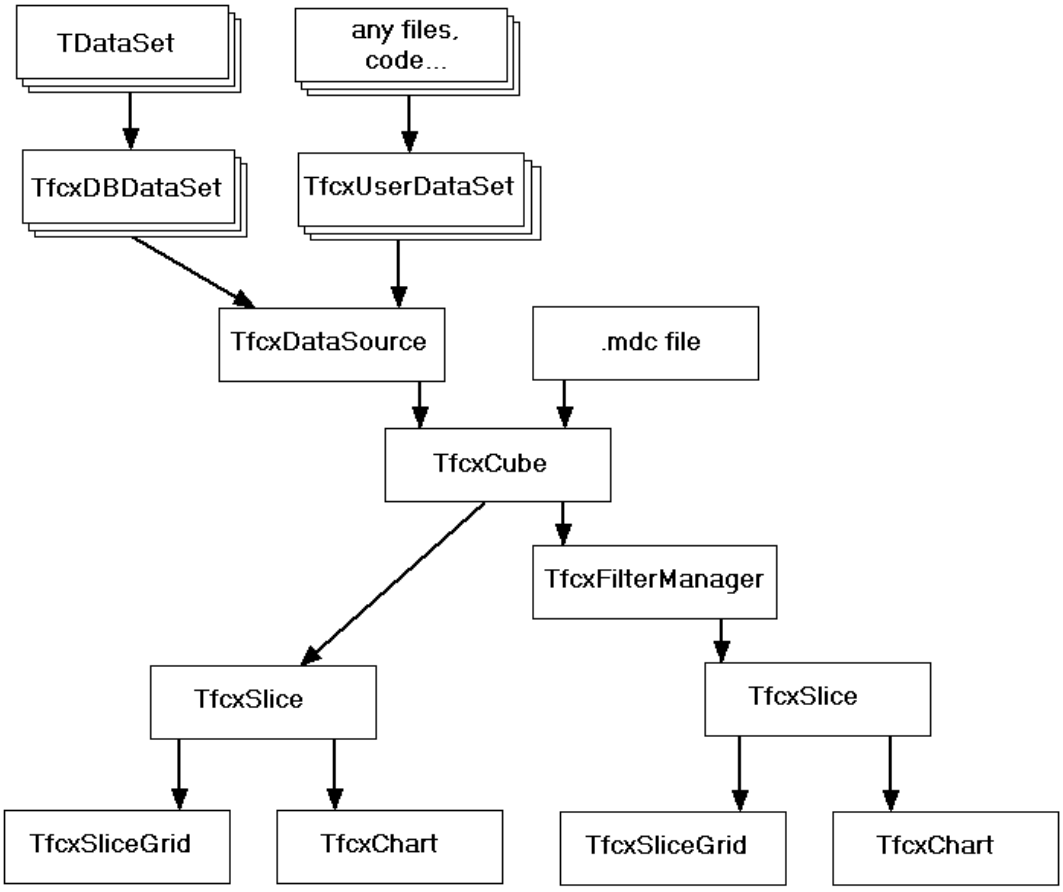
FastCube 2.0 Programmer Manual

Table of contents

Chapter 1

FastCube 2 Architecture

The FastCube 2 component library is a set of non-visual and visual components which store, handle and visualize multi-dimensional data. The FastCube 2 architecture is represented in the following diagram:



FastCube 2 consists of the following components:

Non visual components

- **TfcxDBDataSet** - component for connecting to a database source
- **TfcxUserDataSet** - component for connecting to a user source (based on event handlers)
- **TfcxDataSource** - component which links all the cube data sources and describes the fields and attributes
- **TfcxCube** - (a cube) the main data storage
- **TfcxSlice** - (a slice) structure responsible for data presentation and accordingly preparation of cube data
- **TfcxFilterManager** - manages filtering of the cube data for the slice

Main visual components:

- **TfcxCubeGrid** - (data table) visualizes source data from the cube
- **TfcxSliceGrid** - (cross table) visualizes data based on slice structure and allows users to manipulate the data and structure

