StarOffice Programmer's Tutorial

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part Number 806-5845
May 2000

Contents

CHAPTER 1

Introduction

We wish to acknowledge Christian Kirsch as the author of this tutorial..

1.1 What this book is about

This tutorial provides you with some recipes that should help you to program StarOffice. Although this software suite offers most of the features required in today's offices, there are always tasks that are better handled with a small program. Most notably, a task requiring the same mouse clicks or text entries over and over again should be done by a program.

StarOffice API permits you to automate tasks that would be tedious or hard to perform within StarOffice. For example, you can change many documents in one step or send email to a group of recipients selected from a database automatically. Furthermore, you can use StarOffice API to integrate StarOffice components into your own programs. The details of this integration are beyond the scope of this book, however.

After you have read this book, you should have a fairly good knowledge of all StarOffice API components. However, we can't provide you with detailed information for all services and interfaces. But you'll learn how to use the StarOffice API Reference Manual, available as part of the StarOffice SDK, to find all the details you are interested in.

The StarOffice Programmer's Tutorial consists of several parts that should be read in sequence, since each chapter builds on the previous ones.

- This introduction contains general information on the whole book.
- Chapter 2 presents the parts that make up StarOffice API.
- In Chapter 3, you'll see the basic techniques used in StarOffice API.

- Chapter 4 contains examples of building blocks. These short programs can be used stand-alone or as parts of bigger applications.
- Finally, Chapter 5 features several larger programs that show you how to put the building blocks together.

We chose the name "StarOffice Programmer's Tutorial" to emphasize that this is neither a reference nor a user's manual. Because we are focusing on examples that illustrate important aspects and usages of StarOffice API, this is not a complete description of all StarOffice API features. After you have read this book and worked through the examples you should understand the structure of StarOffice API and be able to write your own programs using the StarOffice API Reference Manual. Most printed sample code is stripped of comments, introductory and finalizing statements (like variable declarations and return statements).

You can find all the examples in the package accompanying this Programmer's Tutorial. If a certain example is too large, we'll only show the more interesting parts of the programs in print. All examples will explore only those StarOffice API features provided by StarOffice. Put in other words: The StarOffice Programmer's Tutorial does not contain any guidance that permit you to extend StarOffice API with your own services.

If you download this document from http://soldc.sun.com/staroffice you will also be able to download the examples.

If you have any suggestions on improving this document send them to StarOfficeAPItutorialfeedback@eng.sun.com.

1.2 Who should read this book

If you want to program StarOffice or if your boss wants you to do that, this book is for you. You should already have some programming experience, since we will not explain what a variable is, what "string" stands for or how to pass parameters to a function. Telling you all that would have blown up this book to at least twice its current size.

There are some situations when programming StarOffice is advantageous:

- You want to customize its behavior. In a big organization, it might be required that certain users work with customized menus or that they find themselves in a particular application whenever they start StarOffice.
- Suppose you wanted to change the tab settings for all but one paragraph style - doing this with dialogs is tedious, but it's a simple job for a small program.
- If you are using StarPortal, you can use StarOffice API to access its components (aka StarOffice Beans) and incorporate them in your application.

You do not have to know a particular programming language to understand StarOffice API. Some basic knowledge of object oriented paradigms and component technology might be advantageous but it is not strictly required.

If you are using StarOffice only for the occasional letter every other month, you have no need for StarOffice API. Similarly, if you have never written a single line of code, not even `goto()`-ridden Basic, this book is not for you.

Even if you are not planning to develop software, you might be interested to read the first three chapters of this book to get an idea of StarOffice API. Just go on reading than, but be aware that the fourth and fifth chapter might seem too technical to you.

1.3 What is StarOffice API

StarOffice API is the "office component model" on which StarOffice is based. This is certainly a nice enough term, but what does it mean? First of all, StarOffice API is *not* a programming language. It is an abstract definition of all the objects and their interfaces that you can use in your programs.

The crucial term here is "abstract definition": StarOffice API does not implement objects, interfaces etc. itself. This implementation is provided by StarOffice, but you could as well write your own code to implement part or all of StarOffice API. You can’t start writing your own implementation of StarOffice API right now, bu it will be feasible in the future. However, as mentioned before, this book will not explain you how to do that.

Although StarOffice implements StarOffice API, not all parts of the office suite are accessible from StarOffice API. For example, you can *access* controls in a document from StarOffice API, but you can’t *build* dialogs with StarOffice API. Eventually, all aspects of StarOffice will be integrated with StarOffice API.

Since StarOffice API provides only concepts, it can be used from different programming languages. Currently, it provides access for StarBasic, StarScript (aka ECMAScript), Java, and C++. Interfaces for other languages can be build, but this is not a topic of the tutorial.

1.4 How you can use StarOffice API

As said before, StarOffice API is no programming language. It only provides interfaces and services that you use in your program. You can employ several languages in which you write StarOffice API programs. This tutorial will show you

how to use StarBasic, but you can use StarScript (aka ECMAScript), C(++) or Java as well.

Although in this tutorial we will focus on the usage of StarBasic, you will be able to use the examples as patterns for developing StarOffice API Java applications as well. This language is particularly important because it permits you to access and provide JavaBeans. These are components used by the StarPortal. With Java, you could thus integrate a StarOffice Writer Bean in your own applet.

StarBasic is similar to other dialects of BASIC, for example the one used by Microsoft in their office products. It provides some object orientation and most simple data types like real and integer number, booleans, strings and arrays. StarBasic offers some shortcuts for StarOffice API so that it might be easier to use than other programming languages.

1.5 Where to find additional information

The primary source of StarOffice API information is the StarOffice API Reference Manual. This manual provides all details on StarOffice API services, interfaces and objects. Most of it is generated automatically from the StarOffice API source files and is thus complete, accurate and up to date. However, it is not targeted towards any particular programming language. You should read Section 3.5 "Understanding the StarOffice API Reference Manual" on page 25 for StarBasic specific information

The website of Sun Microsystems `http://www.sun.com/staroffice/` and the newsgroups hosted by Sun (`news://starnews.sun.com`) provide additional help and information. You should check them regularly for up-to-date information.

