decision on what to include in a runtime image. Here´s how :

8.3.1.1.1. Including Classes

The Packager can be forced to include classes, even if they are not referenced, by implementing the message #packagerIncludeClasses in your application class. This method needs to return a collection of the classes to include in the runtime image. For instance, the method

```
MicFwFitnessClubExample class >> packagerIncludeClasses
```

is implemented to include all the classes in that application and its subapplications:

```
packagerIncludeClasses
  ^(OrderedCollection new
    addAll: self defined asOrderedCollection;
    addAll: self extended asOrderedCollection;
    yourself)
  reject: [:each | each inheritsFrom: SubApplication].
```

If you have classes in your application which you need only at development time, you may want to choose a more sophisticated implementation of this method to ensure that they are not included in your runtime image. However, we suggest that you move development classes into an application of their own.

8.3.1.1.2. Including Methods

Similar to the mechanism for explicitly including classes, there is a message that you can implement in order to force the Packager to include the implementors of specific message selectors. This message is called #packagerIncludeSelectors. It has to return a Collection of Symbols, the implementors of which will be included in the packaged image. In MicFwFitnessClubExample class, the message is implemented as :

```
PackagerIncludeSelectors
  (self packagerIncludeClasses)
    inject: OrderedCollection new
      into: [:res :class |
        res addAll: (class allSelectorsFor: self);
        addAll: (class class allSelectorsFor: self)
        res]
```

Again, this implementation is rather crude, as it includes all selectors that have an implementation. You may choose to be more specific here if the need arises.

8.3.1.1.3. Using Symbols

In typical Smalltalk applications, Symbols are used for two different purposes: one is to provide flexibility in what messages are being sent (the Symbols will be ‘performed’ at some time), the other is simply as a string-like data type.

Since the Packager cannot distinguish between these two usages, it always assumes the worst case: every Symbol might be used as a message selector. Consequently, during the packaging process, it will warn you that aSymbol has no implementors, even if you never intended aSymbol to be used as a selector. There is a way to circumvent these warnings though. You will need to implement the message in your application class, returning a Collection of Symbols you do not intend to use as message selectors. For instance, if you use Symbols for colors throughout your application, have #packagerKnownSymbols return #(red green blue white yellow).

The alternative is to use Atoms instead of Symbols in the first place. Atoms behave like Symbols do but are not assumed to be message selectors by the Packager.

8.3.1.2. The Packaging Process

After taking care of what classes and methods to include in your runtime image and how to tell the Packager to ignore the Symbols that are not of interest to it, packaging your runtime code is a straightforward, step-by-step process. Here´s how it works

- Make sure you really included everything your application needs by means of the and methods
- Create a new runtime image: choose (**Tools** → **Browse Packaged Images**) from the **Transcript** Menu. Click on the tab ‘Instructions in Database’ of the following notebook view. Double-click on ‘AbtBaseRuntimePackaging’ in the List on the left of the view and select ‘AbtBaseEpRuntimeImagePackagingInstructions‘. Click the ‘Next’ button near the bottom of the window.
- In the following notebook view, select the application you want to package and press ‘>>’. Then, select the tab ‘Startup Code’. Enter the code to start your application in the field ‘Application Entry Point’. For the example, this is MicFwFitnessClubExample startApplicationRT. Then, press ‘Next‘. Don´t be confused by the progress indicator that comes up next; the reduction process will take far longer than it suggests.
- after the reduction process has finished you should inspect the List of messages it has produced. Press ‘Next’ to get to a screen that allows you to fix potential problems.
- pressing ‘Next’ again will take you to a screen asking you to save the packaging instructions you generated so far.
- another ‘Next’ will bring up the final, ‘Output’ screen. Saving the packaged image again will take longer than the progress bar suggests.

You´re through ! To start your runtime image, simply execute

- abt –iepappl or nodialog -iepappl
- at a DOS prompt. You may of course also create a desktop icon associated with that image.

Appendix A

API

The following is a list of API methods for the Application Framework.

MicFwViewPort>>
<aContentAspectOfAGroupControl>ChildPaneConnectionMode

MicFwViewPort>>
<aContentAspectOfAGroupControl>ChildPaneConnectionMode defaultChildPaneConnection-ModeValueOf: anAspect

MicFwViewPort>>
<aSourceAspect><anOperation>: aListOfViewPorts onto: aViewPortOrNil

MicFwViewPort>>
<aSourceAspect><anOperation><aTargetAspect>: aListOfViewPorts onto: aViewPortOrNil

MicFwViewPort>>
<aspectName>Accelerator

Set the accelerator object of the Control specified by <aspectName>.

MicFwViewPort>>
<aspectName>BackgroundColor

Set the background color of the Control specified by <aspectName>.

MicFwViewPort>>
<aspectName>Checked

Set the checked state of the Control specified by <aspectName>.

MicFwViewPort>>
<aspectName>Editable

Set the editable state of the Control specified by <aspectName>.

MicFwViewPort>>
<aspectName>Enabled

Set the enabled state of the Control specified by <aspectName>.

MicFwViewPort>>
<aspectName>ForegroundColor

Set the foreground color of the Control specified by <aspectName>.

MicFwViewPort>>
<aspectName>HelpText

Set the Framework help text for the Control specified by <aspectName>.

MicFwViewPort>>
<aspectName>HoverHelpText

Set the Framework Hover Help text for the Control specified by <aspectName>.

MicFwViewPort>>
<aspectName>Label

Set the label of the Control specified by <aspectName>.

MicFwViewPort>>
<aspectName>Manipulator

See <defaultManipulatorValueOf: anAspect>.

MicFwViewPort>>
<aspectName>MetaControlListUpdate: aMetaControlList

See <defaultUpdateOfMetaControlList: aMetaControlList forAspect: anAspect>

MicFwViewPort>>
<aspectName>Readable

Set the readable state of the Control specified by <aspectName>.

MicFwViewPort>>
<aspectName>Visible

Set the visible state of the Control specified by <aspectName>.

MicFwViewPort>>
<columnAspectName>EmbeddedPart

Returns an embedded part.

MicFwViewPort>>
<detailsViewContentAspectName>ItemAspects

MicFwTransactionManager>>
abortAll

Abort all contexts.

MicFwDomainProcess>>
abortAndBegin

Abort the transaction of the receiver and all its child processes and begin a new transaction for the receiver and all its child processes.

MicFwDomainProcess>>
abortAndBeginAlone

Abort the transaction of the receiver and all its child processes and begin a new transaction for the receiver only.

MicFwDomainProcess>>
abortAndBeginTouched

Abort the transaction of the receiver and all its child processes and begin a new transaction for the receiver and all its child processes which have ever opened a transaction level.

MicFwDomainProcess>>
abortAndClose

Obsolete – use abortAndCloseProcess.

MicFwDomainProcess>>
abortAndCloseProcess

Abort the transaction and close the receiver and its associated view. If a child process wants to stay open with the parent, but can't abort and close itselfs, do not not abort and do not close the receiver.

MicFwDomainProcess>>
abortAndCloseView

Same as abortAndCloseProcess.

MicFwAbstractPersistenceManagerRdb>>
abortDatabaseTransaction

Rollback any pending database transaction. Trigger associated event and logger.

MicFwTransactionContext>>
abortToTop

Abort all child contexts of receiver and then abort all transaction levels in receiver.

MicFwTransactionContext>>
abortTransaction

Abort highest transaction level in receiver. If TrLevel1 aborted, then abort all child contexts and receiver.

MicFwTransactedObject>>
abortVersion: aMicFwObjectVersion inContext: aTransactionContext

Abort the receiver to the version contained in aMicFwObjectVersion. Since versioned changes aren't written through to the instance variables, there is no need for action here except triggering an event.

Note: If aMicFwObjectVersion is <nil>, this method was called from a #abortToTop of the transaction context (the transactionContext's transactionLevel is 0). Otherwise the transaction context's transactionLevel is already decremented (e.g. if this is the final commit, aTransactionContext transactionLevel is already 0).

MicFwTransactedObject>>
abortVersion: aMicFwObjectVersion inContext: aTransactionContext inRelationship: aMicFwRelationship

Abort the receiver's relationship to the version contained in aMicFwObjectVersion. Default is to do nothing.

MicFwDomainProcess>>
aboutToAbortTransaction

Called when a user wants to close the corresponding view when the transaction of the receiver process is going to be aborted.

MicFwDomainProcess>>
aboutToCloseTransaction: aTransactionCloseMode

Called when a user wants to close the corresponding view. <aTransactionCloseMode> is a boolean which is meant to indicate if an open transaction is to be committed or aborted.

MicFwDomainProcess>>
aboutToCommitTransaction

Called when a user wants to close the corresponding view when the transaction of the receiver process is going to be committed.

MicFwDomainProcess>>
aboutToStartTransaction

Hook called immediately before the transaction is started. It may have been reimplemented for further actions.

MicFwAbstractWindowingPolicy>>
abstractView

Get the Abstract View.

MicFwAbstractWindowingPolicy>>
abstractView: anAbstractView

Set the Abstract View.

MicFwDomainProcess>>
accessesConnectionNamed: aConnectionName

Answer true if the receiver has access to a Connection named <aConnectionName>.

MicFwMetaPart>>
accessorString: aString partClassName: aSymbol

Set the Connection specification <aString> and the associated view class <aSymbol>

MicFwTransactionContext>>
activate

Receiver becomes the single active context.

MicFwDomainProcess>>
activateConnectorWithRefresh

Obsolete. Activate the receivers Model View Connector. Flagged obsolete because it really is a reactivation. Use reactivateViewInteraction for reactivation or openView for any activation

MicFwTransactionContext>>
activateFor: aBlock

Temporarily make the receiver the currently active transaction context while executing <aBlock>. Reset to the previously active context afterwards. Answer the result of <aBlock>

MicFwDomainProcess>>
activateTransactionContextFor: aBlock

Activate the receiver's Transaction Context, if the receiver uses transaction handling.

MicFwDomainProcess>>
addChildProcess: aDomainProcessClass

MicFwDomainProcess>>
addChildProcess: aDomainProcessClass named: aSymbol

MicFwDomainProcess>>
addChildProcess: aDomainProcessClass named: aSymbol withContext: aContext withSuper-Context: aSuperContext

Obsolete. Add new child processes. Use the addNewChildProcess.

MicFwDomainProcess>>
addNewChildProcess: aDomainProcessClass

Add an instance of <aDomainProcessClass> to our Connector and let it be a child of the receiver. The new process instances will be inserted with the default names.

MicFwDomainProcess>>

addNewChildProcess: aDomainProcessClass named: aSymbol

Add an instance of <aDomainProcessClass> to our Connector. If the Connector contains more than one Domain Process of the the same class, use its name <aSymbol> to distinguish them. Let it be a child of the receiver.

MicFwDomainProcess>>

addNewChildProcess: aDomainProcessClass named: aSymbol withContext: aContext withSuperContext: aSuperContext

Add an instance of <aDomainProcessClass> to our Connector. If the Connector contains more than one Domain Process of the the same class, use its name <aSymbol> to distinguish them. Let it be a child of the receiver.

MicFwDomainProcess>>

addPresetChoiceConnectionNamed: aSymbol

Add a preset choice connection to our connector.

MicFwDomainProcess>>

addPresetChoiceConnectionNamed: aSymbol withContext: aContext withSuperContext: aSuperContext

MicFwDomainProcess>>

addPresetHierarchyListConnectionNamed: aSymbol

Add a preset hierarchy list connection to our connector.

MicFwDomainProcess>>

addPresetHierarchyListConnectionNamed: aSymbol withContext:aContext withSuperContext: aSuperContext

MicFwDomainProcess>>

addPresetListConnectionNamed: aSymbol

Add a preset list connection to our connector.

MicFwDomainProcess>>

addPresetListConnectionNamed: aSymbol withContext: aContext withSuperContext: aSuperContext

Add a preset List Connection to our Connector. This Connection will be used from the framework when a internal List Connection is requested for the Aspect <aSymbol>. <aContext> resp. <aSuperContext> are MLS Contexts. If you preset internal List Connections, you can access them with methods like connectionNamed:.

MicFwTransactionContext>>

allCurrentChanges

Answer a Dictionary with all changes on the current level in the receiver. The changed objects appear as keys, their version containers as values

MicFwPersistentObject>>

allPersistentRelationships

alswer a collection with the relationship objects for all persistent relationships in the receiver.

MicFwPersistentObject>>

allPersistentRelationshipsDo: oneArgumentBlock

evaluate the oneArgumentBlock with the relationship objects for all persistent relationships in the receiver.

MicFwApplicationObject>>

allRelationships

Answer a collection with the relationship objects for all relationships in the receiver.

MicFwRdbAbstractDirectStatement>>

allRowsDo: aOneArgumentBlock

Fetch all rows of the receiver and evaluate @aOneArgumentBlock with each row as argument. Close the cursor afterwards.

MicFwPsExtentIdentifier>>

answerSetSize

Answer the rowcount of the answerset described by the receiver from the persistence medium.

MicFwPsExtentIdentifierSQL>>

answerSetSize

Answer the rowcount of the answerset described by the receiver from the persistence medium.

MicFwPsProjectionDescriptor>>

answerSetSize

Answer the rowcount of the answerset described by the receiver from the persistence medium.

MicFwRdbIntegrityViolation>>

answerSetSize

Answer the rowcount of the answerset described by the receiver from the persistence medium.

MicFwNRelationship>>

any

Answer a target object or <nil>. If there are more than one target objects in the collection, it is not ensured, that multiple calls of this method would return the same target object.

MicFwMetaAspect>>

aspect: aValue

Set the Aspect of the model shown in the column.

MicFwPersistenceManager>>

assertConnected

Assert that the receiver is connected to a data source.

MicFwPersistenceManager>>

assertPersistenceValid: aPersistentObject

Assert that <aPersistentObject> is in a state that would allow it to become persistent. Signal a MicFwPs-NotStorable exception otherwise. Subclasses will most likely overload this according to their definition of persistence.

MicFwPersistentObject>>

assertValidForInsertWith: aPersistenceManager

Assert that the receiver is in a state that would allow it to become persistent in @aPersistenceManager. Signal an exception otherwise. The default implementation delegates this to the persistence manager.

MicFwDomainProcess class>>

authorizePerform: anAspect using: aRule

Use <aRule> (Block or Message) to authorize each perform access to <anAspect>.

MicFwDomainProcess class>>

authorizePerformUsing: aRule

Use <aRule> (Block or Message) to authorize each perform access to object itself.

MicFwModelObject class>>

authorizeRead: aspect using: rule

Use <rule> (Block or Message) to authorize each read access to <aspect>. If the access is prohibited real-Value newProhibited will be returned.

MicFwModelObject class>>

authorizeRead: aspect using: rule ifProhibitedReturn: value

Use <rule> (Block or Message) to authorize each read access to <aspect>. <value> is returned if access is prohibited.

MicFwModelObject class>>

authorizeReadUsing: rule

Use <rule> (Block or Message) to authorize each read access to object itself. If the access is prohibited realValue newProhibited will be returned.

MicFwModelObject class>>

authorizeWrite: aspect using: rule

MicFwModelObject class>>

authorizeWriteUsing: rule

Use <rule> (Block or Message) to authorize each write access to object itself or to <aspect>.

MicFwPersistentObject>>

becomeAllLoaded

Activate persistence for the receiver within the current persistence context and set the loaded flag, so that the receiver will not be inserted on the next transaction commit. Do the same for all objects referenced through persistent relationships. NOTE: this call is only valid "inside" an active persistence transaction context.

MicFwPersistentObject>>

becomeAllNewPersistent

Activate persistence for the receiver within the current persistence context. Do the same for all objects referenced through persistent relationships. This call is only valid "inside" an active persistent transaction context.

MicFwPersistentObject>>

becomeAllPersistent

Activate persistence for the receiver within the current persistence context. Do the same for all objects DIRECTLY referenced through persistent relationships. This call is only valid inside an active persistent transaction context.

MicFwPersistentObject>>

becomeLoaded

Activate persistence for the receiver within the current persistence context and set the loaded flag, so that the receiver will not be inserted on the next transaction commit. NOTE: this call is only valid "inside" an active persistence transaction context.

MicFwPersistentObject>>

becomeLoaded: anArray

Activate persistence for the receiver within the current persistence context and set the loaded flag, so that the receiver will not be inserted on the next transaction commit. Do the same for all objects referenced through persistent relationships whose names are in @anArray. NOTE this call is only valid inside an active persistence transaction context.

MicFwPersistentObject>>

becomeNewPersistent

Activate persistence for the receiver within the current persistence context. Set up the receiver to be stored on the next transaction commit. NOTE: this call is only valid "inside" an active persistence transaction context.