- **TfcxCubeGridToolbar** - (data table toolbar) contains a set of buttons allowing actions on the data table
- **TfcxSliceGridToolbar** - (cross table toolbar) contains a set of buttons allowing actions on the cross table, slice and cube

Charting components:

- **TfcxChart** - visualizes source data from the cube as a chart/ diagram
- **TfcxChartToolbar** - (chart toolbar) contains a set of buttons allowing actions on the chart

Chapter II

Working with FastCube components

2.1 Saving and Loading of Cube and Slice

Data loaded into the cube can be saved for the future use. Saved data can then be loaded into the cube without needing to access the source database. As well as the data, the settings for the slice, groups, filters and charts can also be saved. The cube data and/ or settings can be saved in a file, a stream or in a BLOB-field in a database, using methods of the TfcxCube, TfcxSlice, TfcxFilterManager and TfcxChart components.

2.1.1 Cube data

Cube data (TfcxCube):

function LoadFromFile(ACubeFileName: String): Boolean;
Loads cube data from a file. Returns True if the file has been successfully loaded.
The cube is cleared before the data is loaded.

function LoadFromStream(ACubeStream: TStream): Boolean;
Loads cube data from a stream. Returns True if the stream has been successfully loaded.
The cube is cleared before the data is loaded.

function AppendFromFile(ACubeFileName: String): Boolean;
Appends cube data from a file. Returns True if the file has been successfully loaded.
The cube merges the data it already contains with the loaded data.

function AppendFromStream(ACubeStream: TStream): Boolean;
Appends cube data from a stream. Returns True if the stream has been successfully loaded.
The cube merges the data it already contains with the loaded data.

procedure SaveToFile(ACubeFileName: String; AFilter: TObject = nil);
Saves cube data to a file. If the AFilter argument points to a TfcxFilterManager object the cube saves only that data which passes the filter.

procedure SaveToStream(ACubeStream: TStream; ACompressionLevel: TCompressionLevel = clMax; AFilter: TObject = nil);
Saves cube data to a stream. If the AFilter argument points to a TfcxFilterManager object the cube saves only that data which passes the filter. The ACompressionLevel argument sets the compression level.

The cube saves group and slice settings together with the data. The cube does not save the state of filters and any linked chart settings.
The cube file has an mdc extension by default.

Code examples:

```
fcxCube1.LoadFromFile('c:\cube1.mdc');
fcxCube1.AppendFromFile('c:\cube1.mdc');
fcxCube1.SaveToFile('c:\cube2.mdc');
fcxCube1.SaveToFile('c:\cube2Filter.mdc', fcxFilterManager1);
```

2.1.2 Slice settings

Slice settings (TfcxSlice):

function LoadFromFile(AFileName: String): Boolean;

Loads slice settings from a file. Returns True if the file has been successfully loaded. The slice is reset before the load. If the loaded settings contain information about groups then the cube group settings are cleared before the load. If the loaded settings contain information about filters then the filter manager settings are cleared before the load. If the loaded settings contain information about charts then the chart settings are cleared before the load.

function LoadFromStream(ASliceStream: TStream): Boolean;

Loads slice settings from a stream. Returns True if the stream has been successfully loaded.
The slice is reset before the load. If the loaded settings contain information about groups then the cube group settings are cleared before the load. If the loaded settings contain information about filters then the filter manager settings are cleared before the load. If the loaded settings contain information about charts then the chart settings are cleared before the load.

procedure SaveToFile(AFileName: String; AStoreItems: TfcxItemsForStoreWithSlice = []);

Saves slice settings to a file. The AStoreItems argument sets which additional information is also to be saved (filters, groups, charts).

procedure SaveToStream(ASliceStream: TStream; AStoreItems: TfcxItemsForStoreWithSlice = []);

Saves slice settings to a stream. The AStoreItems arguments sets which additional information is also to be saved (filters, groups, charts).