1.6 Typographic conventions

We'll use different fonts for different items in this book.

- `Method and function names()`
- `Datatypes, variables, and property names`
- `Literal text`
- `File names`
- `Programming examples`

Certain conventions are used in the StarBasic examples throughout this book. The first letter of a variable always indicates its type as described in the following table.

Letter	Meaning
a	Structure
b	Boolean (`TRUE` or `FALSE`)
e	Enumeration. This variable can only have one of a limited set of values.
f	Float or double
m	Array (aka sequence)
n	Integer or long
o	Object, service, or interface
s	String
x	Interface, to indicate that only operations of a particular interface of an object are used
v	Variant, `Any`

CHAPTER 2

StarOffice API - Concepts

2.1 Services and Interfaces

In StarOffice API, a "service" is an abstract concept providing certain interfaces and properties. Every implementation of a particular service must provide the same interfaces. An interface is a collection of methods that provide a certain functionality.

Let's use a car to illustrate these concepts. The abstract car provides two concrete interfaces: `XAccelerationControl` and `XDrivingDirection`. Both interfaces export methods, the first one for accelerating and slowing down, the second one to turn the car left or right. In addition to these interfaces, the service `Car` has the properties `Color` and `Seats`.

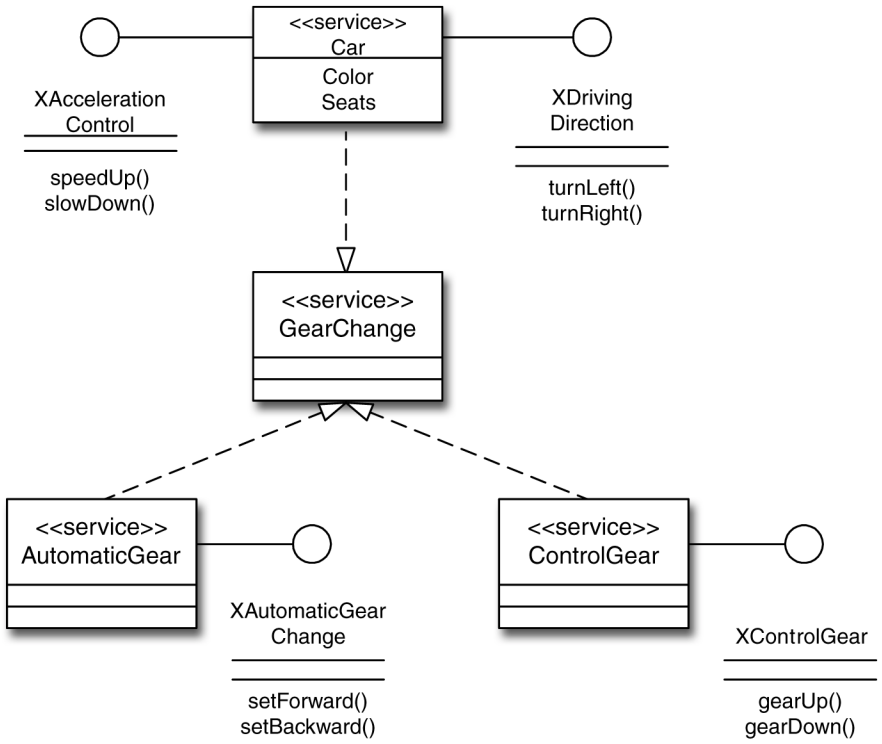

Figure 2–1 StarOffice API service concept

The `Car` service implements another service called `GearChange`. This service can be implemented by either `AutomaticGear` or `ControlGear`Each of these services contains one interface exporting different methods depending on the type of

`GearChange` that is implemented: `setForward()` and `setBackward()` for `XAutomaticGearChange`, `gearUp()`and `gearDown()` for `XControlGear`.

2.2 Modules

Modules group services, interfaces, types, enumerations and data structures. Some StarOffice API modules are `text`, `sheet`, `table`, and `drawing`. Although they correspond with certain parts of StarOffice, there is no strict one-to-one relationship between modules in StarOffice API and StarOffice components: modules like `style` and `document` provide generic services and interfaces that are not specific for one part of StarOffice.

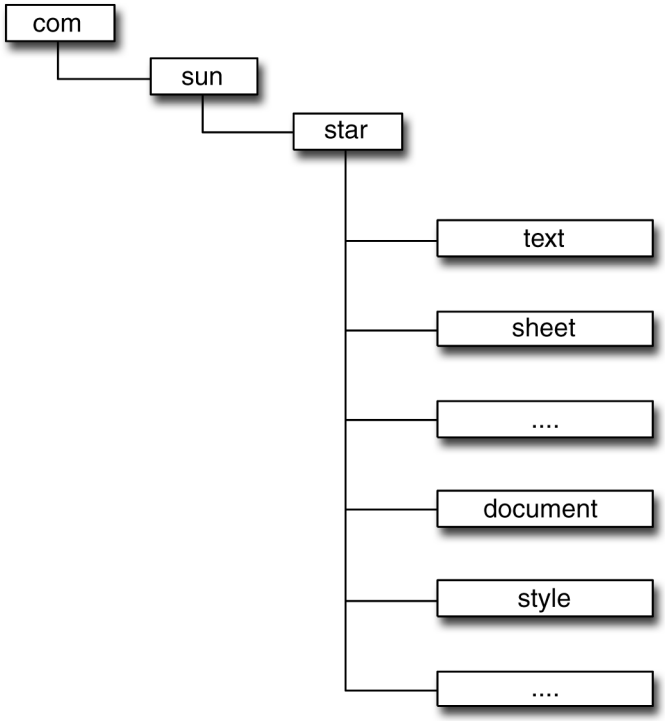

Figure 2–2 StarOffice API module structure

2.3 Components

Components implement StarOffice API services. You are never dealing directly with them when you program in StarOffice API. They are accessible as beans which you can incorporate into your own programs. This book will not cover how to do that.