MicFwPersistentObject>>

becomePersistent

Activate persistence for the receiver within the current persistence context if necessary. If the receiver is already persistent, make it persistent in the current context's POM. This call is only valid "inside" an active persistence transaction context.

MicFwPersistentObject>>

becomeTotalLoaded

Activate persistence for the receiver within the current persistence context and set the loaded flag, so that the receiver will not be inserted on the next transaction commit. Do the same for all objects referenced through persistent relationships. NOTE: this call is only valid "inside" an active persistence transaction context.

MicFwPersistentObject>>

becomeTotalNewPersistent

Activate persistence for the receiver within the current persistence context. Do the same for all objects directly AND INDIRECTLY referenced through persistent relationships. This call is only valid "inside" an active persistent transaction context.

MicFwPersistentObject>>

becomeTotalNewPersistent: aBlock

Activate persistence for the receiver within the current persistence context. Do the same for all objects directly AND INDIRECTLY referenced through persistent relationships, for which aBlock evaluates to true. aBlock will be evaluated with the receiver object as the argument. This call is only valid inside an active persistent transaction context.

MicFwPersistentObject>>

becomeTotalPersistent

Activate persistence for the receiver within the current persistence context. Do the same for all objects directly AND INDIRECTLY referenced through persistent relationships. This call is only valid inside an active persistent transaction context.

MicFwPersistentObject>>

becomeTransient

Make the receiver non-persistent. This will allow instance data to be manipulated outside of persistent transaction contexts. This call only affects the image object, not any stored data.

MicFwTransactionContext>>

beginTransaction

Begin a new transaction level.

MicFwDomainProcess>>

beNotifiedOfDocumentsClose

Can be overwritten to be notified of last child document close.

MicFwAbstractPersistenceManagerRdb>>

cachedObjectsFor: abstractTargetClassMapping with: foreignKeys from: dependentMapping-Interface

Answer an instance of @abstractTargetClassMapping mappedClasswhose primary keys join with @for-eignKeys for @dependentMappingInterface.

MicFwDomainProcess>>

canAbortAndClose

Obsolete. Called when a user wants to close the corresponding view. Called first for all open children, then for the business object itself. The close action is performed only if the receiver and all open children return true.

MicFwDomainProcess>>

canClose

Called when a user wants to close the corresponding view. This method is first called for all open children, then for the business object itself. The close action is performed only if the receiver and all open children return true.

MicFwDomainProcess>>

canClose: closeInfo

Called when a user wants to close the corresponding view. Called first for all open children, then for the business object itself. The close action is performed only if the receiver and all open children return true in this method. <closeInfo> is an object which contains a closeMode(:) and an info(:). closeMode can be one of COMMIT, ABORT or DEFAULT (System close button pressed or default close command) and info is a user-defined value (default is the Domain Process which has received the close request).

MicFwDomainProcess>>

canCommitAndClose

Obsolete. Called when a user wants to close the corresponding view. Called first for all open children, then for the business object itself. The close action is performed only if the receiver and all open children return true.

MicFwDomainProcess>>

canStartInteractionForAspect: anAspect

Answer if an interaction start is possible. see MicFwViewPort >> canStartInteractionForAspect.

MicFwViewPort>>

canStartInteractionForAspect: anAspect

An event-like notification that the receiver is ready for model-view interaction under aspect <anAspect>. View systems will usually respond by setting the input focus on the corresponing control.

MicFwDomainProcess>>

canStartInteractionForAspect: anAspect inConnectionNamed: connectionName

Answer if an interaction start for the model in connection named <connectionName> or the receiver (if connectionName == nil) is ready for model-view interaction under aspect <anAspect>.

MicFwDomainProcess>>

childProcesses

Answer the child processes of the receiver as a Model Aspect.

MicFwDomainProcess>>

childProcessesAreStarting

Call the starting hook for all child processes. Called in the parent process before any of the child processes is created.

MicFwDomainProcess>>

childProcessFinished: aChildProcess named: aConnectionName withResult: aProcessResult

Notification from a child process which has ended.

MicFwViewPort>>
childViewPortBaseClass

Override to change the base class for list hierarchy child viewPorts.

MicFwViewCachingWindowingPolicy class>>
clearCache

Close and remove all Cached Views.

MicFwRdbAbstractDirectStatement>>
close

Close the receiver by releasing all external resources.

MicFwRdbIntegrityViolation>>
close

Close the receiver by releasing all external resources.

MicFwDomainProcess>>
closeProcess

Close the process and its associated view . If a transaction is running use the <commitTransactionByDefault> method to specify if a commit or abort should happen. If a child process wants to stay open with the parent, but can't close itselfs, do not close the receiver.

MicFwDomainProcess>>
closeRequested: aCloseInfo

Called when a close command enters and can be overridden to modify the values in <aCloseInfo>.

MicFwDomainProcess>>
closeView

Same as closeProcess.

MicFwMetaAspect>>
columnWidth: aValue

Set the initial column width.

MicFwDomainProcess>>
commitAndBegin

Commit the transaction of the receiver and all its child processes and begin a new transaction for the receiver and all its child processes.

MicFwDomainProcess>>
commitAndBeginAlone

Commit the transaction of the receiver and all its child processes and begin a new transaction for the receiver only.

MicFwDomainProcess>>
commitAndBeginTouched

Commit the transaction of the receiver and all its child processes and begin a new transaction for the receiver and all its child processes which have ever opened a transaction level.

MicFwDomainProcess>>
commitAndClose

Obsolete – Use commitAndCloseProcess.

MicFwDomainProcess>>
commitAndCloseProcess

Commit the transaction and close the receiver and its associated view. If a child process wants to stay open with the parent, but can't commit and close itselfs, do not not commit and do not close the receiver.

MicFwDomainProcess>>
commitAndCloseView

Same as commitAndCloseProcess.

MicFwAbstractPersistenceManagerRdb>>
commitDatabaseTransaction

Commit any pending database transaction. Answer true if successful. Signal an exception or answer false otherwise.

MicFwPersistenceManagerRdb>>
commitDatabaseTransaction

Commit any pending database transaction. Answer true if successful. Signal an exception or answer false otherwise.

MicFwPersistenceContext>>
commitImageTransaction

Commit the changes of the current transaction level without notification of the owner (persistence manager). This causes the changes to be committed into the objects, but not into the database.

MicFwTransactionContext>>
commitToTop

Commit all transaction levels of receiver and then commit all child contexts of receiver.

MicFwTransactionContext>>
commitTransaction

Attempt to commit the current transaction level. Let all interested parties prepare for the commit if necessary (#aboutToCommit). If the method answers true, proceed. Go to the next lower transaction level.

MicFwDomainProcess>>
commitTransactionByDefault

When the view is closed without specifying if to COMMIT or ABORT (e.g. system close), this method decides if an open transaction is to be committed (^true) or aborted (^false) by default.

MicFwTransactedObject>>
commitVersion: aMicFwObjectVersion inContext: aTransactionContext

Notify the receiver about a commit of @aMicFwObjectVersion in context @aTransactionContext. Trigger the @commit event on a final commit. The commit has already been completed and @aTransactionContext's transactionLevel is already decremented.

MicFwTransactedObject>>
commitVersion: aMicFwObjectVersion inContext: aTransactionContext inRelationship: aMicFwRelationship

Commit the receiver's relationship @aMicFwRelationship to the version contained in aMicFwObjectVersion.

MicFwPersistenceManagerESQL>>
connectDataSource

Connect to the data source.

MicFwPersistenceManagerOdbc>>
connectDataSource

Connect to the data source.

MicFwDomainProcess>>
connectorActivated

Called after a Connector is activated. It must return true if the view should be updated, false else (recommended if activation is initiated by the user). It may be reimplemented to initialize base models or to perform other actions during activation.

MicFwDomainProcess>>
connectorDeactivated

Obsolete – Use one of the process closing hooks instead.

MicFwPersistenceManagerOdbc>>
connectTo: dataSourceName user: userName password: aPassword

Connect to the ODBC driver with <dataSourceName> <userName> and <aPassword>. Answer true if the connection was established, false otherwise.

MicFwPersistenceManagerESQL>>
connectTo: dbName user: userid password: password

Connect to the data source.

MicFwPersistenceManagerOracle>>
connectTo: dbName user: userid password: password

Connect to the data source.

MicFwPersistenceManagerRdb>>
connectTo: dbName user: userid password: password

Connect to the data source named @dbName as user id @userid with password @password.

MicFwPersistenceManagerESQL>>
connectTo: dbName user: userid password: password mvsSsid: ssid mvsPlan: planName

Connect to the data source.

MicFwPersistenceManagerOdbc>>
connectWithDriverComplete

If the POM is not connected to the database: Opens the NT database connect dialog.

MicFwPersistenceManagerOdbc>>
connectWithDriverPrompt

Connect to the ODBC driver. Let the driver prompt for all applicable connection information, using <dataSourceDescription> values as defaults. Answer true if the connection was established, false otherwise.

MicFwPersistenceManagerOdbc>>
connectWithoutPrompt

Connect to the ODBC driver. It is assumed that sufficient information for the connect is either available <dataSourceDescription> instance variable or through environment settings. No prompting will be done by the ODBC driver. Answer true if the connection was established, false otherwise.

MicFwPersistenceManagerOdbc>>
connectWithUser: userName password: aPassword

Connect to the ODBC driver with <userName> and <aPassword>. any additional information necessary for the connect must be available either from the <dataSourceDescription> or throughthe environment. No prompting will be done by the ODBC driver. Answer true if the connection was established, false otherwise.

MicFwViewPort>>
contextAtConnectionNamed: aSymbol

Return the mlsContext of the Connection named <aSymbol>.

MicFwViewPort>>
converter

See chapter MLS for details.

MicFwApplicationObject>>
convertValues

Make the receiver conform to the type declarations that were made on the class level by sending #as<type> to all declard instance variables. If any conversion fails (i.e. message not understood), the normal Error exception will result. If any conversion results in a value of wrong size, signal a MicFwSizeError.

MicFwAbstractDomainProcessContext>>
copyEnvFromParent: aDomainProcessContext

Copy the DP environment dict from a parent DP context. This method will NOT replace envvars which have already been set.

MicFwTransactionContext>>
couldLock: anAspect in: anObject

Answer true if a lock could be obtained for @anAspect in @anObject (i.e. no other context holds a version), false otherwise.

MicFwAbstractDomainProcessContext>>
createChildProcessContextFor: aDomainProcess

Create a Domain Process context for a child process or return nil to have the child process create its own context.

MicFwDomainProcess>>
created

The receiver has been created as a child process. You may reimplemented it to perform further actions. Called by the parent process immediately after the child process has been created. If the receiver is a static child process, this happens at Connector initialization after creation of all Base Connections, and after the method initializeInvolvedBaseConnections has been called for the parent process. Base Connections will only have valid models if these have been set in that method. Sibling Process Connections may or may not have been created at this moment.

MicFwTransactionContext class>>
current

Answer the current transaction context or nil if none active. This is an alternative to MicFwTransactionManager global currentContext.

MicFwDomainProcess>>
currentChoice

Get the last chosen model.

MicFwTransactionManager>>
currentContext

Return the active context.

MicFwTransactionManager class>>
currentContext

Return active context (or nil).

MicFwPersistenceManagerOdbc>>
dataSourceDescription

Answer the dataSourceDescription for the receiver.

MicFwTransactionContext>>
deactivate

If the receiver is the single active context: Deactivate the receiver (ie, no context is active).

MicFwViewPort>>
defaultAcceptedOperationsValueOf: anAspect

Answer the default acceptable actions for the receiver. the default no action.   acceptable actions are valid drop operations. Actions: ’enableCopy’, ’enableMove’, ’enableLink’.

MicFwViewPort>>
defaultCanCopy: aList valueOf: aSourceAspect

Return the default permit status for a copy operation of the receiver.

MicFwViewPort>>
defaultCanLink: aList valueOf: aSourceAspect

Return the default permit status for a link operation of the receiver.

MicFwViewPort>>
defaultCanMove: aList valueOf: aSourceAspect

Return the default permit status for a move operation of the receiver.

MicFwViewPort>>
defaultCheckedValueOf: dispatcherAspect

Answer the default checked value for the <dispatcherAspect>.

MicFwViewPort>>
defaultChildPaneConnectionModeValueOf: anAspect

MicFwDomainProcess>>
defaultClose

Obsolete.

MicFwViewPort>>
defaultCopy: aList onto: aViewportOrNil valueOf: anAspect

Default operation done if a drop copy operation should be performed by the receiver. The default is do nothing.

MicFwViewPort>>
defaultEditableValueOf: aDispatcherAspect

Answer the default editable value for the <dispatcherAspect>. This is taken from the model that was read finally to retrieve the contents of <dispatcherAspect>.

MicFwViewPort>>
defaultEditableValueOf: aDispatcherAspect

Answer the default editable value for the <aDispatcherAspect>. This is taken from the model that was read finally to retrieve the contents of <aDispatcherAspect>.

MicFwViewPort>>
defaultEmbeddedPartValueOf: dispatcherAspect

Answer the default enabled value for the <dispatcherAspect>.

MicFwDomainProcess class>>
defaultHierarchicalPersistentTransactionPolicyClass

Return the class for the default hierarchical persistent transaction policy for instances of the receiver.

MicFwDomainProcess class>>
defaultHierarchicalTransactionPolicyClass

Return the class for the default hierarchical transaction policy for instances of the receiver.

MicFwDomainProcess class>>
defaultInheritedTransactionPolicyClass

Return the default transaction policy for instances of the receiver.

MicFwViewPort>>
defaultLateModelAccessValueOf: aDispatcherAspect

Answer the default late model access value for the <aDispatcherAspect>.

MicFwViewPort>>
defaultLink: aList onto: aViewportOrNil valueOf: anAspect

Default operation done if a drop link operation should be performed by the receiver. The default is do nothing.

MicFwViewPort>>
defaultManipulatorValueOf: aDispatcherAspect

Answer the default manipulator value for the <aDispatcherAspect>.

MicFwViewPort>>
defaultManipulatorValueOf: anAspect

Pass a valid manipulator, which is evaluated in the Control presenting Aspect <anAspect> with the Window Adapter as argument.

MicFwViewPort>>
defaultMaxSearchLevelValueOf: dispatcherAspect

Answer the default max level to search for models in a hierarchy.

MicFwViewPort>>
defaultMove: aList onto: aViewportOrNil valueOf: anAspect

Default operation done if a drop move operation should be performed by the receiver. The default is do nothing.

MicFwDomainProcess class>>
defaultPersistentTransactionPolicyClass

Return the class for the default persistent transaction policy for instances of the receiver.

MicFwDomainProcess class>>
defaultProcessContextFor: anInstance asChildOf: aParentProcess named: aConName

Return the default process context for a child process <anInstance> as a child of the process <aParentProcess> in the Connection <aConName>. See Process Context.

MicFwViewPort>>
defaultProhibitedValueOf: aDispatcherAspect

Answer the default prohibited value for the <aDispatcherAspect>. This is a value signaling, that <aDispatcherAspect> is prohibited to see.

MicFwViewPort>>
defaultProvidedOperationsValueOf: anAspect

Answer the default processable actions of the receiver. the default no action. Processable actions are drag operations form the receiver. Actions: ’enableCopy’, ’enableMove’, ’enableLink’ .

MicFwViewPort>>
defaultProvidedOperationsValueOf: anAspect

Answer the default processable actions of the receiver. The default is no action. Processable actions are drag operations from the receiver. Actions: 'enableCopy', 'enableMove', 'enableLink' .

MicFwViewPort>>
defaultReadableValueOf: aDispatcherAspect

Answer the default readable value for the <aDispatcherAspect>. This is taken from the model that was read finally to retrieve the contents of <aDispatcherAspect>.

MicFwViewPort>>

defaultShowTouchedValueOf: anAspect

By default show a touched Aspect during a Drag & Drop session.

MicFwDomainProcess class>>

defaultTransactionPolicyClass

Return the class for the default transaction policy for instances of the receiver.

MicFwDomainProcess class>>

defaultTransactionPolicyFor: aChildProcess asChildOf: aParentProcess named: conNameSymbol

Return the default transaction policy for instances of the receiverclass. Normally is <aChildProcess> an instance of the receiver and < aParentProcess > is the parent from <aChildProcess>.

MicFwViewPort>>

defaultUndefined: anAbstractValue

Set <anAbstractValue> which holds a MicFwUndefinedModelObject to the defaults for undefined contents.

MicFwViewPort>>

defaultUpdateOfMetaControlList: aMetaControlList forAspect: anAspect

Override to change the default update behavior of <aMetaControlList> for the currently read Aspect <anAspect>, when a model change has been notified. A return value of nil will cause the Aspect get accessor method for <anAspect> to be called again. Answer the default readable value for the <dispatcherAspect>. This is taken from the model that was read finally to retrieve the contents of <dispatcherAspect>.