Slice files can also contain group settings, filter settings and chart settings. Slice files are xml files with an mds extension by default.

Code examples:

```
fcxSlice1.LoadFromFile('c:\schema1.mds');
fcxSlice1.SaveToFile('c:\schema2.mds');
fcxSlice1.SaveToFile('c:\schema3.mds', [fcxiss_Filters, fcxiss_Groups,
fcxiss_Charts]);
```

2.1.3 Other settings

Filter settings (TfcxFilterManager):

function LoadFromFile(AFileName: String): Boolean;
Loads filter settings from a file. Returns True if the file has been successfully loaded. The filter settings are cleared before the load.

function LoadFromStream(AStream: TStream): Boolean;
Loads filter settings from a stream. Returns True if the stream has been successfully loaded.
The filter settings are cleared before the load.

procedure SaveToFile(AFileName: String);
Saves filter settings to a file.

procedure SaveToStream(AStream: TStream);
Saves filter settings to a stream.

The filter settings file has an fcf extension by default.

Group settings (TfcxCube):

function LoadGroupsFromFile(AGroupsFileName: String): Boolean;
Loads group settings from a file. Returns True if the file has been successfully loaded.
The group settings are cleared before the load.

function LoadGroupsFromStream(AStream: TStream): Boolean;
Loads group settings from a stream. Returns True if the stream has been successfully loaded.
The group settings are cleared before the load.

procedure SaveGroupsToFile(AGroupsFileName: String);
Saves group settings to a file.

procedure SaveGroupsToStream(AStream: TStream);
Saves group settings to a stream.

The group settings file has an fcg extension by default.

Chart settings (TfcxChart):

function LoadFromFile(AFileName: String): Boolean;
Loads chart settings from a file. Returns True if the file has been successfully loaded.
The chart settings are cleared before the load.

function LoadFromStream(AStream: TStream): Boolean;
Loads chart settings from a stream. Returns True if the stream has been successfully loaded.
The chart settings are cleared before the load.

procedure SaveToFile(AFileName: String);
Saves chart settings to a file.

procedure SaveToStream(AStream: TStream);
Saves chart settings to a stream.

The chart settings file has an mdt extension by default.

Code examples:

```
fcxFilterManager1.LoadFromFile('c:\Filter1.fcf');
fcxFilterManager1.SaveToFile('c:\Filter2.fcf');

fcxCube1.LoadGroupsFromFile('c:\Group1.fcg');
fcxCube1.SaveGroupsToFile('c:\Group2.fcg');

fcxChart1.LoadFromFile('c:\Chart1.mdt');
fcxChart1.SaveToFile('c:\Chart2.mdt');
```

2.2 Data loading

Data loading from database and user sources.

2.2.1 Loading cube from a single database table

The main goal of the FastCube component library is to create a cross summary table from "flat" data.

The simplest cube data source is a database table.

To load data into the cube a connection to the database has to be created through a descendant of the TDataSet component. The exact choice of component depends on the database component used in the application.

A TfcxDBDataSet component is needed to link the TDataSet descendant with a TfcxDataSource component.

The TfcxDataSource component contains the full description of the data structure for the cube. It describes all the data sources, files for those sources, relations between sources, rules for data conversion, etc. One of the sources must be the main source, which is assigned to the TfcxDataSource.DataSet property. When all the data to be loaded is contained in a single database table then only the main source needs to be assigned.

Next the TfcxDataSource, TfcxCube, TfcxSlice and TfcxSliceGrid components must be linked together. TfcxCube and TfcxSlice can be linked through the filter manager, TfcxFilterManager. If a TfcxFilterManager component is not explicitly added to the application then the TfcxSlice will automatically create an internal TfcxFilterManager. An explicit filter manager component is only needed when one filter manager will be used by more than one slice.

TfcxDataSource can contain a list of the source fields. If the field list is not present then all the source fields will be automatically loaded.