CHAPTER 3

Using StarOffice API - Basics

This chapter explains how to obtain a service. Every StarOffice API program has to do this at least once, so this is a very important aspect. Furthermore, we'll show you how to set and get properties, how to access collections and how to handle events. You will finally read about the StarOffice Reference Manual and how to read it.

3.1 Getting a Service

Services are paramount to programming in StarOffice API: You need interfaces for virtually anything, and interfaces are provided by services. Therefore, the first thing you do in most StarOffice API programs is to acquire a service.

- Independent services don't need an environment to operate on. For example, if you want to define new colors, you would use the `ColorTable()` service. This type of service is created using `createUnoService()`.
- Some services can only operate inside of a certain environment, these are context-dependent services. For example, you can't search for text outside of a document nor can you create a cell without a spreadsheet. This type of services is created by the generic `Desktop()` service. Another class of dependent services are those providing no interfaces at all. The `PrinterDescriptor()` service, for example, contains no interfaces, it is only useful to set and get properties.

3.1.1 Independent Services

To get a StarOffice API service that works independent of a document, you call `createUnoService()` with the service's name as parameter. All services start with

"com.sun.star". The following part is the name of the module proper, for example `text` or `sheet`. More accurately, `com.sun.star.text` is the complete name of the module. The last part of the parameter is the service itself. This naming reflects the hierarchical structure of StarOffice API. Module `com.sun.star` contains the module `text` which itself contains a service `TextDocument`.

To get access to the ColorTable, you'd use

```
Dim oColorTable As Object

oColorTable=createUnoService("com.sun.star.drawing.ColorTable")
```

3.1.2 Context-dependent services

Services that operate on documents have to be created with the help of the `Desktop()` service or through the active document in StarBasic. For example, to use the text service, you'd have to open an existing text document first:

```
Dim mNoArgs()
Dim oDesktop, oDocument As Object
Dim sUrl As String

oDesktop  = createUnoService("com.sun.star.frame.Desktop")
sUrl = _
 "file:///home/testuser/Office52/work/test.sdw"
oDocument = _
 oDesktop.loadComponentFromURL(sUrl,"_blank",0,mNoArgs())
```

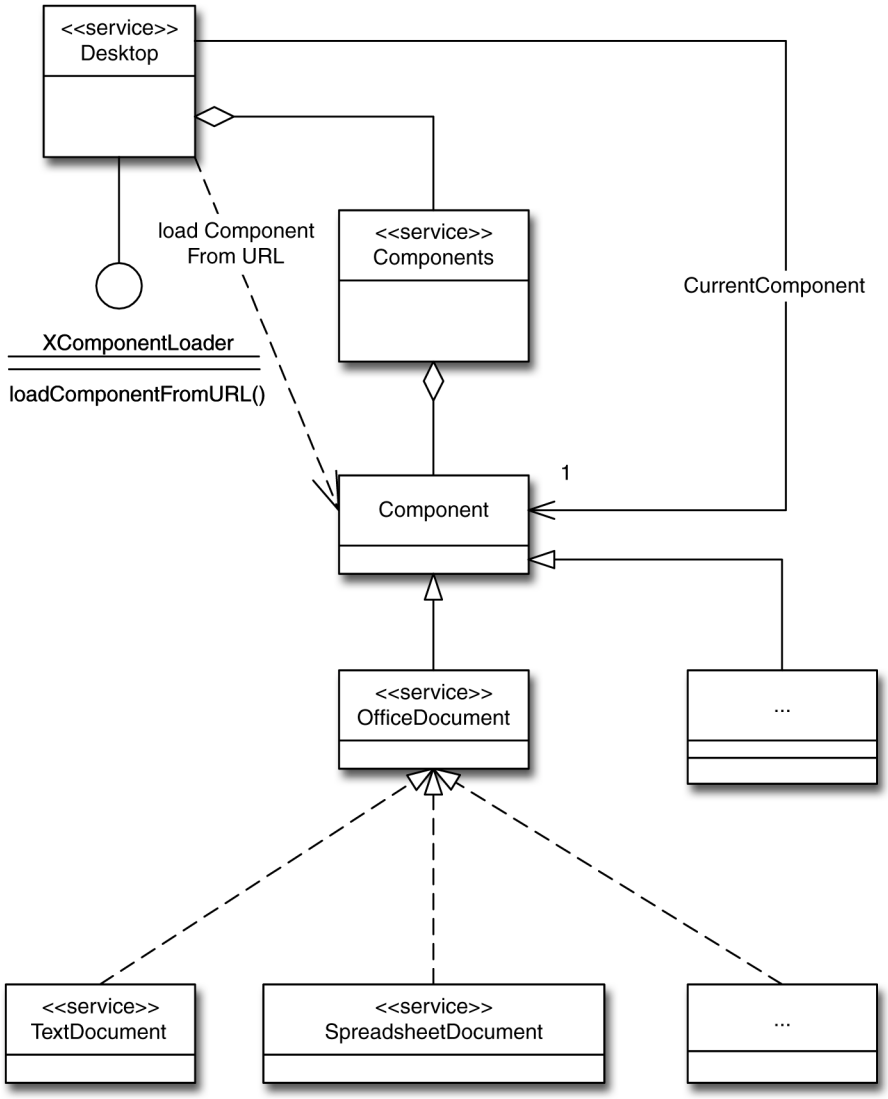

Figure 3–1 Opening an OfficeDocument

As you can see, you first obtain the `Desktop()` service. You then use the `loadComponentFromURL()` method of its `XComponentLoader` interface to open an existing document.