MicFwModelObject class>>

defaultValidationRule

The default validation rule (Block or Message) for all aspects of the receiver. .

MicFwViewPort>>

defaultVisibleValueOf: aDispatcherAspect

Answer the default enabled value for the <aDispatcherAspect>. This is taken from the model that was read finally to retrieve the contents of <aDispatcherAspect>.

MicFwDomainProcess class>>

defaultWindowingPolicyClass

Override this method to get an alternate windowing policy class, i.e. an object which Controls the way how Platform Views are created. A reason may be the use of Cached Views for speeding up startup of huge views. A simple view caching windowing policy is provided as an example. The transaction policy defines and maintain the receivers transactional behavior.

MicFwDomainProcess class>>

defaultWindowingPolicyFor: aProcessInstance asChildOf: aParentProcess named: conName viewClass: aViewClassHint parentViewHolder: aDomainProcess modality: aModality

Return the default windowing policy object for instances the receiver.

MicFwDomainProcess class>>

defaultWindowingPolicyForChildOf: aParentProcess named: conName viewClass: aViewClassHint parentViewHolder: aDomainProcess modality: aModality

Return the default windowing policy for instances the receiver.

MicFwDomainProcess>>

deferUpdateInAllViewsWhile: aBlock

Perform <aBlock> leaving the view update in all views, which are connect to top level process, until the end of block execution.

MicFwViewPort>>

deferUpdateInAllViewsWhile: aBlock

Perform <aBlock> leaving the view update until the end of block execution.

MicFwDomainProcess>>

deferViewUpdateWhile: aBlock

Perform <aBlock> leaving the view update until the end of block execution.

MicFwViewPort>>

deferViewUpdateWhile: aBlock

Perform <aBlock> leaving the view update until the end of block execution.

MicFw1Relationship>>

delete

Delete the relationship maintained by the receiver. Also delete the inverse references.

MicFwNRelationship>>

delete

Delete the relationship maintained by the receiver. Also delete the inverse references.

MicFwPersistentObject>>

delete

Register the receiver for a delete from the underlying persistence medium. As aresult, the receiver will get the #isDeleted property. The actual delete will happen on the next transaction context commit. Isolate the receiver from its object net: remove the receiver from all relationships and remove all targets for relationships of the receiver.

MicFwPsExtentIdentifier>>

delete

let the persistence manager delete the extent identified by the receiver from the underlying persistence medium

MicFwPsObjectIdentifier>>

delete

delete the persistent object described by the receiver

MicFwRelationship>>

delete

Delete the relationship maintained by the receiver, i.e. remove all references to objects. Also delete the inverse references

MicFwTrNRelationship>>

delete

Delete the relationship maintained by the receiver. Also delete the inverse references

MicFwPersistenceManager>>

deleteAllInstancesOf: aPersistentClass

Bulk delete all persistent data for <aPersistentClass> from the underlying persistence medium. This should only be used if the sender handles the cleanup of instances in the image. Otherwise use MicFwPersistentObject>>delete

MicFwPersistenceManager>>

deleteIdentifier: aPersistenceIdentifier

delete the persistent object described by @aPersistenceIdentifier

MicFwPersistenceManagerRdb>>

deleteIdentifier: aPersistenceIdentifier

delete the persistent object described by @aPersistenceIdentifier

MicFwPersistenceManager>>

deleteInstancesOf: aPersistentClass where: aQueryArray

Bulk delete all persistent data for <aPersistentClass> from the underlying persistence medium. That conforms to the query <aQueryArray>. Image objects are not considered

MicFwAbstractPersistenceManagerRdb>>

deleteInstancesOf: aPersistentClass where: aQueryExpression

Bulk delete all persistent data for <aPersistentClass> from the underlying persistencemedium. That conforms to the query <aQueryExpression>. Image objects are not considered

MicFwPersistenceManager>>

deleteObject: persistentObject

Delete the persistent data associated with <persistentObject> from the database. For use outside of persistence contexts. This method does not check the dependency lattice and thus depends on the user to make sure that objects are deleted in the right order. Questionable in cases where object is not identified by direct mappings (i.e. identifying relationships)

MicFwPersistenceManagerRdb>>

deleteObject: persistentObject

Delete the persistent data associated with <persistentObject> from the database. For use outside of persistence contexts. This method does not check the dependency lattice and thus depends on the user to make sure that objects are deleted in the right order. Questionable in cases where object is not identified by direct mappings (i.e. identifying relationships)

MicFwAbstractPersistenceManagerRdb>>
deleteObjectOptimistic: persistentObject

Delete the persistent data associated with <persistentObject> from the database. For use outside of persistence contexts. This method does not check the dependency lattice and thus depends on the user to make sure that objects are deleted in the right order. Questionable in cases where object is not identified by direct mappings (i.e. identifying relationships)

MicFwPersistenceManagerRdb>>
deleteObjectOptimistic: persistentObject

Delete the persistent data associated with <persistentObject> from the database. For use outside of persistence contexts. This method does not check the dependency lattice and thus depends on the user to make sure that objects are deleted in the right order. Questionable in cases where object is not identified by direct mappings (i.e. identifying relationships)

MicFwAbstractPersistenceManagerRdb>>
deleteReferencesOf: aPsNRelationship

Delete the relationship references associated with @aPsNRelationship from the database. Do not delete persistent objects!!

MicFwAbstractPersistenceManagerRdb>>
deleteTargetsOf: aPsNRelationship

Delete the target objects referenced by @aPsNRelationship from the database

MicFwAbstractPersistenceManagerRdb>>
deleteWithReferences: persistentObject

Delete the persistent data associated with @persistentObject from the database. Also delete all data referenced through relationships. For use outside of persistence contexts. Objects already loaded in the image are not considered and must be cleaned up by the application

MicFwPersistenceManagerRdb>>
deleteWithReferences: persistentObject

Delete the persistent data associated with @persistentObject from the database. Also delete all data referenced through relationships. For use outside of persistence contexts. Objects already loaded in the image are not considered and must be cleaned up by the application

MicFwPersistenceManager>>
discardGlobally

Remove the receiver from global storage.

MicFwPersistenceManager>>
disconnectDataSource

disconnect from the data source. Subclasses will have to actually terminate the physical connection

MicFwPersistenceManagerRdb>>
disconnectDataSource

Disconnect from the data source

MicFwViewPort>>
enterInteractionForAspect: anAspect comingFromAspect: lastAspect of: lastViewPort

Called if the process is selected via a Notebook Page select event. May be reimplemented to perform some actions with the process. Return true if the actions are successful (possible), false otherwise

MicFwPsExtentIdentifier>>
except: anExtentIdentifier

Combine the queryAnalyzer of this descriptor with the queryAnalyzer of @anExtentIdentifier with the #except operation

MicFwPsProjectionDescriptor>>
except: aProjectionDescriptor

Combine the queryAnalyzer of this descriptor with the queryAnalyzer of @aProjectionDescriptor with the #except operation

MicFwRdbAbstractDirectStatement>>
executeForFetch

Execute the statement without answering result. Leave the cursor open so results can be fetched rowwise

MicFwRdbDirectStatement>>

executeForFetch

Execute the statement without answering result. Leave the cursor open so results can be fetched rowwise

MicFwRdbIntegrityViolation>>
executeForFetch

Execute the statement without answering result. Leave the cursor open so results can be fetched rowwise

MicFwRdbAbstractDirectStatement>>
executeForFetchWith: anArrayOfParameters

Execute the statement without answering result. Leave the cursor open so results can be fetched rowwise. Use the parameters in <anArrayOfParameters> for the embedded parameter markers in the statement

MicFwRdbDirectStatement>>
executeForFetchWith: anArrayOfParameters

Execute the statement without answering result. Leave the cursor open so results can be fetched rowwise. Use the parameters in <anArrayOfParameters> for the embedded parameter markers in the statement

MicFwRdbIntegrityViolation>>
executeForFetchWith: anArrayOfParameters

Execute the statement without answering result. Leave the cursor open so results can be fetched rowwise. Use the parameters in <anArrayOfParameters> for the embedded parameter markers in the statement

MicFwRdbAbstractDirectStatement>>
executeOnce

Execute the statementstring in the receiver once and answer the number of rows affected. Release all resources afterwards.

MicFwRdbDirectStatement>>
executeOnce

Execute the statementstring in the receiver once and answer the number of rows affected. Release all resources afterwards. Answer the number of rows affected.

MicFwRdbIntegrityViolation>>
executeOnce

Execute the statementstring in the receiver once and answer the number of rows affected. Release all resources afterwards.

MicFwAbstractPersistenceManagerRdb>>
executeSQL: anSQLString

Perform <anSQLString> directly against the Database and answer the number of rows affected

MicFwPersistenceManagerRdb>>
executeSQL: anSQLString

Perform <anSQLString> directly against the Database and answer the number of rows affected

MicFwAbstractPersistenceManagerRdb>>
executeSQL: anSQLString with: parameterArray

Perform @anSQLString directly against the Database without answering any results. Use the parameters in @parameterArray

MicFwPersistenceManagerRdb>>
executeSQL: anSQLString with: parameterArray

Perform @anSQLString directly against the Database without answering any results. Use the parameters in @parameterArray

MicFwAbstractPersistenceManagerRdb>>
executeSQLFetchAll: anSQLString

Execute the sql statement @anSQLString and fetch all result rows. Return the answer set as a collection of arrays of data.

MicFwPersistenceManagerRdb>>
executeSQLFetchAll: anSQLString

Execute the sql statement @anSQLString and fetch all result rows. Return the answer set as a collection of arrays of data.

MicFwAbstractPersistenceManagerRdb>>
executeSQLFetchAll: anSQLString with: parameterArray

Execute the sql statement @anSQLString and fetch all result rows. Return the answer set as a collection of arrays of data.

MicFwPersistenceManagerRdb>>
executeSQLFetchAll: anSQLString with: parameterArray

Execute the sql statement @anSQLString and fetch all result rows. Return the answer set as a collection of arrays of data.

MicFwAbstractPersistenceManagerRdb>>
executeSQLForFetch: anSQLString

Answer an sql statement descriptor object that can be used to directly interface with the database. Subclasses will determine which actual class of descriptor is used. The statement will be prepared and executed, the user can start to fetch (via #nextRow) immediately.

MicFwPersistenceManagerRdb>>
executeSQLForFetch: anSQLString

Answer an sql statement descriptor object that can be used to directly interface with the database. Subclasses will determine which actual class of descriptor is used. The statement will be prepared and executed, the user can start to fetch (via #nextRow) immediately.

MicFwAbstractPersistenceManagerRdb>>
executeSQLForFetch: anSQLString with: parameterArray

Answer an sql statement descriptor object that can be used to directly interface with the database. Subclasses will determine which actual class of descriptor is used. The statement will be prepared and executed, the user can start to fetch (via #nextRow) immediately. Use the parameters in <parameterArray>.

MicFwPersistenceManagerRdb>>
executeSQLForFetch: anSQLString with: parameterArray

Answer an sql statement descriptor object that can be used to directly interface with the database. Subclasses will determine which actual class of descriptor is used. The statement will be prepared and executed, the user can start to fetch (via #nextRow) immediately. Use the parameters in <parameterArray>.

MicFwRdbAbstractDirectStatement>>
executeWith: anArrayOfParameters

Execute the statement without answering result. Close the cursor immediatly. Use the parameters in <anArrayOfParameters> for the embedded parameter markers in the statement. Answer the number of rows affected.

MicFwRdbIntegrityViolation>>
executeWith: anArrayOfParameters

Execute the statement without answering result. Close the cursor immediatly. Use the parameters in <anArrayOfParameters> for the embedded parameter markers in the statement. Answer the number of rows affected.

MicFwPsObjectIdentifier>>
existsInDB

Answer @true when the receiver exists in the underlying persistence medium.

MicFwPersistenceManager>>
extentIdentifierFor: aClass

Answer an extent identifier for <aClass>.

MicFwAbstractPersistenceManagerRdb>>
extentIdentifierSQLFor: aClass

Answer an extent identifier sql for <aClass>.

MicFwRdbStmtExecESQL class>>
for: aPersistenceManager withAccessSet: anAccessSetClass

Create an instance of the receiver that communicates with @aPersistenceManager and references an instance of @anAccessSetClass for ESQL access.

MicFwRdbSQLClause class>>
from: aTableArray where: aWhereClause

Answer an instance of the receiver that represents @aWhereClause. The table names (with optional aliases) referenced in @aWhereClause are given in @aTableArray.

MicFwPsExtentIdentifierSQL>>
from: aTableList where: sqlString

Set the sql clause for the receiver.

MicFwPsExtentIdentifier>>

get

Let the persistence manager load the extent identified by the receiver from the underlying persistence medium.

MicFwPsExtentIdentifierSQL>>
get

Let the persistence manager load the extent identified by the receiver from the underlying persistence medium resolved from the cache.

MicFwPsObjectIdentifier>>
get

Load the object identified by the receiver from the underlying persistence medium.

MicFwRdbIntegrityViolation>>
get

Let the persistence manager load the extent identified by the receiver from the underlying persistence medium resolved from the cache.

MicFwAbstractPersistenceManagerRdb>>
getAllInstancesOf: aPersistentClass

Load ALL instances of <aPersistentClass> from the persistence medium. The instances will be resolved from the persistence context cache. Answer an OrderedCollection with loaded objects.

MicFwPersistenceManager>>
getAllInstancesOf: aPersistentClass

Load all instances of @aPersistentClass from the persistence medium. Resolve loaded objects from the internal cache. This can only be executed inside an active transaction context. Answer an OrderedCollection with the requested objects.

MicFwPersistenceManager>>
getAllInstancesOf: aPersistentClass orderBy: anOrderArray

Load all instances of @aPersistentClass from the persistence medium. Resolve loaded objects from the internal cache. This can only be executed inside an active transaction context. Answer an OrderedCollection with the requested objects. Order by @anOrderArray.

MicFwAbstractDomainProcessContext>>
getEnv: var

Get a DP environment variable.

MicFwPersistenceManager>>
getInstancesOf: aPersistentClass where: aQueryArray

Retrieve instances of @aPersistentClass from the persistence medium that conform to the query expression in aQueryArray. Resolve loaded objects from the internal cache. This can only be executed inside an active transaction context. Answer an OrderedCollection with the requested objects.

MicFwPersistenceManager>>
getInstancesOf: aPersistentClass where: aQueryArray orderBy: anOrderArray

Load instances of @aPersistentClass from the persistence medium that conform to the query expression @aQueryArray. Resolve loaded objects from the internal cache. Answer an OrderedCollection with the requested objects. Order by @anOrderArray.

MicFwAbstractPersistenceManagerRdb>>
getInstancesOf: aPersistentClass whereSQL: aWhereClause

Load instances of @aPersistentClass from the persistence medium that conform to the SQL expression in @aWhereClause. Resolve loaded objects from the internal cache. Answer an OrderedCollection with the requested objects. @aWhereClause can be a String or a MicFwRdbSQLWhereClause.

MicFwViewPort>>
getLocaleNamed: name ofType: type fromContextAtConnectionNamed: aSymbol

Return a locale entity from the mlsContext of the connection named aString.

MicFwViewPort>>
getMetaControlListForAspect: anAspect

Return the MetaControl list registered for <anAspect>.If there is none, return nil.

MicFwViewPort>>
getMetaControlListForAspect: anAspect ifAbsent: absenceBlock

Return the MetaControl list registered for <anAspect>. If there is none, evaluate <absenceBlock> and return the result.

MicFw1Relationship>>
getTarget

Returns the target object of the ->1 relationship.

MicFwNRelationship>>
getTarget

Returns a Collection of target objects referenced by the ->N relationship.

MicFwRelationship>>
getTarget

Returns the target object (the actual object or a collection, depending on the type of relationship) of the relationship.

MicFwTr1Relationship>>
getTarget

Returns the transacted target object of the ->1 relationship.

MicFw1Relationship>>
getTargetNoTr

For an non-transacted relationship: Returns the object referenced by the ->1 relationship. For a transacted relationship: Returns the variable value (not the transacted object).

MicFwNRelationship>>
getTargetNoTr

For an non-transacted relationship: Returns the object version of the Collection of objects referenced by the ->N relationship. For a transacted relationship: Returns the variable value of the Collection (not the transacted object).

MicFwRelationship>>
getTargetNoTr

For an non-transacted relationship: Returns the object referenced by the relationship. For a transacted relationship: Returns the variable value (not the transacted object).

MicFwTransactionManager class>>
global

Return single instance of receiver.

MicFwViewPort>>
graphicsPresentation

Get the grahic presentation of an object. Returns an Instance of AbtGraphicsDescriptor.