TfcxDataSource.Fields contains the list of main source fields. The field list can be deleted by calling TfcxDataSource.DeleteFields. A field list can be loaded from the source by calling TfcxDataSource.AddFields. The field list is loaded only when the source contains field objects (which were defined in the form designer or created automatically when the DataSet was opened). If the field list is not going to be changed then it is not necessary to call AddFields, since it is called automatically when the source is opened.

The cube source can be a TfcxDataSource component, a cube file or a cube stream. The type of source to be used is specified in the TfcxCube.CubeSource : TfcxCubeSource property. The TfcxCubeSource enumeration is:

```
TfcxCubeSource = (
  fccs_None,            // None
  fccs_DataSource,      // load from fcxDataSource
  fccs_CubeFile,        // load from file
  fccs_CubeStream       // load from Stream
  );
```

In our case we need to use the value fccs_DataSource.

The cube loads its data when the TfcxCube.Open method is called. The cube automatically opens the specified source and loads the data from the required fields.

The cross-table is ready for use after the data has been loaded.

Code examples:

```
// Create required components at run time
fcxDBDataSet1 := TfcxDBDataSet.Create(Self);
fcxDataSource1 := TfcxDataSource.Create(Self);
fcxCube1 := TfcxCube.Create(Self);
fcxSlice1 := TfcxSlice.Create(Self);
fcxSliceGrid1 := TfcxSliceGrid.Create(Self);
fcxSliceGrid1.Parent := Self;
fcxSliceGrid1.Align := alClient;

// Setup links between them
fcxDBDataSet1.DataSet := DataSet1;
fcxDataSource1.DataSet := fcxDBDataSet1;
fcxCube1.DataSource := fcxDataSource1;
fcxSlice1.Cube := fcxCube1;
```

```
fcxSliceGrid1.Slice := fcxSlice1;

// clear field list
fcxDataSource1.DeleteFields;

// set cube source type
fcxCube1.CubeSource := fccs_DataSource;

// load data
fcxCube1.Open;
```

2.2.2 Setting up fields of TfcxDataSource

The field list of TfcxDataSource only needs to be specified in the following cases:

- if only some of the datasource fields are needed
- if data needs to be converted
- if data-time fields need to be split into parts (e.g. day, month)
- if several datasources need to be linked

The TfcxDataSource.Fields property contains the field list of the main source.

The attributes of the source field SourceFieldProperties depends on its type - SourceFieldType : TfcxAttributeType. The TfcxAttributeType enumeration is:

```
TfcxAttributeType = (
    fcxsft_Reference,    // a field from source
    fcxsft_Custom,       // a user field
    fcxsft_Date,         // a date field (able to be split into parts)
    fcxsft_Time          // a time field (able to be split into parts)
  );
```

The DataField property describes a source data type, with attributes :

- name and caption of the field in the source
- requirement to convert the data, together with the target data type
- name and caption of the field in the cube.

The DataField property attributes depend on the SourceFieldType.

Code examples:

```
// Load field list
fcxDataSource1.AddFields;

// change display label for the field indexed 2
fcxDataSource1.Fields[2].DataField.CubeFieldDisplayLabel :=
'Customer';

// set a rule to convert 'Population' field to a string
TfcxReferenceDataField(fcxDataSource1.Fields.FieldByName
['Population'].DataField).Convert := True;
```

```
TfcxReferenceDataField(fcxDataSource1.Fields.FieldByName
['Population'].DataField).CubeFieldType := fcdt_String;
```

2.2.3 Creating and adjusting attributes of TfcxDataSource

SplitProperty: TfcxSplitProperty describes how to split a field on its "attributes". Attributes can be parts of date and time fields (year, day, hour etc) or the fields from a data source that is linked to the main data source of the cube. An attribute can have it own sub-attributes. The nesting level of attributes is not limited.

A field can have an attribute CaptionSourceAttribute (to replace value captions from possible values of that attribute) and an attribute OrderSourceAttribute (to order values according to the attribute value order). If those attributes are not set then the field will use its own values as captions and for ordering.