Alternatively, you could create a new StarOffice Calc or StarOffice Writer document like this:

```
Dim oSpreadsheetDocument As Object
Dim oTextDocument As Object
```

```
oDesktop  = createUnoService("com.sun.star.frame.Desktop")
sUrl ="private:factory/scalc"
oSpreadsheetDocument = _
 oDesktop.loadComponentFromURL(sUrl,"_blank",0,mNoArgs())
sUrl ="private:factory/swriter"
oTextDocument = _
 oDesktop.loadComponentFromURL(sUrl,"_blank",0,mNoArgs)
```

The format and meaning of URLs used in StarOffice API are explained in Section 3.5.4 “URLs” on page 28.

When StarOffice API opens a document, it recognizes its type either from its file name (for existing documents) or from the name of the factory (for new documents). To open a file with a filter enforcing a certain format, cf. Section 4.2.1 “Importing other Formats” on page 41.

Finally, you can use an active document to obtain access to its interfaces:

```
Dim oText As Object

oDocument = ThisComponent
oText = oDocument.Text
```

gets the `Text()` service of the current active `TextDocument` service.

3.1.3 Service Factories

Some services, especially the document services, provide a special `XMultiServiceFactory` interface which can be used to acquire new services. You can call `createInstance()` of this interface with the service’s name as parameter to acquire a service.

To create a new RectangleShape object for your Calc document, you’d use

```
Dim oRectangleShape As Object

oRectangleShape = _
 oCalcDocument.createInstance("com.sun.star.drawing.RectangleShape")
```

Similarly, to create a new paragraph style, you’d use

```
Dim oStyle as Object

oStyle=oDocument.createInstance("com.sun.star.style.ParagraphStyle")
```

3.1.4 Which services does StarOffice API provide

The list of services provided by StarOffice API is too long to include here. We will just mention some of the available modules and the most important services they offer

- com.sun.star.chart contains the services for charting. The most important one is ChartDocument() that specifies the data to use in the chart and some general characteristics.
- com.sun.star.drawing collects all services used for drawing line, rectangles, circles etc.
- com.sun.star.frame contains the Desktop() service. You use this service to open existing documents or create new ones.
- com.sun.star.presentation provides all services to create and work with presentations.
- com.sun.star.sheet contains services for spreadsheets. Its SpreadheetDocument() service is used to work with spreadsheets.
- com.sun.star.table provides all services for tables in text documents and spreadsheets.
- com.sun.star.text groups the services dealing with text documents. The TextDocument() service provides all interfaces needed to work with text documents.

3.2 Properties

Properties (also called attributes sometimes) are values that determine the characteristics of a service. For example, the fill color is a property of the ShapeDescriptor service. Some services have a fixed set of properties (again, ShapeDescriptor is an example). Others use varying property sets, since certain properties need not always exist.

Properties are "name-value" pairs. The "name" is the name of the property, while "value" contains its current value. For example, FillColor is the name of the property describing the fill color in a ShapeDescriptor and its value can be something like RGB(255,0,0)() (which is full red). If the number of properties is fixed, you can use simple assignments in StarBasic to set or retrieve their values:

```
Dim nOldColor As Long

nOldColor = oRectangleShape.FillColor
oRectangleShape.FillColor = RGB(255,0,0)

REM ...

oRectangleShape.FillColor = nOldColor
```

In this example, we save the current setting of the fill color in nOldColor and then change the fill color to red. Later, the old color is restored.

Sometimes you are not dealing with single properties, but with a sequence of them. Such sequences are implemented as arrays in StarBasic. For example, to open a

document, you will have to provide several properties to `loadComponentFromURL()`. The approach is straightforward:

```
Dim mFileProperties(2) As New com.sun.star.beans.PropertyValue

mFileProperties(0).Name="FilterName"
mFileProperties(0).Value="swriter: StarWriter 5.0"

mFileProperties(1).Name="AsTemplate"
mFileProperties(1).Value=true
```

Note that in StarBasic's `Dim()` statement you specify the highest array index, not the number of elements. See Section 3.5.2.2 "Structures" on page 26 for related information.

3.3 Collections and Containers

Some of the StarOffice API components are bundled in "collections" and "containers". For example, the tables in a spreadsheet are a considered as a "collection of spreadsheets". Similarly, a text document can be looked at as a "collection of paragraphs". Collections come in three flavours:

1. Each element has a name, like the tables in a spreadsheet (named access).
2. Elements have no names but can be accessed by index.
3. Elements have no names and can be accessed only in sequential order (enumeration access).

Some containers provide several access methods. For instance, tables in a sheet document can be addressed either by name or by index.

3.3.1 Named access

To access collections with named elements, you use the `XNameAccess` interface. Although this may sound complicated, its usage is fairly natural:

```
Dim oSheets, oSheet As Object

oSheets = oCalcDocument.Sheets
oSheet = oSheets.getByName("Sheet1")
Dim oStyleFamilies As Object
oStyleFamilies = oDocument.StyleFamilies
oParagraphStyles = oStyleFamilies.getByName("ParagraphStyles")
```

stores the table with the name "Sheet1" in `oSheet`. We assume here that `oSheets` is a collection of spreadsheets. The next lines copy the names of all paragraph styles to `oParagraphStyles`.

You can find all names in a named collection with the method `getElementNames()` which returns a sequence of strings (see Section 3.5.2.3 “Sequences” on page 27. The following code snippet displays the names of all style families for a document:

```
oStyleFamilies = oDocument.StyleFamilies
mFamilyNames = oStyleFamilies.getElementNames
For i%=LBound(mFamilyNames) To UBound(mFamilyNames)
  Print mFamilyNames(i%)
Next i%
```

To find out if a named collection contains a certain element, you can use the `hasByName()` method. It returns `TRUE` if the element exists, `FALSE` otherwise.