MicFwAbstractPersistenceManagerRdb>>
handleCompareConflict: cached with: loaded

A compare conflict has occured. Handle the problem accordingly.

Alternatives:

- Refresh the state of the cached object with the state of the loaded object

```
self refresh: cached with: loaded.
^cached
```

- Copy the state of the loaded object into the cached object and answer the cached object:

```
self persistenceCopyTo: cached from: loadedObject.
^cached
```

- Return the cached object without synchronizing (default)

```
^cached.
```

MicFwPersistenceManagerRdb>>
handleDeleteFailed: deleteConflict

The delete conflict @deleteConflict occured. This can happen when the objects optimistic columns have changed or when the object has been deleted. Handle the conflict accordingly. This is the default implementation.

Implemented for backward compatibility: signal the notification to be handled by the application.

MicFwPersistenceManagerRdb>>
handleIntegrityViolation: integrityViolation

Handle the integrity violation @integrityViolation. This is the default implementation: Do nothing (causes a non-resumable IntegrityError to be signaled).

Alternatives:

- integrityViolation repeat.
- integrityViolation skip.(deprecated, can cause inconsistencies between image objects and database state).
- self abortDatabaseTransaction. The integrityViolation might have been raised during a commit of a persistence context. This context is still running. You can decide to commit it again or to abort it.

Implemented for backward compatibility: resignal the notification to be handled by the application.

MicFwPersistenceManagerRdb>>

handleUpdateConflict: updateConflict

The update conflict @updateConflict occured. This can happen when the objects optimistic columns have changed or when the object has been deleted. Handle the conflict accordingly. This is the default implementation.

Implemented for backward compatibility: signal the notification to be handled by the application.

MicFwDomainProcess>>

hasConnectionNamed: aConnectionName

Answer true if the receiver has a Connection named <aConnectionName>.

MicFwTransactionManager>>

hasCurrentContext

Return True if there is a active context.

MicFwRdbForeignKeyDescription>>

hasDeleteCascade

Answer whether deleteRule is CASCADE.

MicFwRdbForeignKeyDescription>>

hasDeleteRestrict

Answer whether deleteRule is RESTRICT or NOACTION.

MicFwRdbForeignKeyDescription>>

hasDeleteSetNull

Answer whether deleteRule is SET NULL.

MicFwDomainProcess>>

hasDynamicChildProcessNamed: aConnectionName

Answer true if <aConnectionName> is the name of a dynamically added child process.

MicFwAbstractPersistenceManagerRdb>>

hasPKUpdateFor: persistentObject

Answer true if at least one of the primary key selectors has a version in the current TA context.

MicFwTransactionManager>>

hasRunningTransaction

Return True if the active context has a TrLevel1.

MicFwDomainProcess>>

hasViewInteraction

Return true if the receiver is involved in view interaction.

MicFwDomainProcess>>

hierarchicalPersistentTransactionPolicyClass

See <defaultHierarchicalPersistentTransactionPolicyClass>.

MicFwDomainProcess>>

hierarchicalTransactionPolicyClass

See <defaultHierarchicalTransactionPolicyClass>.

MicFwMetaPart>>

icon: aIcon

Set the icon.

MicFwDomainProcess>>

inheritedTransactionPolicyClass

See <defaultInheritedTransactionPolicyClass>.

MicFwViewPort>>

initialize

Initialize the receiver. WARNING: This is the initialization of a new instance for the framework. May cause trouble when called during lifetime of the instance in order to perform a reset for example.

MicFwModelObject class>>
initializeAuthorizationTable

Create a new authorization table for the receiver.

MicFwDomainProcess>>
initializeChildProcessConnections

Called after a Connector and the involved Process Connections are created. It may be reimplemented to prepare the Process Model Connections with their process models.

MicFwDomainProcess>>
initializeInvolvedBaseConnections

Called after a Connector and the involved Base Connections are created, but before the child Process Connections are added. It may be reimplemented to prepare the Base Model Connections with their base models.

MicFwDomainObject class>>
initializeValidation

Class method that sends all validateWrite:using: messages for all instance variables of the class that should be validated.

MicFwModelObject class>>
initializeValidationTable

Create a new validation table for the receiver. This may be reimplemented. It is called during initialization of the class. The validation table must be filled with associations between aspects (Symbols) and validation rules (Block or Message) that return a boolean.

MicFwDomainProcess>>
innerAbortTransaction

Aborts a transaction without any further considerations.

MicFwDomainProcess>>
innerCommitTransaction

Commits a transaction without any further considerations.

MicFwDomainProcess>>
innerStartTransaction

Starts a transaction without any further considerations.

MicFwPersistenceManager>>
insertObject: persistentObject

Insert the persistent data associated with <persistentObject> into the database. For use outside of persistence contexts. This method does not check the dependency lattice and thus depends on the user to make sure that objects are stored in the right order.

MicFwPersistenceManagerRdb>>
insertObject: persistentObject

Insert the persistent data associated with <persistentObject> into the database. For use outside of persistence contexts. This method does not check the dependency lattice and thus depends on the user to make sure that objects are stored in the right order.

MicFwPersistenceManager>>
instanceName

Answer the <instanceName> under which the receiver is kept.

MicFwAbstractWindowingPolicy>>
interceptAbortClose

Abort closing the view. Return the 'abort close' value. Default is to return the request to the caller.

MicFwAbstractWindowingPolicy>>
interceptClosePlatformView

Close the Platform View for the connected Abstract View. Default is just to return the request to the Abstract View.

MicFwAbstractWindowingPolicy>>
interceptCommitClose

Commit closing the view. Return the 'commit close' value.Default is to return the request to the caller.

MicFwAbstractWindowingPolicy>>
interceptOpenPlatformView

Open the Platform View for the connected Abstract View. Default is just to return the request to the Abstract View.

MicFwAbstractWindowingPolicy>>
interceptPrepareClose

Prepare closing the view. Default is to return the request to the caller.

MicFwPsExtentIdentifier>>
intersect: anExtentIdentifier

Combine the queryAnalyzer of this descriptor with the queryAnalyzer of @anExtentIdentifier with the #intersect operation.

MicFwPsProjectionDescriptor>>
intersect: aProjectionDescriptor

Combine the queryAnalyzer of this descriptor with the queryAnalyzer of @aProjectionDescriptor with the #intersect operation.

MicFwPersistentObject>>
invalidateRelationships

Send #invalidate to all persistent relationships of the receiver. This causes the relationships to be resolved from the persistence medium the next time they try to get their target(s). All versions for these relationships are removed from transaction versioning mechanism.

MicFwRelationship>>
is1ToN

Answer whether the receiver describes a 1-to-N relationship.

MicFwTransactionContext>>
isActive

Answers True if receiver is the single active context.

MicFwDomainProcess class>>
isAuthorizationActive

Answer true if authorization for instancevariables is switched on for the receiver.

MicFwPersistenceManager>>
isConnected

Answer whether the receiver is currently connected to a persistence medium.

MicFwPersistenceManagerOdbc>>
isConnected

Returns true if the POM is connected to the database.

MicFwPersistenceManagerRdb>>
isConnected

Answer whether the receiver is currently connected to a persistence medium.

MicFwPersistentObject>>
isCreated

Answer whether the receiver has been scheduled for a database create operation.

MicFwViewPort class>>
isDefaultViewPort

This method and the method <portName> decide whether or not to use for a ViewPort.

MicFwPersistentObject>>
isDeleted

Answer whether the receiver is to be deleted from the persistence medium on the commit of the persistence context. This will answer true if the receiver has been explizitly or implizitly sent the #delete message. This will answer false again, after the receiver has actually been deleted.

MicFwPersistentObject>>
isDeletedInPom: pom

Answer whether the receiver is to be deleted from the persistence medium on the commit of the persistence context. This will answer true if the receiver has been explizitly or implizitly sent the #delete message. This will answer false again, after the receiver has actually been deleted.

MicFwDomainProcess class>>
isExecAuthorizationActive

Answer true if authorization for actions (e.g. buttons) is switched on for the receiver.

MicFwDomainProcess class>>
isExecValidationActive

Answer true if validation for actions (e.g. buttons) is switched on for the receiver.

MicFwViewPort>>
isInterestedInChangesOf: anAspect

Answer true if the receiver should propagate each change of <anAspect> to the model. A change can be a single keystroke within a text Control, but not actions like focus changes.

MicFwViewPort>>
isInterestedInUpdatesOf: anAspect

Answer true if the receiver should propagate each update of <anAspect> to the model. An update can be initiated by a user interaction causing a focus change for the current Control.

MicFwTransactionContext>>
isIsolated

Answers True if receiver mode is isolated (not uncommitedRead).

MicFwPersistentObject>>
isLoaded

Answer whether the receiver has been loaded from a persistence medium.

Object>>
isMicFwPersistentObject

Answer whether the receiver is a persistent object, i.e. can be made persistent on request.

MicFwRelationship>>
isMicFwRelationship

Answer whether the receiver is a relationship object.

MicFwRelationship>>
isNToM

Answer whether the receiver describes a N-to-M relationship.

MicFwTransactionContext>>
isolate

Set the mode of the receiver to #isolate. No uncommited changes from other transaction contexts will be visible through this context.

MicFwDomainProcess>>
isOpen

Obsolete. Use isViewOpen.

MicFwPersistentObject>>
isPersistenceModified

Answer true if the receiver has been modified within the current persistent transaction in a way that will cause a store operation to the persistence medium on the final transaction commit.

MicFwPersistentObject>>
isPersistent

Answer the #persistent property.

MicFwPersistentObject>>
isPersistent

Answer the #persistent property.

Object>>
isPersistent

Answer whether the receiver is in a persistent state, i.e. has been made persistent.

MicFwPersistentObject>>
isPersistentIn: aPersistenceManager

Answer whether the receiver is persistent within @aPersistenceManager.

Object>>
isPersistentIn: aPersistenceManager

Answer whether the receiver is persistent within @aPersistenceManager.

MicFwRelationship>>
isPrimitive

Answer whether the receiver is a primitive (one-sided) relationship object.

MicFwTransactionContext>>
isRunning

Answer whether the receiver is in an uncommited state.

MicFwDomainProcess class>>
isTechnicalDomainProcess

A technical domain process can be a browser process, which uses the Application Framework to implement a part of the development environment. The default is a user domain process which is part of a custom application.

MicFw1Relationship>>
isTo1

Answer whether the receiver describes a to-1 relationship.

MicFwRelationship>>
isTo1

Answer whether the receiver describes a to-1 relationship.

MicFwNRelationship>>
isToN

Answer whether the receiver describes a to-N relationship.

MicFwRelationship>>
isToN

Answer whether the receiver describes a to-N relationship.

MicFw1Relationship>>
isValid

Answer true if the receiver conforms to the declared cardinality constraints, false otherwise.

MicFwNRelationship>>
isValid

Answer true if the receiver conforms to the declared cardinality constraints, false otherwise.

MicFwRelationship>>
isValid

Answer true if the receiver conforms to the declared cardinality constraints, false otherwise.

MicFwDomainProcess class>>
isValidationActive

Answer true if validation for instance variables is switched on for the receiver.

MicFwDomainProcess>>
isViewOpen

Answer the state of our modelViewConnector. True means the view is present to the user.

MicFwViewPort>>
itemViewPortBaseClassForAspect: anAspect

Override to change the base class for item viewPorts. <anAspect> is the Aspect of the list content as returned by the receiver.

MicFwAbstractPersistenceManagerRdb>>
iteratorClass

Answer the class to be used for persistence iterations on MicFwPersistentObject collections.

MicFwAbstractPersistenceManagerRdb>>
iteratorClass

Answer the class to be used for persistence iterations on MicFwPersistentObject collections.

MicFwAbstractPersistenceManagerRdb>>
iteratorClass

Answer the class to be used for persistence iterations on MicFwPersistentObject collections.

MicFwAbstractPersistenceManagerRdb>>
iteratorOn: aPersistentClass where: aQueryArray

Answer an iterator capable of loading instances of <aPersistentClass> from the persistence medium that conformto the query expression in aQueryArray.

MicFwAbstractPersistenceManagerRdb>>
iteratorOn: aPersistentClass whereSQL: aSQLString

Answer an iterator capable of loading instances of <aPersistentClass> from the persistence medium that conform to the SQL-Where-Clause in aSQLString.

MicFwPsObjectIdentifier>>
keyValues

Answer the receivers key values (OrderedCollection). RETURN VALUE OrderedCollection.

MicFwPsObjectIdentifier>>
keyVariables

Answer the receiver’s key variables (<OrderedCollection> of <Symbol>).

MicFwMetaPart>>
label: aLabel

Set the initial label (can be also done with the proper framework protocol).

MicFwViewPort>>
leaveInteractionForAspect: anAspect goingToAspect: nextAspect of: nextViewPort

Called if the process is selected via a Notebook Page select event. May be reimplemented to perform some actions with the process. Return true if the actions are successful (possible), false otherwise.

MicFwViewPort>>
listHierarchyParentViewPort

Return the parent Viewport for the receiver as list hierarchy item.

MicFwPsExtentIdentifier>>
load

Let the persistence manager load the extent identified by the receiver from the underlying persistence medium.

MicFwPsExtentIdentifierSQL>>
load

Let the persistence manager load the extent identified by the receiver from the underlying persistence medium.

MicFwPsObjectIdentifier>>
load

Load the object identified by the receiver from the underlying persistence medium.

MicFwPsProjectionDescriptor>>
load

Let the persistence manager load the extent identified by the receiver from the underlying persistence medium.

MicFwRdbIntegrityViolation>>
load

Let the persistence manager load the extent identified by the receiver from the underlying persistence medium.

MicFwAbstractPersistenceManagerRdb>>
loadAllInstancesOf: aPersistentClass

Load instances of <aPersistentClass> from the persistence medium. The instances will NOT be resolved from the context cache. Thus this can be used outside of transaction contexts. Answer an OrderedCollection with loaded objects.

MicFwPersistenceManager>>
loadAllInstancesOf: aPersistentClass

Load ALL instances of <aPersistentClass> from the persistence medium. Answer an OrderedCollection with loaded objects.

MicFwPersistenceManager>>
loadAllInstancesOf: aPersistentClass orderBy: anOrderArray

Load all instances of @aPersistentClass from the persistence medium. The instances will NOT be resolved from the context cache. Thus this can be used outside of transaction contexts. Answer an OrderedCollection with the requested objects. Order by @anOrderArray.

MicFwPersistenceManager>>
loadInstancesOf: aPersistentClass where: aQueryArray

Load instances of @aPersistentClass from the persistence medium that conform to the query expression in aQueryArray.The instances will NOT be resolved from the context cache. Thus this can be used outside of transaction contexts. Answer an OrderedCollection with the requested objects.

MicFwPersistenceManager>>
loadInstancesOf: aPersistentClass where: aQueryArray orderBy: anOrderArray

Load instances of @aPersistentClass from the persistence medium that conform to the query expression in aQueryArray.The instances will NOT be resolved from the context cache. Thus this can be used outside of transaction contexts. Answer an OrderedCollection with the requested objects. Order by @anOrderArray.

MicFwAbstractPersistenceManagerRdb>>
loadInstancesOf: aPersistentClass whereSQL: aWhereClause

Load instances of <aPersistentClass> from the persistence medium that conform to the SQL expression in @aWhereClause. The instances will NOT be resolved from the context cache. Thus this can be used outside of transaction contexts. Answer an OrderedCollection with the requested objects. @aWhereClause can be a String or a MicFwRdbSQLWhereClause.

MicFwViewPort>>
localize: aLocale(MLS)

Localize the MLS Context hierarchy of the Connection belonging to this Viewport to aLocale.

MicFwViewPort>>
localizeConnectionNamed: aString to: aLocale

Localize the mlsContext of a connection named aString to aLocale.

MicFwRdbForeignKeyDescription>>
manager

Answer the receiver's persistence manager.

MicFwAbstractPersistenceManagerRdb>>
mappingFor: aClass

Answer the classMapping for aClass.

MicFwAbstractPersistenceManagerRdb>>
mappingFor: aClass ifNone: aBlock

Answer the classMapping instance for aClass. Execute <aBlock> if not there.

MicFwAbstractPersistenceManagerRdb>>
mappingFor: selector inClass: aClass

Answer the variable mapping for @selector in @aClass. Answer nil, if not there.

MicFwPersistenceManagerRdb>>
mappingFor: selector inClass: aClass

Answer the variable mapping for @selector in @aClass. Answer nil, if not there.

MicFwRdbSQLClause>>
markNotToBeCompiled

Set that the receiver needs not to be compiled before an SQL statement can be generated (Receiver does not contain parameter markers or is already compiled).