Attributes and the main field are of type TfcxSourceField. TfcxSplitProperty.Attributes contains the list of attributes.

DateSplitPaths and TimeSplitPaths specify which date and time parts are needed for the the field (when the field is either a date or time).

Code examples:

```
var
  ARefField: TfcxReferenceAttributeSFProperties;
  AAttribute: TfcxSourceField;
begin
  // load field list
  fcxDataSource1.AddFields;

  // specify that Day, Month and Year attributes are required for the
'Date1' field
  fcxDataSource1.Fields.FieldByName
['Date1'].SplitProperty.DateSplitPaths := [odt_Day, odt_Month,
odt_Year];

  // create an attribute for the 'IdClient' field caption...

  // create a new attribute for the 'IdClient' field
  AAttribute := TfcxSourceField(fcxDataSource1.Fields.FieldByName
['IdClient'].SplitProperty.Attributes.Add);
  // set correct attribute type
  AAttribute.SourceFieldType := fcxsft_Reference;
  ARefField := TfcxReferenceAttributeSFProperties
(AAttribute.SourceFieldProperties);
  // set the attribute source as the same as the main field source
  ARefField.DataSet := TfcxReferenceSourceFieldProperties
(fcxDataSource1.Fields.FieldByName
['IdClient'].SourceFieldProperties).DataSet;
  // set the field name of the attribute in the source as 'FullName'
```

```
  ARefField.DataField.DataFieldName := 'FullName';
  // set the name of the created attribute as the source of captions
for the field
  fcxDataSource1.Fields.FieldByName
['IdClient'].SourceFieldProperties.CaptionSourceAttribute :=
'FullName';

  // the same task but with the client name taken from another
source...

  // create a new attribute for the 'IdClient' field
  AAttribute := TfcxSourceField(fcxDataSource1.Fields.FieldByName
['IdClient'].SplitProperty.Attributes.Add);
  // set correct attribute type
  AAttribute.SourceFieldType := fcxsft_Reference;
  ARefField := TfcxReferenceAttributeSFProperties
(AAttribute.SourceFieldProperties);
  // set the attribute source as fcxDataSet2 - which is a reference
table containing a key 'Id' and a name 'FullName'
  ARefField.DataSet := fcxDataSet2;
  // key field in the attribute source is 'Id'
  ARefField.IdField.DataFieldName := 'Id';
  // name field in the attribute source is 'FullName'
  ARefField.DataField.DataFieldName := 'FullName';
  // set the name of the created attribute as the source of captions
for the field
  fcxDataSource1.Fields.FieldByName
['IdClient'].SourceFieldProperties.CaptionSourceAttribute :=
'FullName';
end;
```

2.3 Setting up Slice

The TfcxSlice component is used to configure a slice.

Slice fields (TfcxSliceField) are automatically built based on the cube fields.

The slice has containers which may contain region fields (TfcxCommonFieldOfRegion):

- **XAxisContainer** - X-axis, contains fields of TfcxAxisField type
- **YAxisContainer** - Y-axis, contains fields of TfcxAxisField type
- **PageContainer** - filter region, contains fields of TfcxAxisField type
- **MeasuresContainer** - measures, contains fields of TfcxMeasureField type

TfcxAxisField fields are created based on the slice fields.

Measures TfcxMeasureField can be created either based on the slice fields, or based on a FastScript script.

Any slice field can be placed in any container. A measure based on slice fields does not prevent this field being placed in another container at the same time.

2.3.1 Creating Slice structure

To add a dimension (field) to an axis region or to the filter region use the following container methods:

function AddDimension(ASliceField: TfcxSliceField; AName: TfcxString = ''; ACaption: TfcxString = ''): integer;
Adds a dimension based on ASliceField field to the end of the field list of the specified region. Returns the field position in the region's field list.
If the region field based on ASliceField already exists then it will be moved to the specified position in the specified region, otherwise a new region field is created.

procedure InsertDimension(ASliceField: TfcxSliceField; AIndex: integer; AName: TfcxString = ''; ACaption: TfcxString = '');
Inserts a dimension based on ASliceField field to the specified position in the specified region.
If the region field based on ASliceField already exists then it will be moved to the specified position in the specified region, otherwise a new region field is created.

procedure DeleteDimension(AIndex: integer);
Deletes a dimension specified by the index. The region field is destroyed.