While the methods mentioned so far are available for all objects providing the `XNameAccess()` interface, some of the more advanced facilities are only available with the `XNameContainer()` interface. It provides two methods to add and remove elements by name. To add a new element to a name container, you’d use `insertByName()` like this:

```
oParagraphStyles.insertByName("myParagraphStyle", oStyle)
```

and to remove an element, you’d write

```
oParagraphStyles.removeByName("myParagraphStyle")
```

Please note that some interfaces might overwrite the `insertByName()` method with its own version, requiring additional parameters. You should therefore always check the reference manual. A third interface, namely `XNameReplace()`, permits you to replace existing elements:

```
oParagraphStyles.replaceByName("myParagraphStyle", oStyle)
```

Notice that the object you replace the old one with must exist before you can use `replaceByName()`.

3.3.2 Index access

To access indexed collections, you use the `XIndexAccess` interface like this:

```
oSheets = oCalcDocument.Sheets
oSheet = oSheets(0)
```

This code gets the collection of sheets from the current document and assigns the first of them to `oSheet`. It will work only if the current document is a spreadsheet. The simple usage of parentheses to access an indexed collection is some syntactic sugar provided by StarBasic. The complete method call would be `oSheet = oSheets.getByIndex(0)()`.

Warning - Please note that the first element of an indexed collection is numbered zero, not one. To access all elements in a collection with three elements, you'd use something like

```
For i%=0 To 2
  oSheet=oSheets(i%)
  REM do something with oSheet
Next i%
```

You can determine the number of elements in an indexed collection with the `getCount()` method:

```
For i%=0 To oSheets.getCount() - 1
  REM do something with oSheets(i%)
Next i%
```

To add a new element to an indexed collection, you can use the method `insertByIndex()`, if the service provides the `XIndexContainer` interface. To append an element, use the highest index plus one. In some cases, `insertByIndex()` might require additional parameters, so you should always make sure how to call it by checking the StarOffice API Reference Manual for the particular service. Similarly, to remove an element you use `removeByIndex()`, provided that the service provides the `XIndexContainer` interface.

Finally, you can replace an element with another object if the object provides the `XIndexReplace()` service.

```
oList.replaceByIndex(0,oNew)
```

This replaces the first element of `oList` with `oNew`, which you must have created before you can call `replaceByIndex().()`

3.3.3 Enumeration access

Finally, to access enumerated collections, you use the `XEnumerationAccess` interface. The elements of an enumerated collections can only be accessed sequentially, starting from the first one. An example looks like this:

```
oParagraphEnumeration = oDocument.Text.createEnumeration
While (oParagraphEnumeration.hasMoreElements)
  oParagraph = oParagraphEnumeration.nextElement
  REM do something with oParagraph
Wend
```

This code can be used to step through all paragraphs in a text document.

3.4 Events

Events are a well known concept in GUI models. They enable the application to react to asynchronous input: When the user clicks on a button, the underlying GUI system forwards this event to a previously registered procedure inside the application, the event handler. This procedure "knows" how to react on the click.

StarOffice API uses a similar concept which is closely modeled after the one used in JavaBeans. Events are, for example, a user typing something into an input field or an e-mail arriving in the inbox. A program interested in events has to register a handler, called "listener" in StarOffice API. These listeners are indispensable if you are working with local or remote files, newsgroups, or email in StarOffice API.

Many event listeners handle several events at once. They use one of the input parameters (often called `event`) to determine the kind of event passed to them. A notable exception to this rule are small subroutines handling user clicks and input in StarBasic dialogs. Each of them is associated with a certain control in the dialog and handles only the user interaction with this control. All this happens outside of StarOffice API, but you'll probably put these event handlers in the same StarBasic module as the other subroutines.

3.5 Understanding the StarOffice API Reference Manual

The StarOffice API Reference Manual is generated automatically from the specification of StarOffice API. This guarantees that it is up-to-date and accurate. But as StarOffice API is not written for any particular programming language, the manual doesn't provide specialized information needed to write StarBasic programs. This chapter sheds some light on the terminology and structure of the StarOffice API Reference Manual and explains how to translate its terminology into StarBasic statements and types.

3.5.1 Properties

All properties are accessed directly by name in StarBasic. To set the fill color of a rectangle, you write for example

```
oRectangleShape.fillColor = RGB(255,0,0)
```

and to retrieve the current font name, you use

```
Dim sFont As String

sFont = oStyle.CharFontName
```

3.5.2 Types

StarOffice API implements its own datatypes. These types fall in three categories: enumerations, structures, and sequences. We'll explain each of these categories in the following sections.

3.5.2.1 Enumerations

Enumerations are sets of named constants. For example, an enumeration `color` might consist of `red`, `blue`, and `green`. A variable of type `color` could only represent one of these values. Usually, the values are small integers (0, 1, ...), but this is an implementation detail. You should always use the name of the enumeration constant, never the number itself. Enumerations are represented like this in the StarOffice API Reference Manual:

```
enum VerticalAlignment: VerticalAlignment

Field Summary

  TOP

  MIDDLE

  BOTTOM
```

This example is from the module `com.sun.star.style`. To specify a middle vertical alignment, you'd have to write `com.sun.star.style.VerticalAlignment.MIDDLE`. Case is important here. Using `com.sun.star.style.VerticalAlignment.middle` instead would thus lead to unexpected results.

3.5.2.2 Structures

The use of structures is straightforward. You append the name of the structure element to the variable name like for properties:

```
Dim aProperty As New com.sun.star.beans.PropertyValue

aProperty.Name="Font"
aProperty.Value="Times"
```

sets the two elements in a `com.sun.star.beans.propertyValue` structure.