MicFwRdbSQLClause>>
markToBeCompiled

Set that the receiver needs to be compiled before an SQL statement can be generated.

MicFwModelObject class>>
MicFwModelObject >> permitWriteDuringReadWhile: aBlock

MicFwPersistenceManager>>
minimalCopy

Answer a new instance of the receiver, that shares most of the receiver's state, such as: core configuration ready to use. The following is also shared (by default) but may be exchanged to allow instance-specific state: instanceName - statementExecutor (DB connection) - cache.

MicFwDomainProcess>>
mlsContext

Returns the MLS Context of the default Domain Process of the current Connector.

MicFwAbstractWindowingPolicy>>
modality

Get the modality.

MicFwDomainProcess class>>
modality

Valid return values are: MicFwSystemModal, MicFwFullApplicationModal, MicFwPrimaryApplicationModal nil (no modality). The default implementation returns nil. To change this behavior you have to overwrite this method.

MicFwAbstractWindowingPolicy>>
modality: aModality

Set the modality.

MicFwViewPort>>
model

This method returns the Domain Object which this Viewport works on.

MicFwViewPort>>
modelAtConnection: aConnectionName

Answer the model of the Connection named <aConnectionName>. The connection that the viewport belongs to uses the connection.

MicFwViewPort>>
modelAtDefaultBaseConnection

Return the Domain Object which lies in the Default Base Connection of the current Connector.

MicFwViewPort>>
modelAtDefaultProcessConnection

Return the Domain Object which lies in the Default Process Connection of the current Connector.

MicFwDomainProcess>>
modelChosen: aConnection

Called after a model is chosen (enter, double click) in a List by the user. <aConnection> is a internal List Connection.

MicFwDomainProcess>>
modelItemChosen: aConnection

Called after a model item is silently chosen (e.g. by item Menu request) in a List by the user. It may be reimplemented to work on the selected base model(s). <aConnection> is an internal List Connection.

MicFwDomainProcess>>
modelSelected: aConnection

Called after a model is selected (cursor, click) in a List by the user. It may be reimplemented to work on the selected base model(s).

MicFwDomainProcess>>
modelViewConnector

Answer the model view connector. The connector hold all base connections and child processes connections (to all domain objects).

MicFwPersistenceManagerOdbc>>
named: aString

Selects a POM (with name aString) as the persistence manager.

MicFwPersistenceManager class>>
named: aString

Get the instance of the recieiver that is associated with <aString> the name can also be a name of a subclass-pom.

MicFwPersistenceManager class>>
named: aString ifAbsent: aBlock

Get the instance of the recieiver that is associated with <aString> the name can also be a name of a subclass-pom, or the result of evaluating <aBlock>.

MicFwDomainProcess>>
nameInParent

Answer the connection name of the connection in the parent process which holds the receiver.

MicFwViewPort>>
newMetaControlListForAspect: anAspect

Create and return a new empty MetaControl list registered for <anAspect>.

MicFwPersistenceManagerOdbc>>
newPersistenceContext

Creates a new persistent context.

MicFwDomainObject>>
newPersistent

Creates a persistent instance of the DO.

MicFwPersistentObject class>>
newPersistent

Create an instance of the receiver. The instance is immediately persistence-aware. This method can only be called inside a running persistent transaction context. An exception will be signaled if this is not adhered to.

MicFwAbstractWindowingPolicy>>
newPlatformViewFor: anAbstractViewClass

Create and return a new Platform View. Has to be reimplemented in each subclass.

MicFwAbstractPersistenceManagerRdb>>
newProjectionDescriptor

Answer a projection descriptor.

MicFwPersistenceManager>>
newTransactionContext

Create a new <code>TransactionContext</code> with the receiver as its owner and the global TransactionManager as its manager. This context isn’t activated automatically but can be activated by sending #beginTransaction or #activate to the context.

MicFwTransactionContext>>
newTransactionContext

Create new context as child of receiver.

MicFwTransactionManager>>
newTransactionContext

Create a new <code>TransactionContext</code> with this manager as its owner and manager. This context isn't activated automatically but can be activated by sending #beginTransaction to the context.

MicFwRdbForeignKeyDescription>>
newWithAccessSet: anAccessSetClass

Create an instance of the receiver that communicates with @aPersistenceManagerand references an instance of @anAccessSetClass for ESQL access.

MicFwRdbAbstractDirectStatement>>
nextRow

Fetch the next row from the receiver. Answer an array with column values.

MicFwRdbIntegrityViolation>>
nextRow

Fetch the next row from the receiver. Answer an array with column values.

MicFwDomainProcess>>
openView

Open the view specified by the <viewName> method which can be set by defining a default view in the Domain Processes Browser.

MicFwDomainObject>>
openView: aViewClass

Opening a view for a Domain Object without using a special Domain Process will result in running under a generic Domain Process. In order to open the view use this method.

MicFwDomainProcess>>
openView: aViewClass

Open a view of the class <aViewClass> for the Domain Process.

MicFwDomainProcess>>
openView: aViewClass withParentProcess: aDomainProcess

Open a view of class <aViewClass> for the receiver as a child view of the parent view from <aDomain-Process >. If the receiver is open, activate our Connector.

MicFwDomainProcess>>
openViewWithParentProcess: aDomainProcess

Open the receiver's view. Use the answer of method <viewName> as the receiver's view class as a child view of the parent view from<aDomainProcess>. If the receiver is open, activate our Connector.

MicFwAbstractPersistenceManagerRdb>>
optionCompare

Answer whether any loaded object will be compared to the cached representation when instances are read (explicitly or on relationship traversal).

MicFwAbstractPersistenceManagerRdb>>
optionCompare: aBoolean

Set the value of option #compare. See getter method for details.

MicFwAbstractPersistenceManagerRdb>>
optionReadLimit

Answer the read limit for retreival operations. This limit is used on object retreival operations. The value 0 signifies no limit. This is currently not supported. This option has no effect.

MicFwAbstractPersistenceManagerRdb>>
optionReadLimit: anInteger

Set the read limit for retreival operations. This limit is used on object retreival operations. The value 0 signifies no limit. This is currently not supported. This option has no effect.

MicFwNRelationship>>
orderAscending

Define the receiver to be sorted on the attributes named in @arrayOfSelectors in ascending order. The attributes are assumed to be base attributes (not relationships).

MicFwPsExtentIdentifier>>
orderAscending: arrayOfSelectors

Define the receiver to be sorted on the attributes named in @arrayOfSelectors in ascending order. The attributes are assumed to be base attribute symbols for the extent target class (not relationships).

MicFwNRelationship>>
orderBy: selectorsOrClause

Define the sort order for the receiver. @selectorsOrClause can be either an array of selectors, optionally postfixed with #asc or #desc designators, or a previously created MicFwOrderClause. Answer the orderClause.

MicFwPsExtentIdentifier>>
orderBy: selectorsOrClause

Define the receiver to be sorted on the attributes named in @arrayOfSelectors. For each attribute the sort order can be defined separately by adding #asc or #desc, e.g. #(#name asc number #desc).

MicFwPsProjectionDescriptor>>
orderByIndexes: indexesOrClause

Define the receiver to be sorted on the indexes named in @indexesOrClause. For each index the sort order can be defined separately by adding #asc or #desc, e.g. #(1 asc 2 desc). Only INTEGER-Values are supported.

MicFwNRelationship>>
orderDescending: arrayOfSelectors

Define the receiver to be sorted on the attributes named in @arrayOfSelectors in descending order. The attributes are assumed to be base attributes (not relationships).

MicFwPsExtentIdentifier>>
orderDescending: arrayOfSelectors

Define the receiver to be sorted on the attributes named in @arrayOfSelectors in descending order. The attributes are assumed to be base attribute symbols for the extent target class (not relationships).

MicFwPsExtentIdentifierSQL>>
parameterAt: aName put: aValueObject

Store @aValue for the parameter named @aName.

MicFwRdbIntegrityViolation>>
parameterAt: aName put: aValueObject

Store @aValue for the parameter named @aName.

MicFwEnhancedQuery>>
parameterAt: aNameSymbol put: aValue

Store @aValue for the parameter named @aNameSymbol.

MicFwPsExtentIdentifier>>
parameterAt: aNameSymbol put: aValueObject

Store @aValue for the parameter named @aNameSymbol.

MicFwPsProjectionDescriptor>>
parameterAt: aNameSymbol put: aValueObject

MicFwDomainProcess>>
parent

Answer with the parent process from the receiver.

MicFwAbstractWindowingPolicy>>
parentView

Get the parent view.

MicFwAbstractWindowingPolicy>>
parentView: anAbstractView

Set the parent view.

MicFwDomainProcess>>
performAbortAndClose

Enforce close of transaction and corresponding view of the receiver and all children. Transactions are aborted. A permission to close is neither from the receiver nor it's children asked.

MicFwDomainProcess>>
performClose

Enforce close of transaction and corresponding view of the receiver and all children. Transactions are closed as returned by #commitTransactionByDefault. No permission is asked via the canClose protocol.

MicFwDomainProcess>>
performCommitAndClose

Enforce close of transaction and corresponding view of the receiver and all children. Transactions are committed. No futher permission is asked via the canClose protocol.

MicFwDomainProcess>>
permitsPerformAccessTo: aspect

Answer the authorization to perform <aspect>.

MicFwModelObject>>
permitWriteDuringReadWhile: aBlock

Set the permission flag for write during traced read while <aBlock> is executed. Please use <writeDuringReadIgnored>.

MicFwAbstractPersistenceManagerRdb>>
persistenceCompare: aPersistentObject

Compare the values of @aPersistentObject with the persistent representation. Signal a MicFwPsCompareConflict or a MicFwPsDeletedConflict for any changes. Else signal nothing and answer self.

CAUTION: You must not perform any persistent relevant actions (db-accesses) in the exception handler blocks!!

MicFwAbstractPersistenceManagerRdb>>
persistenceCompare: oneObject to: anotherObject

Compare the value equivalence of two objects. Answer true if equal, false otherwise. This implementation uses low level read operations on the objects (i.e. #instVarNamed:) and thus bypasses the transaction mechanism. The state compared is the one that existed when the objects were loaded.

MicFwPersistentObject>>
persistenceCompareTo: loadedObject

Compare the receiver to <loadedObject>, which was just loaded from the persistence medium and determined to be of the same identity as the receiver. If the objects are different, signal a MicFwPsCompareConflict.

MicFwPersistentObject>>
persistenceCopyFrom: loadedObject

Copy all persistence relevant data from <loadedObject>, which was just loaded from the persistence medium and determined to be of the same identity as the receiver but different persistent data. Copy all relevant data into the receiver. The default implementation lets the presistence manager handle the job.

MicFwAbstractPersistenceManagerRdb>>
persistenceCopyTo: oneObject from: anotherObject

Copy the persistent data from <anotherObject> to <oneObject>. The copy is done using basic accessors, and thus affects the basic state of the object copied to. Any changes inside the current transaction are still kept and will be commited if not discarded. This method only needs to be reimplemented if any persistent object that is manipulated through the receiver keeps the default implementation of MicFwPersistentObject>>persistenceCopyFrom: and a MicFwPsCompareConflict is signaled and not handled.

MicFwPersistentObject>>
persistenceDelete

The persistent representation of the receiver is about to be deleted from the persistence medium. Overriding this methods allows subclasses to perform any relevant actions!

MicFwPersistentObject>>
persistenceIdentifier

Answer the receiver's persistence identifier within the current persistence manager.

MicFwAbstractPersistenceManagerRdb>>
persistenceIdentifierFor: aClass

Answer a persistence identifier for <aClass>. The ID will have the format appropriate to retrieve data from the underlying persistence medium.

MicFwPersistenceManager>>
persistenceIdentifierFor: aClass

Answer a persistence identifier for <aClass>. The ID will have the format appropriate to retrieve data from the underlying persistence medium.

MicFwPersistenceManager>>
persistenceIdentifierFrom: aPersistentObject

Answer a persistence identifier derived from @aPersistentObject. The identifier can be used to reload @aPersistentObject from the underlying persistence medium.

MicFwPersistentObject>>
persistenceInitialize

Protected - Perform any initializations that should be done after the receiver has newly become persistent. This method will be overriden by subclasses that need to perform persistence- relevant initializations!

MicFwPersistentObject>>
persistenceInsert: aVersionContainer

Protected - the persistent representation of the receiver is about to be inserted with the version data in <aVersionContainer>. Overriding this methods allows subclasses to perform any relevant actions (e.g. generating primary keys)!

MicFwPersistentObject>>
persistenceLoaded

Protected - Perform any initializations that should be done after the receiver has been loaded from the persistence medium. This method will be overriden by subclasses that need to perform persistence-relevant initializations!

MicFwDomainProcess>>
persistenceObjectManager

Obsolete - Answer the receiver's persistence object manager. It must be reimplemented if the receiver wants to use a persistent transaction context. Better handle this setting the correct transaction policy object.

MicFwPersistentObject>>
persistenceUpdate: aVersionContainer

Protected - the persistent representation of the receiver is about to be updated with the version data in

<aVersionContainer>. Overriding this methods allows subclasses to perform any relevant actions!

MicFwDomainProcess>>
persistentTransactionPolicyClass

See <defaultPersistentTransactionPolicyClass>.

MicFwMetaPart>>
positionHint: aPositionHint

Set the position hint. The position hint specifies the position of the page relativ to the static Notebook Pages. The position towards other dynamic pages is defined by the position of the metapart within the metapart collection. The position hint is zero based.

MicFwMetaAspect>>
positionHint: aValue

Set the position relative to the static columns in the view. The position towards other dynamic columns is determind by the position of the meta Aspect within the collection.

MicFwDomainProcess>>
prepareAbort

Flag the transaction to be aborted. Used to change requested transaction close mode for the receiver.

MicFwDomainProcess>>
prepareChildForInteraction: aChildProcess

A view has requested interaction with a child ofthe receiver. Ensure that the child process is in the state to perform its task.

MicFwDomainProcess>>
prepareCommit

Flag the transaction to be committed. Used to change requested transaction close mode for the receiver.

MicFwAbstractPersistenceManagerRdb>>
primaryKeysColumnsFor: aPersistentClass

Answer primary key columns for @aPersistentClass Answer a Dictionary with keys = TableNames and values = (OrderedCollection with: ColumnNames). Only the primary key columns are mentioned.

MicFwDomainProcess>>
processClosed

Called after the process has disconnected its view. Can be overridden to perform cleanups or exit the system.

MicFwDomainProcess>>
processContext

Return the process context.

MicFwDomainProcess class>>
processContextEvaluator

Return the process context evaluator for instances the receiver. An evaluater encapsulate an action, e.g. an access to an instance variable.

MicFwAbstractDomainProcessContext>>
processFinished

The corresponding Domain Process has been closed. May be overridden to perform cleanups.

MicFwDomainProcess>>
processResult

Return something to be passed to the parent as result.

MicFwPsExtentIdentifier>>
query

Answer the query that was constructed from the input array, or an empty query.

MicFwPsProjectionDescriptor>>
query

Answer the query that was constructed or nil.

MicFwPsExtentIdentifier>>
queryStream

Answer a query stream which can be used to dynamically construct the query expression in the receiver.

MicFwPsProjectionDescriptor>>

queryStream

Answer a query stream which can be used to dynamically construct the query expression in the receiver.

MicFwDomainProcess>>
reactivateConnector

Obsolete. Use reactivateViewInteraction instead.

MicFwDomainProcess>>
reactivateView

Reactivate the receivers view.

MicFwDomainProcess>>
reactivateViewInteraction

Reactivate the receivers Model View Connector.

MicFwViewPort>>
readAccessRejectedForAspect: anAspect vetoValue: aProhibitedModel

Override this method to control the reaction on a rejection of a read access to the model.

MicFwDomainProcess>>
readModelAtConnection: aConnectionName

Answer the model of the Connection named <aConnectionName>.

MicFwViewPort>>
readOnlyAspect: anAspect

Set Aspects essentially read only. These Aspects can only be displayed on the view.

MicFwPersistentObject>>
refresh

Refresh the persistent instance variables of the receiver by loading and comparing the values from the persistence medium. Triggers and handles MicFwPsDeletedConflict or MicFwPsAspectCompareConflict when neccessary.

MicFwAbstractPersistenceManagerRdb>>
refresh: persistentObject

Refresh the persistent instance variables of @persistentObject by loading the associated data from persistence and comparing the values. Signal adequate exceptions in case of inconsistencies.

The following inconsistencies are signalled:

- Concurrent changes in persistence and current transaction context.
- Object deleted from persistence.

CAUTION: You must not perform any persistent relevant actions (db-accesses) in the exception handler blocks!!

MicFwPersistenceManager>>
refresh: persistentObject

Refresh the persistent instance variables of @persistentObject by loading the associated data from persistence and comparing the values. Signal adequate exceptions in case of inconsistencies.