To edit the Measures field use the following container methods:

function AddMeasuresField: integer;
Moves the "Measures" field to the end of the specified region. Returns position of the "Measures" field.

function InsertMeasuresField(AIndex: TfcxSmallCount): integer;
Moves the "Measures" field to the specified position of the specified region. Returns position of the "Measures" field.

procedure DeleteMeasuresField;
Deletes the "Measures" field from the specified region. The "Measures" field is automatically moved to the first position of the filters region.

IMPORTANT!

The "Measures" field is a virtual field which is always created but never present in the list of fields of the corresponding container.

To access its properties use the **MeasuresContainer** property of the Slice object.

The position of the "Measures" field in the region is defined by the **MeasuresContainer.Position** property. All the region fields with index equal to or greater than **MeasuresContainer.Position** are shown after the "Measures" field.

The **MeasuresContainer.Container** property defines a container which corresponds to the "Measures" field.

All operations changing the slice structure are best placed between **BeginUpdate** and **EndUpdate** calls. This prevents unnecessary recalculations and rebuilds after each change.

Code example:

```
// begin structure change - suspend recalculations on slice
fcxSlice1.BeginUpdate;
// add slice field indexed 0 to the Y-axis
fcxSlice1.YAxisContainer.AddDimension(fcxSlice1.SliceField[0]);
// add slice field indexed 1 to position 0 of the Y axis
fcxSlice1.YAxisContainer.InsertDimension(fcxSlice1.SliceField[1], 0);
// add a slice field having 'FullName' name to the X-axis
fcxSlice1.XAxisContainer.AddDimenssion(fcxSlice1.SliceFieldByName
['FullName']);
// add 'Measures' field to the X-axis
fcxSlice1.XAxisContainer.AddMeasuresField;
// finish structure change, start recalculations on slice
fcxSlice1.EndUpdate;
```

2.3.2 Measure management

A measure can be created based on either a slice field or a FastScript script.

function AddMeasure(ASliceField: TfcxSliceField; AName, ACaption: TfcxString; AAgrFunc: TfcxAgrFunc): Integer;
Adds a measure based on ASliceField with aggregate function AAgrFunc. Returns the position of the measure in the container.

function AddCalcMeasure(AName, ACaption: TfcxString; AAgrFunc: TfcxAgrFunc; AScriptFunctionName: String; AScriptFunctionCode: TfcxString): Integer;
Adds a calculated measure based on script function AScriptFunctionName with aggregate function AAgrFunc. AScriptFunctionCode - the function's code. Returns the position of the measure in the container.

function AddMeasure(AField: TfcxMeasureField): Integer;
Adds specified measure AField. Returns the position of the measure in the container.

procedure InsertMeasure(ASliceField: TfcxSliceField; AName, ACaption: TfcxString; AAgrFunc: TfcxAgrFunc; AIndex: TfcxSmallCount);
Inserts a measure based on ASliceField with aggregate function AAgrFunc in the specified container position.

procedure InsertCalcMeasure(AName, ACaption: TfcxString; AAgrFunc: TfcxAgrFunc; AScriptFunctionName: String; AScriptFunctionCode: TfcxString; AIndex: TfcxSmallCount);
Inserts a calculated measure based on script function AScriptFunctionName with aggregate function AAgrFunc in the specified container position. AScriptFunctionCode - the function's code.

procedure InsertMeasure(AField: TfcxMeasureField; AIndex: TfcxSmallCount);
Inserts specified measure AField in the specified container position.

procedure DeleteMeasure(AMeasureIndex: TfcxSmallCount; ADoStopChange: Boolean = False);
Deletes the measure with the specified index.