Due to design decisions in StarBasic, you can't set the elements of a structure inside an object directly. Instead, you have to use

```
aStruct.element1 = value
oObj.structMember1 = aStruct
```

Here aStruct is a structure with at least one element element1. This element is set to value. The whole structure is then assigned to the structMember1 of oObj, assuming that oObj is a structured object. Trying to assign a value to an element inside an object's struct in one step will not work in StarBasic. Consequently, you can't say

```
REM The following code will NOT work
oObj.structMember1.element1 = value
```

3.5.2.3 Sequences

sequence <aType> before a value means that you have to provide an array of values or that an array of values is returned. For example sequence <string> getAvailableServiceNames() means that the method getAvailableServiceNames() returns an array of strings. To step through it, you'd use something like

```
Dim n As Integer
Dim mServiceNames As Variant

mServiceNames=oDocument.getAvailableServiceNames()
For n = LBound(mServiceNames) To UBound(mServiceNames)
  print mServiceNames(n)
Next n
```

getAvailableServiceNames() is provided by the interface XMultiServiceFactory, the sequence of strings is the list of available Services that can be created with this factory. If you are only interested in the number of elements, you can use the getCount() method. On the other hand, if a method expects a <sequence> of parameters, you'll have to create the appropriate array first. If the base type of the array is a simple one like string, you'd use something like

```
Dim mString(9) As String
```

Objects like mPropertyArray are defined in StarBasic using the new() operator and the name of the object:

```
Dim mPropertyArray(9) As New com.sun.star.beans.PropertyValue
```

Warning - This statement creates an array with 10 elements, numbered from 0 to 9. If you have some programming experience with C or C++, you should keep in mind that you specify the *highest index* for an array, not the number of elements.

If a method expects a sequence as input parameter, you have to pass it with trailing empty parentheses in StarBasic, for example

```
Dim mFileProperties(0) As New com.sun.star.beans.PropertyValue
```

```
Dim sUrl As String

oDesktop  = createUnoService("com.sun.star.frame.Desktop")
sUrl =  "file:///home/testuser/Office52/work/csv.doc"
mFileProperties(0).Name = "FilterName"
mFileProperties(0).Value = _
 "scalc: Text - txt - csv (StarCalc)"
oDocument = oDesktop.loadComponentFromURL(sUrl,"_blank",_
 0,mFileProperties())
```

The parameter `mFileProperties` is a sequence, which is indicated by the `()` following its name.

3.5.3 Constants

Similarly, `constants` are used in StarBasic by writing down their full name. For example, the Reference Manual for `text` module contains this part

```
constants ControlCharacter
{
  const short PARAGRAPH_BREAK = 0;
  const short LINE_BREAK = 1;
  const short HARD_HYPHEN = 2;
  const short SOFT_HYPHEN = 3;
  const short HARD_SPACE = 4;
};
```

To specify a line break in StarBasic, you'd use `com.sun.star.text.ControlCharacter.LINE_BREAK`. `Text` is the module, `ControlCharacter` the name of the constants group and `LINE_BREAK` the name of the constant.

If you are used to the fact that StarBasic treats all variables, functions, methods, and subroutines as case insensitive, you must be very attentive when you use constants. Their names are *always* case sensitive. This means that using `com.sun.star.text.controlCharacter.LINE_BREAK` will generate a runtime error, because the correct name of the constants group is `ControlCharacter` with an uppercase C.

3.5.4 URLs

Some functions expect a URL as a parameter, for example `loadComponentFromUrl()`. Most URLs are written as usual. However, to specify a file on a Windows machine, you have to provide the drive like this: `"file:///E|/ ..."`. "E" is the drive letter.

Some URLs are used to create empty documents of a certain type. For example `"private:factory/swriter"` is an URL that causes StarOffice API to create an empty StarOffice Writer document. All these URLs begin with `private:factory:`.

The next part describes the type of document to create: `swriter` for StarOffice Writer, `scalc` for StarOffice Calc, `sdraw` for StarDraw.

Another kind of URL refers to StarOffice concepts. For example, `".chaos/news-box"` refers to the newsgroup folder in StarOffice and `".chaos/out-box"` designates the folder for outgoing email.

The following table lists the URLs used in StarOffice API. The first part of an URL (before the colon) is the service. This should not be confused with a StarOffice API service. Most service names in URLs are used throughout the internet, while StarOffice API services have no meaning outside of StarOffice API.

Service	Meaning	Example
http:	Get an HTML file from a local or remote machine	`http://www.sun.com/staroffice` retrieves the staroffice home page from www.sun.com.
ftp:	Get any file from a local or remote machine, usually via anonymous FTP.	`ftp://ftp.uu.net` opens an anonymous FTP connection to ftp.uu.net.
file:	Get a file from the local machine	`file:///home/ck/.emacs` retrieves the file `.emacs` from the directory /home/ck on the local machine.
news:	Establish a connection to an NNTP-News server	`news://newshost.somewhere.com` connects to the NNTP server on the machine newshost.smoewhere.com
private:factory:	Private URL used in StarOffice API to create documents.	`private:factory/swriter` creates a StarOffice Writer document.

CHAPTER 4

Using StarOffice API - Building blocks

This chapter explains some basic techniques useful for StarOffice API applications. Some of the examples are not standalone programs but building blocks which you can fit together with others to build an application. They show you how to solve the basic tasks in each StarOffice component. We'll occasionally use UML diagrams to illustrate the interfaces and services. Appendix A explains the meaning of these diagrams.

4.1 Styles

Styles are collections of formatting attributes that are accessible under a common name. If you have ever worked with a word processor, you are probably already familiar with styles: You use a `heading1` style for first level headings, `heading2` for second level headings, and `body` or `standard` for normal text. If you alter one aspect of the style, all paragraphs using it change as well. This makes it possible to change the font for all level one headings from Times to Helvetica or to change the font size for all third level headings with one single command.

Usage of styles is not limited to paragraphs, nor even to text documents. Frames, cells in a spreadsheet, graphics shapes, even single characters can be formatted with styles. Since they are ubiquitous, we present them here rather then in the sections on text or spreadsheets. Some of the examples here will contain code that becomes clear only later when you read the section on the corresponding module.