MicFwAbstractPersistenceManagerRdb>>
refresh: persistentObject with: otherObject

Refresh the persistent instance variables of @persistentObject by comparision with the values of @otherObject. CAUTION: otherObject and persistentObject should be instances of the same class (e.g. reloading the same object). Signal adequate exceptions in case of inconsistencies. The following inconsistencies are signalled: 1. concurrent changes in persistence and current transaction context.

MicFwDomainProcess>>
refreshAllViews

Refresh all views connected to domain processes in the complete process hierarchy.

MicFwDomainProcess>>
refreshView

Refresh the receivers view. Enforce an update of the view with the actual model contents.

MicFwAbstractPersistenceManagerRdb>>
registeredClauseAt: aName

Answer the clause stored as @aName. The result is of type MicFwPsRegistrableObjectInterface. NOT FOR PUBLIC USAGE. The API use of this method is deprecated. Use one of the more concrete API methods instead, e.g. registeredExtentAt: or registeredStatementAt:.

MicFwAbstractPersistenceManagerRdb>>
registeredClauseAt: aName ifAbsent: aBlock

Answer the clause stored as @aName. The result is of type MicFwPsRegistrableObjectInterface. NOT FOR PUBLIC USAGE. The use of this method is deprecated for public usage. Use one of the more concrete API methods instead, e.g. registeredExtentAt:ifAbsent: or registeredStatementAt:ifAbsent:.

MicFwAbstractPersistenceManagerRdb>>
registeredExtentAt: aName

Answer the registered object-query-extent stored as @aName. The result is of type MicFwPsExtentIdentifier.

MicFwAbstractPersistenceManagerRdb>>
registeredExtentAt: aName ifAbsent: aBlock

Answer the registered object-query-extent stored as @aName. The result is of type MicFwPsExtentIdentifier.

MicFwAbstractPersistenceManagerRdb>>
registeredExtentSQLAt: aName

Answer the registered SQL-query-extent stored as @aName. The result is of type MicFwPsExtentIdentifierSQL.

MicFwAbstractPersistenceManagerRdb>>
registeredExtentSQLAt: aName ifAbsent: aBlock

Answer the registered SQL-query-extent stored as @aName. The result is of type MicFwPsExtentIdentifierSQL.

MicFwAbstractPersistenceManagerRdb>>
registeredOrderAt: aName

Answer the registered order clause stored as @aName. The result is of type MicFwRdbOrderClause.

MicFwAbstractPersistenceManagerRdb>>
registeredOrderAt: aName ifAbsent: aBlock

Answer the registered order clause stored as @aName. The result is of type MicFwRdbOrderClause.

MicFwAbstractPersistenceManagerRdb>>
registeredProjectionAt: aName

Answer the registered object-query-projection stored as @aName. The result is of type MicFwPsProjectionDescriptor.

MicFwAbstractPersistenceManagerRdb>>
registeredProjectionAt: aName ifAbsent: aBlock

Answer the registered object-query-projection stored as @aName. The result is of type MicFwPsProjectionDescriptor.

MicFwAbstractPersistenceManagerRdb>>
registeredStatementAt: aName

Answer the registered direct statement stored as @aName. The result is of type MicFwRdbDirectStatement.

MicFwAbstractPersistenceManagerRdb>>
registeredStatementAt: aName ifAbsent: aBlock

Answer the registered direct statement stored as @aName. The result is of type MicFwRdbDirectStatement.

MicFw1Relationship>>
releaseTarget

Remove the committed target from the receiver without any transacted changes.

MicFwNRelationship>>
releaseTarget

Remove the committed target from the receiver without any transacted changes.

MicFwRelationship>>
releaseTarget

Remove the committed target from the receiver without any transacted changes.

MicFwPersistentObject>>
reload

Answer the a new loaded instance of the receiver’s class which has the same persistence identifier as the receiver.

MicFwPersistenceManager>>

reloadInstance: persistentObject

Reload a copy of @aPersistentObject from the persistence medium.

MicFwNRelationship>>

removeAll

Obsolete. Remove all objects from the relationship maintained by the receiver. Also delete the inverse references.

MicFwDomainProcess>>

removeChildProcess: aChildProcess

Remove the child process <aChildProcess> from the receiver process.

MicFwDomainProcess>>

removeChildProcessNamed: aConnectionName

Remove the child process at the Connection named <aConnectionName> from the receiver process.

MicFwViewPort>>

removeMetaControlListForAspect: anAspect

Remove the MetaControl list registered for <anAspect>. If there is none, do nothing.

MicFwNRelationship>>

removeOrder

The order clause for the handled object was removed. Remove the sorted order from the target collection.

MicFwTransactionContext>>

removeVersionsFor: anAspect in: anObject

Remove all version information for <code>anAspect</code> in <code>anObject</code> in all transaction levels. Information will also be removed in child contexts. @param anObject the object containing the desired aspect @param anAspect the aspect for which to remove version information.

MicFwTransactionContext>>

removeVersionsFor: anObject

Remove all version information accumulated for <code>anObject</code>. Information will als be removed in child contexts. @param anObject the object for which to remove version information @return true when a version has been removed in one of the contexts.

MicFwRdbIntegrityViolation>>

repeat

Set the receiver to be retry when the active handler has terminated (on the next lower level).

MicFwPersistenceManager>>

resetCache

Reset the internal cache object.

MicFwPersistentObject>>

resetRelationships

Send #resetLoaded to all persistent relationships of the receiver. This causes the relationships to be resolved from the persistence medium the next time they try to get their target(s).

MicFwDomainProcess>>

resetTransactionClose

Remove the flag which enforces how the transaction is to be closed.

MicFwViewCachingWindowingPolicy>>

resetViewForCache

Override to reset the view to be cached into its initial state.

MicFwAbstractPersistenceManagerRdb>>

resolveFromCache: aPersistentObject

Try to find an object that is key equal to <aPersistentObject> from the POMs object cache.

MicFwViewPort>>

resolveKey: aMicMlsKey forLocale: aLocale

Resolve a mlsKey using its information for aLocale.

MicFwAbstractPersistenceManagerRdb>>

resolveNetFromCache: aCollection

Resolve the object net whose root objects are in @aCollection from the cache and rebuild the relationships.

MicFwAbstractPersistenceManagerRdb>>
resolveRootFromCache: anObject

Resolve the object net whose root object are is @anObject from the cache and rebuild the relationships.

MicFwViewPort>>
returnContextNamed: aContextClassSymbol

Return a context instance named aContextName.

MicFwTransactionContext class>>
running

Answer the currently running transaction context or nil if none. This is an alternative to MicFwTransactionManager global runningContext.

MicFwTransactionManager class>>
runningContexts

Return the running contex (or nil).

MicFwRdbIntegrityViolation>>
secondObject

Answer the second object that caused the error.

MicFwDomainProcess>>
selectAspect: anAspect

Obsolete. Bring its groupControl to the front. Use startInteractionForAspect: instead.

MicFwDomainProcess>>
selectProcess

Obsolete - bring its groupControl to the front. Use startInteractionForAspect: #defaultProcess instead.

MicFwAbstractDomainProcessContext>>
setDomainProcess: aDomainProcess

Set the DP controlled by the receiver.

MicFwAbstractDomainProcessContext>>
setEnv: var to: value

Set an DP environment variable. Setting an envvar to nil means that it is removed.

MicFwPsProjectionDescriptor>>
setQuery: aBlock

Set the query to @aBlock.

MicFwPsExtentIdentifier>>
setQuery: aQuery

Set the query that specifies the extent. @aQuery can be an Array of Symbols (old query syntax) or a Block (extended query syntax). Answer the receiver if successful. Signal an exception otherwise.

MicFw1Relationship>>
setTo: anObject

Create the relationship connection between @anObject and #thisObject. Validate the type of @anObject and the connection, if the receiver's #validateType is <true>. Delegate to connection handler.

MicFwPsObjectIdentifier>>
setValueFor: keyObject to: aValue

Set the value for the specified key.

MicFwViewPort>>
shouldEmulateEditable

This method turn on the emulation of the editable protocol for widgets which don't implement this behaviour by themselves.

MicFwRdbIntegrityViolation>>
skip

Set the receiver to be resumed when the active handler has terminated (on the next lower level).

MicFwAbstractPersistenceManagerRdb>>
sqlDescriptorFor: anSQLStatement

Answer a sql statement descriptor object that can be used to directly interface to SQL. Subclasses will determine whcih actual class of descriptor is used.

MicFwPersistenceManagerRdb>>
sqlDescriptorFor: anSQLStatement

Answer a sql statement descriptor object that can be used to directly interface to SQL. Subclasses will determine whcih actual class of descriptor is used.

MicFwAbstractPersistenceManagerRdb>>
sqlDescriptorFor: anSQLStatement withParameters: anArray

Answer a sql statement descriptor object that can be used to directly interface to SQL.

MicFwPersistenceManagerRdb>>
sqlDescriptorFor: anSQLStatement withParameters: anArray

Answer a sql statement descriptor object that can be used to directly interface to SQL. Subclasses will determine whcih actual class of descriptor is used.

MicFwDomainProcess class>>
start

Obsolete. Start the receiver and answer the created instance.

MicFwDomainProcess>>
startingChildProcess: aChildProcess

Called in the parent process when a dynamically created child process is started. The default implementation calls the starting hook (see above) for the new child process.

MicFwViewPort>>
startInteraction: aConnection

Obsolete. Use enterInteractionForAspect:comingFromAspect:of: instead.

MicFwViewPort>>
startInteractionForAspect: anAspect

An event-like notification that the receiver is ready for model-view interaction under Aspect <anAspect>. View systems will usually respond by setting the input focus on the corresponding Control.

MicFwDomainProcess>>
startInteractionForAspect: anAspect inConnectionNamed: connectionName

This method should be called if the process is ready to activate model-view interaction under Aspect <anAspect>. Using this method, <anAspect> can be in the model of the Connection specified by <connectionName>. Because of this, you can reach each Aspect of each model available in the Connector with this method. View systems will usually respond by setting the input focus on the corresponding Control.

MicFwDomainProcess>>
startInteractionForAspect:anAspect

This method should be called if the process is ready to activate model-view interaction under Aspect <anAspect> . View systems will usually respond by setting the input focus on the corresponding Control.

MicFwDomainProcess>>
startTransaction

Domain processes which won't open a ModelViewConnector and a view respectively are able to start transaction handling by sending this message.

MicFwDomainProcess>>
stayOpenWithParent

This method has the effect that when the receiver is requested to be closed by a parent business which is going to be closed, and the receiver can be closed but not the parent, then the receiver also stays open if this method returns true, but it is closed if this method returns false.

MicFwViewPort>>
stopInteraction: aConnection

Obsolete. Use leaveInteractionForAspect:goingToAspect: of: instead.

MicFwPersistenceManager>>
storeGlobally

The receiver so that it can be retreived with the instance name later.

MicFwMetaPart>>
tabType: aSymbol

Set the tab type of the page. Valid tab types are: MAJOR, MINOR, NONE.

MicFwMetaAspect>>
title: aValue

Set the initial header title of the column.

MicFwPersistentObject>>
to1Relationship: instVarSymbol references: aPersistentObject

Determine whether the relationship defined on the instance variable <instVarSymbol> holds a reference to the object <aPersistentObject>. This should be done in a way that avoids persistence actions (DB access) as much as possible.

MicFwRdbSQLClause>>
toBeCompiled

Answer whether the receiver contains named parameter markers (?instVar) and thus needs to be compiled before an SQL statement is generated.

MicFwDomainProcess>>
topLevelProcessClosed

Called after the top level process has disconnected its view. Can be overridden to perform cleanups or exit the system.

MicFwNRelationship>>
transact

Answer true if the relationship access handler handles transacted accesses, false otherwise.

MicFwTr1Relationship>>
transact

Answer true if the relationship access handler handles transacted accesses, false otherwise.

MicFwNRelationship>>
transact: aBoolean

Set the transaction property for the receiver. If @aBoolean = true -> handle accesses transacted.

MicFwTr1Relationship>>
transact: aBoolean

Set the transaction property for the receiver. If @aBoolean = true -> handle accesses transacted.

MicFwDomainProcess>>
transactionAborted

Called when a user wants to close the corresponding view after the transaction of the receiver process is aborted.

MicFwDomainProcess>>
transactionClosed: aTransactionCloseMode

Called when a user wants to close the corresponding view. <aTransactionCloseMode> is a boolean which is meant to indicate if an open transaction is to be committed or aborted.

MicFwDomainProcess>>
transactionCommitted

Called when a user wants to close the corresponding view after the transaction of the receiver process is committed.

MicFwTransactionContext>>
transactionLevel

Return the active (highest) transaction level of receiver (as Integer).

MicFwAbstractDomainProcessContext>>
transactionPolicy

Get the transaction policy.

MicFwDomainProcess>>
transactionPolicy

Return the transaction policy object.

MicFwDomainProcess>>
transactionPolicy

Return the transaction policy object.

MicFwAbstractDomainProcessContext>>
transactionPolicy: aTransactionPolicy

Set the transaction policy.

MicFwDomainProcess>>
transactionPolicyClass

See <defaultTransactionPolicyClass>.

MicFwDomainProcess>>
transactionStarted

Hook for initializations immediately after starting a transaction.

MicFwPomRdbCoreComplete>>
typeConverter

Answer the type converter for the receiver. The object answered is used to convert types in places where no specialized converter is available.

MicFwPersistentObject>>
undelete

Deregister the receiver for delete from the underlying persistence medium. CAUTION: Cascaded deletion will not be deregistered for the objects deleted through the cascade-foreign keys. This method is OBSO-LETE and should not be used anymore. Abort the transaction instead to undo the delete.

MicFwPsExtentIdentifier>>
union: anExtentIdentifier

Combine the queryAnalyzer of this descriptor with the queryAnalyzer of @anExtentIdentifier with the #union - operation.

MicFwPsProjectionDescriptor>>
union: aProjectionDescriptor

Combine the queryAnalyzer of this descriptor with the queryAnalyzer of @aProjectionDescriptor with the #union - operation.

OdbcObject class>>
useOdbcDLL

Before executing this method make sure that all your conections are closed.

MicFwDomainProcess class>>
validatePerform: anAspect using: aRule

Use <aRule> (Block or Message) to validate each perform access to <anAspect>. <aRule> must return a boolean.

MicFwDomainProcess>>
validatesPerformAccessTo: anAspect

Answer the validation performing <anAspect>.

MicFwDomainObject>>
validateWrite: #var using: aBlock

Causes any attempted assignment of an object to the variable var to be allowed only if the object to be assigned meets the criteria specified in the block aBlock.

MicFwModelObject class>>
validateWrite: aspect using: rule

Use <rule> (Block or Message) to validate each write access to <aspect>. <rule> must return a boolean.

MicFwPsObjectIdentifier>>
valueFor: keySymbol

Answer the value for the specified key.

MicFwAbstractPersistenceManagerRdb>>
valueIteratorClass

Answer the class to be used for persistence iterations on (undefined) values, i.e. no assumptions are made about the type of objects iterated upon.

MicFwAbstractWindowingPolicy>>
viewClass

Get the view class.

MicFwAbstractWindowingPolicy>>
viewClass: aClassHint

Resolve and set the view class from <aClassHint>.

MicFwDomainProcess>>
viewClosed

Called after the process has closed its view. Can be overridden to perform cleanups or exit the system.

MicFwViewPort>>
wantsCachedReadOf: aDispatcherAspect

This method controls the cache mechanism.Its default behavior is to cache all Aspects.

MicFwViewPort>>
wantsImmediateViewUpdateOf: anAspect

Return true for Viewport Aspects which require view update as soon as they change (e.g. Aspects controlling progress bars).

MicFwRdbIntegrityViolation>>
wasNMDelete

Answer true if the operation that caused the error was a DELETE operation on a NM relationship.

MicFwRdbIntegrityViolation>>
wasNMInsert

Answer true if the operation that caused the error was an INSERT operation on a NM relationship.

MicFwRdbIntegrityViolation>>
wasObjectDelete

Answer true if the operation that caused the error was a DELETE operation on a persistent object.

MicFwRdbIntegrityViolation>>
wasObjectInsert

Answer true if the operation that caused the error was an INSERT operation on a persistent object.

MicFwRdbIntegrityViolation>>
wasObjectUpdate

Answer true if the operation that caused the error was an UPDATE operation on a persistent object.

MicFwRdbSQLClause class>>
where: anSQLWhere

Answer an instance of the receiver that represents @anSQLWhere.

MicFwPersistentObject class>>
where: aQueryArray

Answer a collection of instances of the receiver that conform to the query condition in @aQueryArray. Retrieve instances from persistence medium.