MeasuresContainer methods and properties allow access to and editing of measures.
Measures can be hidden. Hidden measures are also calculated.

function MoveMeasure(AFromIndex, AToIndex: TfcxSmallCount): boolean;
Moves measure within the container.

property Measures[AIndex: TfcxSmallCount]: TfcxMeasureField;
Measure access property.

TfcxMeasureField properties and methods allow editing of measure properties:

property Visible: Boolean;
Measure visibility.

property DisplayAs: TfcxDisplayAs;
Display style.

All operations changing the slice structure are best placed between **BeginUpdate** and **EndUpdate** calls. This prevents unnecessary recalculations and rebuilds on each change.

Code examples:

```
// begin structure change - suspend recalculations on slice
fcxSlice1.BeginUpdate;
// add a measure based on slice field indexed 3 and aggregate function
af_Sum
fcxSlice1.MeasuresContainer.AddMeasure(fcxSlice1.SliceField[3],'Sum1',
'Income', af_Sum);
// add a calculated measure, which calculates half of Income
fcxSlice1.MeasuresContainer.AddCalcMeasure('Calc1', 'Half of Income',
af_Formula, 'CalcScript1', 'Result := measures
[''Sum1''].currentvalue / 2');
// move the measure indexed 1 to position 0
fcxSlice1.MeasuresContainer.MoveMeasure(1, 0)
// hide the measure indexed 1
fcxSlice1.MeasuresContainer.Measures[1].Visible := False;
// finish structure change, start recalculations on slice
fcxSlice1.EndUpdate;
```

2.4 Filter management

Filters are needed to limit the quantity of calculated data, according to specified criteria.

Filters can be edited using slice methods and properties.

Code examples:

```
// clear filter indexed 3 of the slice field indexed 0
fcxSlice1.SliceField[0].UVFilterOfValue[3] := False;

// begin changing the filter
fcxSlice1.SliceFieldByName['FirstName'].BeginUpdateFieldFilter;
// inactivate filter for all values of 'FirstName' field
fcxSlice1.SliceFieldByName['FirstName'].SetNoneFilter;
// activate filter with value 'Sergey' for 'FirstName' field
fcxSlice1.SliceFieldByName['FirstName'].UVFilterOfValue['Sergey'] :=
True;
// activate filter with value indexed 12 for 'FirstName' field
fcxSlice1.SliceFieldByName['FirstName'].UVFilterOf[12] := True;
// finish changing the filter (apply changes)
fcxSlice1.SliceFieldByName['FirstName'].EndUpdateFieldFilter;

// activate filter only for the value indexed 4 for the slice field
indexed 0
fcxSlice1.SliceField[0].UVSingleIndex := 4;

// invert filter activation for the values of slice field indexed 0
fcxSlice1.SliceField[0].InverseFilter;

// set filter activation according to the criterion specified by
ARange
fcxSlice1.SliceField[0].SetRangeFilter(ARange);

// set filter type to "radio"
SliceField[1].UVFilterType := uvft_Single;
```

2.5 Group management

Groups improve the representation of the data.

Groups can be edited using the methods and properties of a slice field and the group manager (GroupManager) of the slice field.

Groups can be edited (created, changed, etc) after turning on the grouping mode (CanGroup) for the slice field.

Code examples:

```
// turn on the grouping mode
ASliceField.CanGroup := True;
// check if we can use groups
if ASliceField.CanGroup then
begin
  // create a group with name 'Group1'
  AGroupIndex := ASliceField.GroupManager.CreateGroup('Group1').Index;
  // add values with index 3 into the group with index AGroupIndex
  ASliceField.GroupManager.AddUVInGroup(3, AGroupIndex);
  // add values 30 to the group with index AGroupIndex
  ASliceField.GroupManager.AddUVValueInGroup(30, AGroupIndex);
  // create pre-defined group "Others"
  ASliceField.GroupManager.CreateOtherGroup;
end;
```