4.1.1 Style basics

For the following examples, we will assume that you have a small text document containing a headline and just one paragraph of text. It can be created like this:

```
Global oDesktop As Object
Global oDocument As Object
Global oText As Object
Global oCursor As Object
Global oStyleFamilies As Object
Global oParagraphStyles As Object
Global oStyle As Object
Global n As Integer

Sub style_init
  Dim mNoArgs() REM Empty Sequence
  Dim sMyText As String
  Dim sUrl As String

  oDesktop  = createUnoService("com.sun.star.frame.Desktop")
  sUrl = "private:factory/swriter"
  oDocument = oDesktop.LoadComponentFromURL(sUrl,"_blank",0,mNoArgs)
  oText = oDocument.Text
  sMyText = "A very short paragraph for illustration only"
  oCursor = oText.createTextCursor()
  oText.insertString(oCursor,"Headline",FALSE)
  oText.insertControlCharacter(oCursor,_
   com.sun.star.text.ControlCharacter.PARAGRAPH_BREAK,FALSE)
  oText.insertString(oCursor,sMyText, FALSE)
End Sub
```

The new document contains two paragraphs, both formatted with the `standard` style. Don't worry about the details here, the meaning of `insertString()` and the other methods will be explained in the section on text. To change the first paragraph so that it uses a heading style, you'd use these lines:

```
oCursor.gotoStart(FALSE)
oCursor.gotoEndOfParagraph(TRUE)
oCursor.paraStyle = "Heading"
```

This code first selects the first paragraph, which contains only the single word `Headline`, by moving the cursor to the start of the document and then to the end of this paragraph. We'll explain cursors in more detail later (see Section 4.3.2 "Moving around" on page 51). Finally, the `paraStyle` property of the paragraph is set to the name of the `heading` style. This last step effectively applies the style to the paragraph.

The approach just shown works if you are certain about the predefined paragraph styles. However, in a German version of StarOffice, a heading style would be called `Überschrift` or `Titel`. If you are not sure which paragraph styles are available, you can find out by:

```
Dim sMsg As String

oStyleFamilies = oDocument.StyleFamilies
oParagraphStyles = oStyleFamilies.getByName("ParagraphStyles")
```

```
For n = 0 To oParagraphStyles.Count - 1
  sMsg=sMsg + oParagraphStyles(n).Name + " "
Next n
MsgBox sMsg,0,"ParagraphStyles"
```

The styles are grouped in named families. Which families are available depends on the type of document you are working on. For example, text documents provide `PageStyles`, `CharacterStyles`, `FrameStyles`, and `NumberingStyles` besides the `ParagraphStyles`. A spreadsheet, on the other hand, knows about `CellStyles` and `PageStyles` families only.

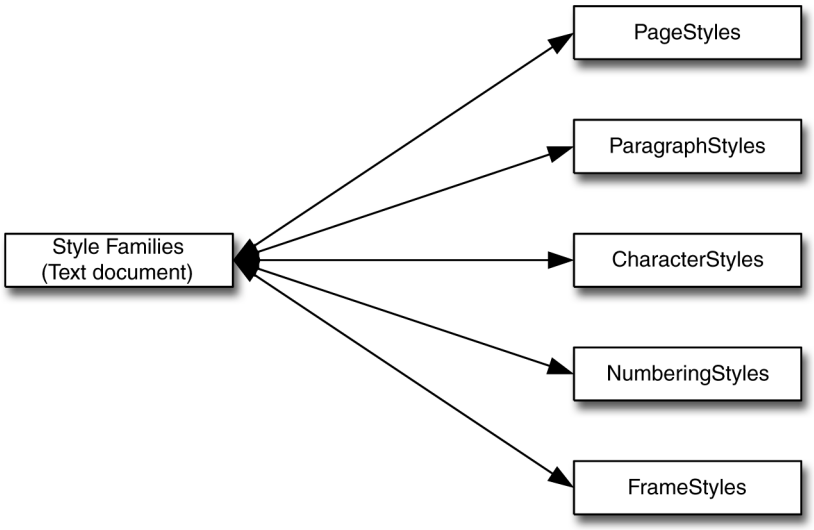

Figure 4–1 Style families of a TextDocument

To get the names of all style families available in a document, you can use this code:

```
Dim mFamilyNames As Variant

sMsg=""
oStyleFamilies = oDocument.StyleFamilies
mFamilyNames = oStyleFamilies.getElementNames()
For n = LBound(mFamilyNames) To UBound(mFamilyNames)
   sMsg=sMsg +  mFamilyNames(n) + " "
Next n
MsgBox sMsg,0,"StyleFamilies"
```

To summarize: If you want to be sure that a certain style exists, you must

1. check if the desired style *family* exists
2. find out if it contains the style you need.

Let's get back to the example above. Suppose that you don't know the name of the style used for headings and that you can't determine it from the available paragraph styles. You can then either try to find a suitable style or define your own one.

4.1.2 Finding a suitable style

If you don't know the name of the paragraph style you want to apply, you might want to search through the available styles to find one that suits your needs best. Suppose you wanted to use the style providing the largest bold Helvetica font for your headline. You could then try to find one like this:

```
Dim nFontSize As Long
Dim sFoundStyle As String
Dim sFontName As String

oStyleFamilies = oDocument.StyleFamilies
oParagraphStyles = oStyleFamilies.getByName("ParagraphStyles")
nFontSize = -1
For n = 0 To oParagraphStyles.Count - 1
  oStyle = oParagraphStyles(n)
  sFontName=lCase(oStyle.charFontName)
  If ( sFontName = "helvetica" And _
      oStyle.charHeight > nFontSize) Then
    nFontSize = oStyle.charHeight
    sFoundStyle = oStyle.Name
    oCursor.paraStyle = sFoundStyle
    Exit For
  End If
Next n
```

You should be aware that this example might not produce exactly what you want. It will certainly find the paragraph style providing the tallest bold Helvetica setting, if such a style exists at all. However, if all paragraph styles use a different font (e.g. Arial), it will use the style `heading` for the first paragraph. This approach certainly fails if `heading` is not defined. Furthermore, although the style found by this program provides the tallest bold Helvetica setting, but it might have other attributes that you don't like. For example, the alignment could be defined as centered or right, while you'd rather have a left aligned heading. Defining your own style overcomes these shortcomings.