MicFwPersistentObject class>>
where: aQueryArray orderBy: anOrderArray

Answer a collection of instances of the receiver that conform to the query condition in @aQueryArray. Retrieve instances from persistence medium. Order by @anOrderArray.

MicFwPsExtentIdentifierSQL>>
whereSQL

Answer the sql clause.

MicFwPsExtentIdentifierSQL>>
whereSQL: sqlString

Compatibility (from:where) - set the sql clause for the receiver.

MicFwAbstractDomainProcessContext>>
windowingPolicy

Get the windowing policy.

MicFwDomainProcess>>
windowingPolicy

Return the windowing policy object.

MicFwAbstractDomainProcessContext>>
windowingPolicy: aWindowingPolicy

Set the windowing policy.

MicFwDomainProcess>>
windowingPolicyClass

See <defaultWindowingPolicyClass>.

MicFwRdbForeignKeyDescription>>
with: aSelector
Answer an instance of the receiver that will query the SQLClausefor relationship resolution by sending @aSelector to the source object of the relationship at runtime.

MicFwRdbDynamicRelSpec class>>
with: aSelector
Answer an instance of the receiver that will query the SQLClause for relationship resolution by sending @aSelector to the source object of the relationship at runtime.

MicFwViewPort>>
writeAccessRejectedForAspect: anAspect vetoValue: aProhibitedModel
These method is called when read / write access to a model has been rejected. <aProhibitedModel> is afterwards passed to the Abstract Control as result of the model access. Certain properties can be set in <aProhibitedModel> to control the behavior of the Control. Actually setting a property WINDOWACTION to a one argument block will cause the block to be evaluated with the Abstract Control as argument. CAUTION: This feature is still under development, so API may change.

MicFwModelObject class>>
writeDuringReadIgnored
Answer if changes should be ignored during a traced read.

MicFwDomainProcess>>
writeModelAtConnection: aSymbol value: aModel
Write <aModel> into the Connection named <aSymbol>.

MicFwPersistenceContext>>
writePersistentChanges
Attempt to commit the current transaction level. Let all interested parties prepare for the commit if necessary (#aboutToCommit). If the method answers true, proceed. Go to the next lower transaction level. Answer the new transaction level.

Appendix B

Trouble Shooting Guide

This section recommends trouble-shooting approaches to solving common problems when using the Frameworks.

TS 1. Application

TS 1.1. Prerequisite that I want to assign to an application is not available from the list of prerequisites

The required configuration map must be loaded. For example, in order to assign MicFwPersistenceOdbc as a prerequisite for an application, the configuration map micPersistenceOdbc Runtime and micPersistenceOdbc Development must be loaded.

TS 2. DB2

TS 2.1. DB2 hangs up when trying to display the Sample Contents

The POM may still be connected to the database. Open the STOPF for the POM (from the **System Transcript** select **micFrameworks / Browse Persistence Manager**, select the POM) and disconnect the POM from the database (**Manager / Disconnect from database**).

TS 3. Domain Object (DO)

TS 3.1. DO doesnt appear in the Tear-off pop-up window for the DM

The DP must first be assigned to the DM by double-clicking on the DM and entering the DP class in the dialog. AND the DO must be specified in a base connection for the DP (using the DPB).

TS 4. Domain Process

TS 4.1. DP doesnt appear in the Tear-off pop-up window for the DM

The DP must first be assigned to the DM by double-clicking on the DM and entering the DP class in the dialog.

TS 5. Domain Processes Browser (DPB)

TS 5.1. A newly created DP, DO, Viewport, View, etc. does not appear in a drop-down list in the DPB

Try closing and then reopening the DPB.

TS 5.2. Can’t save a change that I just made in the DPB... The Save option is greyed out

Click anywhere else in the DPB (so that the focus changes). The Save option should appear.

TS 6. Envy debugger messages

TS 6.1. "Invalid view. A default view is not set for a process"

A default view was not set for the process. This can sometimes happen because the a change in the DPB was not saved. To solve this problem:

- Close the DPB.
- Open the DPB.
- Reassign the view to the process.
- Save the changes.

TS 6.2. "MicFwTransactionContext is not running"

If a transaction context is aborted within the TB, then the dialog that the context was assigned to cannot be closed. Normally contexts should be aborted or committed by clicking a button in the dialog.

TS 7. Object Net Browser (ONB)

TS 7.1. The changes I made in the ONB did not take effect

Always select **Class / Save** after making any changes in the ONB. NO changes are automatically saved.

TS 8. Transaction Browser (TB)

TS 8.1. MicFwTechnicalTransactionContext’s are displayed in the TB.

MicFwTechnicalTransactionContext is a context that is created when you are using certain Frameworks tools (such as the Domain Processes Browser). These contexts are for internal use only. Do not alter these contexts (ie, commit, abort, etc.) in any way.

TS 8.2. When entering data in the view no changes can be observed in TransactionsBrowser's data even though the Domain Process has started a transaction context.

View / Display in the Connectors Browser: Displayed in the column of the Base Connection in question is

The Connection is empty. The Domain Process is responsible for attaching a valid Domain Object to the Connection (via the model write accessor).

TS 8.3. When entering data in the view, no instance variable can be seen in the TransactionsBrowser even though its contents should have changed and the Domain Process has started a Transaction Context.

View / Display in the Connectors Browser: In the column in the List below no entry corresponding to the name of the instance variable can be seen for the Connection in question.

The name of the widget was not entered in the GUI-Builder (e.g. the of VisualAge™) according to the conventions with underlining following and was therefore not registered when starting the Domain Process.

TS 8.4. When leaving the entry field the Debugger appears with the message <*Class*> *does not understand:* <*Accessorname:*>. In the Transactions Browser the instance variable cannot be seen even though its contents should have changed and the Domain Process has started a Transaction Context.

View / Display in the Connectors Browser: For the Connection in question, a Domain Process is assigned and all accessors are registered.

The name of the widget does not correspond to a method or instance variable of the Domain Object. Either an incorrect Domain Object was placed in the Connection or the name of the widget was incorrectly specified.

TS 9. Transactions

TS 9.1. Changes to variable are not transacted

If a variable has not been specified as a transacted variable (by checking the checkbox "Transacted" in the ONB) then changes to the varible will not be transacted.

TS 10. ViewPort

TS 10.1. No objects are set in the List.

The widget name does not comply with the list convention. Either the name of the Connection for the list entry is not correct or the name of the accessor method is incorrect.

TS 10.2. You confirm a command control (for example: a button), with the result that a Debugger with the message *'a method named <method> was missed in class <aProcessObjectClass>'* appears, even though you have implemented the method in a Viewport.

When running the Domain Process the Viewport for the Domain Object / Domain Process was not found. The class method , which contains the class name of the corresponding Domain Object / Domain Process as a String,

- was not implemented,
- was implemented as an instance method or
- the name of the Domain Object / Domain Process was accidentally misspelled.
- the name of the Domain Object / Domain Process was not enclosed by apostrophes

A second possibility to connect a Viewport to a model is to declare it in the DomainProcessesBrowser at the Connection of the Domain Object / Domain Process to its parent Domain Process. You have to check this as well.

You confirm a command control (for example: a button), with the result that a Debugger with the message appears, even though you have implemented the method in a Viewport.

TS 10.3. You have implemented a method in a Viewport class and you wonder why it is not called even though you have named a widget accordingly. Worse scenario: A Debugger is also activated with the message '<...> *does not understand* <...>'.

When running the Domain Process the Viewport for the Domain Object / Domain Process was not found. The class method , which contains the class name of the corresponding Domain Object / Domain Process as a String,

- was not implemented,
- was implemented as an instance method or
- the name of the Domain Object / Domain Process was accidentally misspelled.
- the name of the Domain Object / Domain Process was not enclosed by apostrophes

A second possibility to connect a Viewport to a model is to declare it in the DomainProcessesBrowser at the Connection of the Domain Object / Domain Process to its parent Domain Process. You have to check this as well.

You confirm a command control (for example: a button), with the result that a Debugger with the message appears, even though you have implemented the method in a Viewport.

TS 11. Visual Age Organizer

TS 11.1. A menu item is not shown

VA Organizer has an option to display Full or Basic menus. To display Full Menus: From the VA Organizer: Select **Options / Full Menus**.

Appendix C

Frequently-asked questions

This section provides the answers to the most frequently-asked questions about Frameworks..

FAQ 1. Connections

FAQ 1.1. How do I connect a dynamic container widget (ie, a Form or a Notebook Page) to the Framework?

The accessor name of container widgets has 4 parts. The first two parts are used to get the content of the container or set Subaspects like "<accessorName> visible" or "<accessorName> enabled". So, if you want to set the visible state of a Form, you have to use the second part of the forms accessor name.

If you write a method to set the content of such widget, this method must return a
, which contains (see Metapart Collections).

Now for the last two parts of the accessor name: They are used to connect the widgets inside the container to a special default Base or default Process Connection.

It's important to understand, that all the Connection Names must be seen relativ to the surrounding container. If you connect a Form A to a child process A and connect a Form B inside Form A to a child process B, process B must be a child process of process A.

A Form should be reusable and because of this, widgets inside the Form shouldn't access Connections directly. You should always try to use "defaultBase", "defaultProcess" or a capitalized dummy name like "VOID". As mentioned above, you can use the third and fourth part of the accessor name of the surrounding Form to exchange the "VOID" names with the Connection Names you want to..

FAQ 2. Domain Processes

FAQ 2.1. How can a child process can prevent the closing of the parent process ?

A child process can prohit the closing of the parent process.

The main process attempts to close the view.

If #commitAndCloseProcess fails, ask the user if to abort or cancel and perform the appropriate action:

```
defaultClose
   self commitAndCloseProcess ifFalse:[self askUserIfAbortOrCancel ifTrue:[self
abortAndCloseProcess]]
```

The child process can block the attempt to close the window with:

```
canClose: aCloseInfo
   ^ self checkForCloseInfo: aCloseInfo
```

checkForCloseInfo: returns <false>.

FAQ 2.2. How can I implement additional custom behavior before actually closing a process ?

The method (for example userAbortAndCloseProcess) that is executed when a process is to be closed (committed, aborted, etc) can send addtional messages before actually sending the message that closes the process.

Assume in the following example code that a view has an abort and a commit button. Clicking a button sends the userAbortAndCloseProcess or userCommitAndCloseProcess message to the process respectively. Both of the methods send the required message (userCloseCode) before actually sending the message to itself to abort or commit the process:

```
#UserProcess >> userAbortAndCloseProcess
   self userCloseCode.
   self abortAndCloseView.
#UserProcess >> userCommitAndCloseProcess
   self userCloseCode.
   self commitAndCloseView.
```

FAQ 2.3. How can the user be allowed to choose whether or not the default close of a main process causes the entire application to commit or abort?

Normally when a main process closes, a CloseInfo object with CloseMode of ##DEFAULT is sent to the child processes. This informs the child processes to close with the default commit/abort setting returned by sending the message commmitTransactionByDefault to self.

This method is implemented by the MicFwDomainProcess class to return true. The child processes can override this method, in which case the child process commits or aborts according to the internal settings in the child process.

You have a problem if one childs performs a commit by default while the other one performs an abort, but the user wants to enforce an abort for the entire application.

The solution to the upper problem is to change the close mode of close info, which is handed down to all child processes to ##COMMIT or ##ABORT according to the user input.

When this is done, the child is supposed to abort or commit accordingly ignoring its default behaviour.

Example

Create a method in the main process similar to the following (valid values are ##DEFAULT, ##COMMIT)

```
closeRequested: aCloseInfo
  self userWantsToAbort ifTrue: [aCloseInfo closeMode: ##ABORT]
```

<userWantsToAbort> is a method that verifies the status of the process and whose return value determines with which mode (abort or commit) the process should be closed.

This method could pop up a message box and ask the user if he/she wants to abort or commit.

By changing the close mode to ##ABORT the process and all its childs try to perform an abort and close process.

FAQ 2.4. How do I override the default close behavior for a process ?

Normally, when the ”X” button in window is clicked (close window) or Alt-F4 is pressed, the process closes in the default manner.

However, the process can override the default behavior by overriding the commitTransactionByDefault method.

```
#MyDomainProcess >> commitTransactionByDefault
  "API - Override optional - When the view is closed, is an open transaction
   to be committed by default (^true) or aborted (^false)?"
^true
```

FAQ 3. Errors

FAQ 3.1. How can i register an InputError or WidgetError in the FrameWork

False input in an entry field (false date types) will not be registered by the Framework.

The converterManager (if defined in the Settings) verifies the input of the View.

For example, if a Date converterManager was defined for an EntryField (SettingsPane -> converter -> Data Type = Date). The converterManager will check the validity of the input to this field. For example, if a String is entered, the convertere Manager will verify that the input represents a valid Date; otherwise, the input will be rejected. If the input is rejected, an AbtConverterError (userInputError) is generated and the ErrorString (*** error ***) is displayed in the entry field.

Normally the userInputConvertError is not passed on to the Framework. If userInputConvertError should be passed on to the Framework, then the utilized widget class must be subclassed and the Event must be registered. See ‘View Adapter’ (page 226).

For example, the following widget is subclassed:

```
AbtTextView subclass: #UserAbtTextView
```

And the following view implementation is created:

```
MicFwAbtTextViewImplementation subclass: #MicFwUserAbtTextViewImplementation
MicFwUserAbtTextViewImplementationclass >> implementedEvents
  "Public - Answer a dictionary with supported real events as keys and
    abstract events as values. "
    ^IdentityDictionary new
      at: #losingFocus            put: MicFwObjectUpdateEvent;
      at: #stringChanged          put: MicFwObjectChangeEvent;
      at: #helpRequested          put: MicFwHelpRequestEvent;
      at: #userInputConvertError  put: UserFwObjectInputErrorEvent;
      yourself
```

The Event class must be created. The input can be rejected in the method <triggerWithCallData: data> (as shown in the following example) or a method in the Viewport of the Domain Model that receives the input can be called.

Implementation of a custom error event:

```
MicFwObjectAspectEvent subclass: #UserFwObjectInputErrorEvent
UserFwObjectInputErrorEvent >> triggerWithCallData: data
  MicSystem messageBox message: 'The Input: ' , data offendingInput , ' is
incorrect '.
  self abstractWindow windowImplementation window converterManager userInput-
```

```
ErrorObject: nil.
  self abstractWindow windowImplementation window converterManager converter-
AttributesChanged.
  self abstractWindow realValue: self abstractWindow abstractValue.
    self abstractWindow windowImplementation window widget setInputFocus.
```

FAQ 4. Forms

FAQ 4.1. Why should I avoid adding MicFwMetaPartCollection when working with dynamically exchanged forms?

You should work with the visible Subaspect and have only one visible at a time! This approach is supported by the new Framework behavior, which is to have model access only for visible widgets.

FAQ 5. Lists

FAQ 5.1. How do I delete elements from a list with the Del or other key

Deleting elements with the Del key requires MicVaMnemonics R3.4 V1.1 or higher and MicVaMnemonicsD R3.4 V1.1 or higher.

In the Composition Editor (in which the view is displayed): Create a new free popup menu and add a menu item to it.

On the settings page of the menu item: Select the Del key as the accelerator. Open the context menu on the popup menu (not on the item) and connect micKeyboardConnection with the micKeyboardConnection of the required List. Set the fw-accessor name of the menu item as for a button. This button will be clicked and the method will be performed when the Del key is clicked, assuming that the List has the focus.

FAQ 6. Notebooks

FAQ 6.1. How Do I Get Informed When a Notebook Page is About to be Left or About to be Entered?

You should use

`enterInteractionForAspect:comingFromAspect:of:`

and / or

`leaveInteractionForAspect:goingToAspect:of:`

These methods can also be used to prevent the change. (See Dynamic Notebooks).

FAQ 7. Transactions

FAQ 7.1. How do I commit an application with commitAndBegin and open a new transaction level for all processes that had an open transaction level at the time of the commitAndBegin?

The main process executes #commitAndBeginTouched. For each child process the following method is implemented:

```
processClosed
  ... "required close code" ...
  ... "other required code" ...
  self setProcessFlaggedUntouched "this is internal"
```

If the flag is not set to untouched when a process closes, then a new transaction will be opened with <commitAndBeginTouched> for all processes that had an open transaction (even if the process is currently closed).

FAQ 8. ViewPorts

FAQ 8.1. How can I use an incompatible ViewPortRequestBroker?

Tools such as the Domain Processes Browser are written using the Application Framework. Therefore, a compatible VPBroker is must be available.

The tools require #MicFwCachingViewPortRequestBroker in the development image. If an incompatible VPBroker was created, then during development an additional mapping must be implemented and a different dispatcher used.

The Domain Processes of the tools are marked as <isTechnicalDomainProcess>. All requests to the

VPBroker are then forwarded via a Dispatcher.

If a custom ViewPortBroker that is not compatible with MicFwViewPortRequestBroker is to be used, then the #MicFwViewPortRequestBrokerDispatcher must be used.

All requests will then be forwarded via the #MicFwViewPortRequestBrokerDispatcher to the #customerVP-Broker or the #technicalVP-Broker. The #technicalVP-Broker is normally #MicFwCachingViewPortRequestBroker.

The following code shows how the custom VPBroker should be registered in the Broker mapping:

```
registerForDevelopment
   self broker use: ACustomVPBroker for: #customViewPort;
   self broker use: MicFwCachingViewPortRequestBroker for: #technicalViewPort;
   self broker use: MicFwViewPortRequestBrokerDispatcher for: #viewPort;
```

During runtime (with device components), a ViewPortRequestBroker must be available via the mapping #viewPort. The ViewPortBroker can also be a custom broker.

```
registerForRuntime
   self broker use: ACustomVPBroker for: #viewPort;
```

Overwriting the Dispatcher or the customer viewport when loading the application #MicFwViewPorts can cause problems. The Dispatcher is therefore only available in development; this is not critical for performance.

For more information about the VPBroker, see **‘ViewPort-RequestBroker’ (page 175)**.

Appendix D

Glossary

This glossary defines terms that are presented throughout this manual.

->1 relationship: In ->1 relationships, a source instance variable references 0..1 target objects (nil or 1 object). No target instance variable refers to the source object.

An example of a ->1 relationship is the relationship between Person (source) and PersonName (target). Person instance variable name references 1 PersonName object (a person has only 1 name). No PersonName instance variable references the Person object (a PersonName object never needs to know which Person object is referencing it).

->N relationship: In ->N relationships, a source instance variable references min...max target objects (where min...max is specified by the cardinality).

An example of a ->N relationship would be the relationship between Person (source) and PersonName (target), with the assumption that a person can have more than 1 name. Person instance variable name references min...max PersonName objects, where min = 1 and max is unspecified. No PersonName instance variable references the Person object.

1<->1 relationship: In 1<->1 relationships, a source instance variable references 0..1 (signified by the "1" on the RIGHT of "1<->1") target object. The target instance variable references the same 1 (signified by the "1" on the LEFT of "1<->1") source object or nil. The relationship is not primitive, because the target object should have a reference to the source object.

An example of a 1<->1 relationship would be the relationship between Customer (source) and Portfolio (target). Customer instance variable portfolio references 1 Portfolio object. Portfolio instance variable customer references the 1 Customer object. The relationship is not primitive, because a Portfolio object should know which object is its Customer.

1<->N relationship: In 1<->N relationships, a source instance variable references N (where N is any number with the range max...min specified by the cardinality of the relationship) target objects. There is an instance variable in all of the referenced target objects that reference the 1 source object.

An example of a 1<->N relationship would be the relationship between Employee (source) and Customer (target). Employee instance variable ownedCustomers would reference N (cardinality min <= N <= cardinality max) Customer objects. The instance variable ownerEmployee in each referenced Customer object would reference the 1 Employee object.

Abort a context: A context is aborted when all version objects within the context are dereferenced (ie, none of the changes that were transacted while the context was active will actually be implemented) and the context ceases to exist. See: abortcontext.

Abstract Control:: An Abstract Control simplifies external access to the Framework, as such access can only take place via real Control. There are only a small number of genuinely different Abstract Controls (see command example in the MVC chapter). The actual access attempts are handled with real Control via an appropriate Adapter.

Focus change, issuing commands and transfer of data - including a range of state information like validation or authorization - will be done in this abstract layer. The real Controls are completely decoupled from all Domain Process actions and Domain Model data; all technical details of external interfaces (GUI, DDE, ...) are completely hidden in the Abstract Control implementation.

Abstract Event: Abstract Window Events are the objects that really perform the communication between the view system and the model world, whereas Abstract Windows are merely containers for Abstract Events which additionally may provide some services for them. Abstract events can be divided into two functional types: Abstract events which propagate changes in the model world to the view and Abstract Events which propagate changes or requests from the user interface to the model world.

Abstract Value: An Abstract Value is a container object that is used by Viewports to keep and propagate information about a Viewport Aspect to and from the view. It does not only contain a value for the content of a control, but also different kinds of state information like whether or not the control should be enabled, what the (background) color is, which context help text is displayed, and so on.

Abstract View: When a real View is created, an Abstract View begins to exist in its shadow. The lifetime of the Abstract View is exactly determined by the lifetime of the real View. The Abstract View's purpose is to manage the Abstract Controls corresponding one-to-one to the real Controls on the view. Moreover, the Abstract View performs coordination between abstract and platform layer when opening, activating and closing the view. It communicates with the real Platform View using a Platform Adapter.

Accessor Generator: The accessor generator is the object that creates the OBF accessors for a variable.

Active Context: Only 1 context can be active at anytime. While a context is active, any changes to any transacted variables will be recorded in object versions. This variable will be locked by the context, which means that no changes may be made to the variable while a sibling context or parent context is active as long as this context exists.

Adapter: A Platform Adapter system is introduced which does the translation of protocols and overcomes the architectural differences between the host Smalltalk system architecture and the Application Framework architecture. This makes the core part of the framework itself portable.

Archiver: See: Code Archiver.

Authorization: The access to individual objects controlled on the model level. Authorization works both for accessing attributes of Domain Objects and for executing Aspects of business processes.

Broker: To allow subsystems or specific requests to them to be exchanged with an own implementation, Application Framework uses Broker classes that offer a thin public interface with the internal knowledge how to delegate the call to the subsystem. A common Broker concept enables the developers to modify request algorithms or subsystem behavior quite easy.

Cardinality: The cardinality of a relationship determines the required min and max number of target objects referenced by the source variable. In 2-way relationships, there are 2 cardinalities. The second cardinality determines the requried min and max number of source objects referenced by the target variable.

CB: Connections Browser.

Child context: A child context can change any variable locked by its parent. A variable, having been changed by the child context, is now locked by the child context. The parent context cannot change a variable locked by the child. See: parent Context.

Code Generator: See Accessor Generator.

Commit a context: Committing a context has the same effect as committing all transaction levels (TrLevels) in the context.

Committed target: In a VersionObject: A getter message to a source object will return the committed target object referenced by the variable if no context is active OR if [the active context read mode is isolate AND the source object’s variable is not locked by the active context AND the active context is not a child context of the context with the variable lock]. See: transacted Target.

Concurrent contexts: 2 contexts are concurrent if there is no parent-child relationship between them. Such contexts can also be referred to as "sibling" contexts. A sibling context may not change a variable locked by another sibling context.

Concurrent transactions: See Concurrent Contexts.

Connector: A Connector is used to connect objects (Domain Objects and Domain Processes) to an external view. The Connector decouples view and model objects and provides access to connected Domain Objects and Domain Processes on a low level. In this respect, it forms a bridge between the objects, decoupling also Domain Objects from Domain Processes and thus making them more combinable and exchangeable.

Context: See Transaction Context.

DDL: A language enabling the structure and instances of a database to be defined in a human- and machine-readable form.

Default Base Connection: The Default Base Connection will be used to hold the Domain Object if no explicit Base Connection is defined for this process.

Delegation Model: An concept used to decouple subsystems from each other. Application programmers implement subclasses of Viewport to isolate the model from the interaction subsystem.

Domain Model: A Domain Model is a design phrase for real world concepts like Customer, Policy or Address. Its instances are called Domain Objects.

Domain Object: Domain objects describe that part of the MVC architecture which corresponds to business terms. The behavior and structure are primarily defined in this respect. The appropriate tool (object net browser) from the Object Behavior Framework is used for this purpose, and mapping to the database is described with STOPF from Persistence Framework . The Domain Object also has open interfaces for connecting authorization and validation.

Domain Process: A Domain Process is a design phrase for processing and workflow-oriented tasks and control flow. Its instances are also called Domain Processes. Due to their controlling-oriented nature, they take also responsibility for managing a Transaction Context if appropriate.

Domain Processes Browser: A tool, supplied with the Application Framework to browse the exiting Domain Processes

DPB: Domain Processes Browser.

Extended Description (MicFwExtendedDescription): The object that completely describes the typing of all variables in a class. The class method createExtendedDescription creates the MicFwExtendedDescription object.

Framework Logger: The tracing facility of the Framework's. Events and messages will be routed through this logger.

Framework: A Framework is a software architecture for certain tasks, which components can be easily reused by application developers. It provides the system with a basic structure in being a collection of cooperating and conceptual concise classes and methods, which are designed to support a task-oriented work progress in application development.

Inactive Context: A non-active context. See: Active Context.

InstanceVariableDescription (MicFwInstanceVariableDescription): The object that completely describes the typing of a single instance variable in a class. The class method createExtendedDescription creates the MicF-wInstanceVariableDescription object (which is referenced within the MicFwExtendedDescription object).

Isolated Context: A variable getter sent to a source object that is locked by another context while an isolated context is active will return not the committed target of the transaction object in the other context, but rather the uncommitted target. See: uncommitedread context.

M<->N relationship: In M<->N relationships, a source instance variable references M (where M is any number with the range max...min specified by the cardinality LEFT of the relationship) target objects. There is an instance variable in all of the referenced target objects that reference the N (where N is any number with the range max...min specified by the cardinality RIGHT of the relationship) objects of the source object class, including the current source object.

An example of an M<->N relationship would be the relationship between Person (source) and Address (target). Person instance variable addresses would reference M (cardinality min <= M <= cardinality max) Address objects. Address instance variable persons would reference N (cardinality min <= N <= cardinality max) Person objects, including the current Person object.

Mapper: To maintain registration of loosely coupled elements within the Framework, Mappers are used to set, get and remove associations between unique, constant names and their corresponding classes, which themselves may change or may be reimplemented in your project. Mappers are implemented as singletons and can be reached through an easy to use interface as they extend Object with one method per Mapper.

MicFwTransactionContext:

Model View Connector: Is the object that holds the child-process- and Base-Connections for one process.

Model View Controller: The MVC (concept of a Model View Controller) defines an architecture intended to yield a strict decoupling of Domain Model Aspects, flow control and external views - mostly displayed graphically for user interaction.

Nested transaction levels: The transaction levels in a context are nested if more than 1 level exists.

OBF (Object Behavior Framework): OBF is a library of classes that when added to your Smalltalk development environment provides a Framework for defining the requirements and restrictions for the behavior of objects.

Object Net Browser: A visual OBF tool that dispayes both the object net of a class and information about the instance variables of the class (key, validated, transacted, persistent, type, relationship). The visual OBF tools Type Editor and Relationship Editor are opened from the Object Net Browser.

Object Net: An Object Net consists of objects and its relationships between them.

ObjectVersion (version object): An ObjectVersion is created if a transaction context is active and a target object is assigned to a transacted variable of a source object. The ObjectVersion is assigned to the highest TrLevel in the active context. If an ObjectVersion already exists in this TrLevel, it is replaced. The ObjectVersion references the committed target object (committed) and the uncommitted target object (uncommitted) referenced by the variable.

ONB: Object Net Browser.

Packaging: The process of creating a runtime executable.

Parent context: A parent context is a context object that created its child context when it responded to the newTransactionContext message. The parent context can have any number of child contexts. A parent context can also be a child context. If parent context B has a parent context A and B also has a child context C, then A and C also have a parent child relationship. See: Child Context.

POM: Persistent Object Manager.

Primitive relationship: A 1-way relationship. See: 1-way relationship.

Relationship Editor: A visual OBF tool for establishing relationships between 2 classes. See: object Net Browser.

Relationship: In Smalltalk, relationships between objects are simply represented as object pointers. No distinction is made between complex and simple (scalar) data types since all are full-scale objects and there is conceptually no difference between a relationship and an "embedded" value.

Beyond this, the PMS MICADO Object Behavior Framework provides an elaborated relationship concept maintaining referential integrity between objects. Those relationships may even be mapped to (relational) databases with the help of the PMS MICADO Persistence Framework

RelationshipDescription (MicFw~): Object which contains complete information about a relationship.

Running context: A context that has a TrLevel1 (it may have higher levels also).

Sibling contexts: See Concurrent Contexts.

STOPF / ODBC: Smalltalk Object Persistence Framework

Transacted variable: A variable that has been marked as transacted in the Object Net Browser. NOTE:Changes to a variable will not be transacted even if a transaction context is active if the variable is not marked as transacted.

Transaction Browser: A Visual OBF tool for displaying and manipulating (aborting / committing) transactions.

Transaction Context: A logical unit that can contain transaction levels and that can be related to other contexts as a parent, sibling, or child.

Transaction Level: A subunit of a transaction context. A transaction level has a single object version for each variable of each object that was assigned a new target object while the the transaction level was the highest level in its context and its context was active.

Transaction Manager: The transaction manager is a single instance (singleton) of MicFwTransactionManager and manages all running contexts.

Transaction: A defined state of an object that runs under transactional control will be stored. If the interaction on this object fails by some reason, this object can be restored to this state, if the interaction succeeds, the persistent state of this object will be modified to this new state and the old state will be dropped.

TrLevel1, 2, ...: Transaction level 1, 2, ...

Type Converter: A type converter is assigned to a typed variable (including variables in a relationship). When an object is assigned to the variable that is not of the type specified for the variable, the type converter will be used to attempt to create the correct type of object containing the information in the original object and assign this correct type of object to the variable.

Type Editor: A visual tool for setting a variable type as a simple type (Integer, Date, etc.). The type converter and size / scale for the variable type can also be selected. The Type Editor is opened from the Object Net Browser.

TypeDescription (MicFw~): Object which contains complete information about the typing of an instance variable.

Typing: The term typing defines the assignment of types (normally basic classes like Integer or String) to instance variables. Since Smalltalk is an untyped language, the PMS MICADO Object Behavior and Persistence Frameworks introduce typing in order to support storing of objects into (typed) relational databases.

UML (Unified Markup Language): Graphical notation for OO analysis and design.

uncommitedRead context: If an ObjectVersion exists for aSourceObject>>aSOVariable (ie, aSOVariable is transacted and anUncommittedTargetObject was assigned to aSOVariable while aContext1 was active): If aContext2 is an active uncommitedRead context and getter message sOVariable is sent to aSourceObject, the object returned will be anUncommittedTargetObject. See: Isolated context.

Note: The message is spelled "uncommitedRead".

Uncommitted target: In a VersionObject: A getter message to a source object will return the uncommitted target object referenced by the variable if [the active context read mode is uncommitedReadisolate AND the active context does not have a lock on the source object variable AND the active context is not a child context of the context that has the lock] OR if the source object's variable is locked by the active context. See: Committed Target.

Validation of ExtendedDescription: Validation of an ExtendedDescription involves checking the Object Net for errors.

Validation: Like the authorization service, validation ties in at the model layer and is used for checking value-based access to and from attributes based on certain rules.

VariableDescription (MicFw~): Object which contains complete information about an instance variable.

Viewport: In order to separate view-related state handling from Domain Objects and Domain Processes, Application Framework implements a Viewport as a Delegation Model concept. Functionality for transportation and the filtering of data and states concerning an object net from a certain object's or view's direction is delegated from the Domain Models to them, leading to a more lightweight and view independent kind of Domain Object.

Viewports "learn" the dependencies between their Aspects and the corresponding Model Aspects while running the application. This read trace frees the programmer from dealing with changed-events and allows Application Framework to update both sides automatically and in a generic way.

A Dispatcher is called Viewport if Domain Objects are concerned.

Bibliography

The following documents were referenced throught this document.

[BOOC] Grady Booch et. al. (1991): Object-Oriented Design With Applications; Benjamin / Cummings.

[ENVY] IBM : Visual Age User's Guide / Reference, Rel. 4.0; IBM.

[LALO] Wilf LaLonde (1994): Discovering Smalltalk; Benjamin / Cummings.

[OBFW] PMS MICADO (1998): Object Behavior Framework R3.3 V1.3, User’s Manual; PMS MICADO.

[PSFW] PMS MICADO (1998): Persistence Framework R3.3 V1.3, User’s Manual; PMS MICADO.

[RUMB] James Rumbaugh et. al. (1993): Object Oriented Modeling and Design; Prentice Hall.

[STVP] Digitalk (1994): Smalltalk V Programming Reference; Digitalk Inc.

[WBPR] Object Share Systems (1995): WindowBuilder Pro/V 3.1, Tutorial and Reference Guide; Object Share Systems Inc.

[BJR] Fowler / Scott (1998): UML Distilled Addison / Wesley

[RATIONAL] www.rational.com/uml/

[OBJECT INT] http://www.oi.com/

List Of Tables

List Of Figures

Index