4.1.3 Defining your own style

To be absolutely sure that a style has all the properties you want it to have, your best bet is to define it yourself. We will use the example from Section 4.1.1 "Style basics" on page 32 again, this time defining the style `myheading` and applying it to the first paragraph. The following code fragment shows the necessary steps to specify your own style.

```
oStyleFamilies = oDocument.StyleFamilies
oParagraphStyles = oStyleFamilies.getByName("ParagraphStyles")
oStyle = _
 oDocument.createInstance("com.sun.star.style.ParagraphStyle")
oParagraphStyles.insertByName("myheading",oStyle)
oStyle.Name = "myheading"
oStyle.CharFontName = "Helvetica"
oStyle.CharHeight = 36
oStyle.CharWeight = com.sun.star.awt.FontWeight.BOLD
oStyle.CharAutoKerning = TRUE
oStyle.ParaAdjust = _
 com.sun.star.style.ParagraphAdjust.LEFT
oStyle.ParaFirstLineIndent = 0
oStyle.breakType = _
 com.sun.star.style.BreakType.PAGE_BEFORE
oCursor.gotoStart(FALSE)
oCursor.gotoEndOfParagraph(TRUE)
oCursor.paraStyle = "myheading"
```

First of all, this example gets the named collection of all paragraph styles. It then creates a new paragraph style using `createInstance()` and adds it as new element to the collection with `insertByName()`. Note that this must be done *before* you can set the properties for the new style. Those properties are then defined so that a 36 point bold Helvetica is used. We set `CharAutoKerning` because kerning should always be used at this point size. The paragraph will be left-adjusted, and we don't want any indentation. The `breakType` property specifies if a page or column break occurs before this paragraph, after it, both before and after, or not at all. It is set to "before" here, because we want each `myheading` heading to start on a new page.

This last point is fairly important. If you have ever worked with StarWriter, you may have inserted a page break manually via Insert/Manual Break. The same dialog offers to insert a line break, a column break or a page break. While a line break is represented by a control character (see Section 4.3.4 "Inserting paragraph breaks, special characters, and page breaks" on page 53), column and page breaks are actually properties of a paragraph. They can be set for the paragraph style or directly for the paragraph itself. In the first case they apply to all paragraphs with this style, in the latter case they influence only the current paragraph.

When all properties for the new heading are set, it is assigned to the paragraph as in the previous section.

4.1.4 Hard formatting

What you've seen so far is called "soft" formatting: You define a style and apply it to all objects that you want to appear in that style. They are displayed using all properties of this style. If you change one style property, all objects change their appearance as well. "Hard" formatting is what many people are used to in a word processor. They select a single word and click on the *B* icon to make it appear bold. This formatting is called "hard" because it will always override the style defined for

this paragraph. In other words: Hard formatting means to set a certain property directly without changing the object's style.

Hard formatting is generally considered bad practice because it makes changes very difficult. Suppose you had a large document with captions of figures like this: `Figure 1.1: Descriptive Text`. In order to make the `Figure 1.1:` part stand out, you have it hard formatted as bold. The first time your boss sees the document, she tells you to change all these captions to use normal weight for the whole text - who'll have to walk through your document manually to modify every single caption. Had you soft formatted them, using a character format for the first part of the caption, you could have simply changed that.

Having said that, we will show you how to hard format text in StarOffice API. Since many documents contain hard formats, you should know how to create and change them. We assume the document has been created as described before and contains only the headline and the single paragraph .

`A very short paragraph for illustration only:`

```
oCursor.gotoStart(FALSE)
oCursor.gotoNextParagraph(FALSE)
oCursor.gotoNextWord(FALSE)
oCursor.gotoEndOfWord(TRUE)
oCursor.charWeight = com.sun.star.awt.FontWeight.BOLD
```

The preceding piece of code changes the word `very` to bold. As you can see, most of the code is needed to position the cursor so that it addresses the desired word. Changing it to bold is just a simple assignment.

As you have seen above, you can format a paragraph in a certain style by assigning the style's name to the `paraStyle` property of the paragraph. Of course, you can also inquire a paragraph's style. However, the result of this inquiry might require some interpretation if the paragraph contains hard formatting. If you change the previous code so that it looks like this:

```
oCursor.gotoStart(FALSE)
oCursor.gotoNextParagraph(FALSE)
oCursor.gotoEndOfParagraph(TRUE)
msgbox "Style: " + oCursor.paraStyle _
   + Chr(13) + "Font: " + oCursor.charFontName _
   + Chr(13) + "Weight: " + oCursor.charWeight
oCursor.gotoStartOfParagraph(FALSE)
oCursor.gotoNextWord(FALSE)
oCursor.gotoEndOfWord(TRUE)
oCursor.charWeight = com.sun.star.awt.FontWeight.BOLD
oCursor.gotoStartOfParagraph(FALSE)
oCursor.gotoEndOfParagraph(TRUE)
msgbox "Style: " + oCursor.paraStyle _
   + Chr(13) + "Font: " + oCursor.charFontName _
   + Chr(13) + "Weight: " + oCursor.charWeight _
   + Chr(13) + "Weight(Default): " + oCursor.getPropertyDefault("CharWeight")
```

Both message boxes will display the style name, the font name and a weight of usually 100, which is the value of the constant

com.sun.star.awt.FontWeight.NORMAL. This is in fact the default setting for this paragraph style. Changing very to bold doesn't directly affect the default weight of the paragraph. But since this attribute is no longer the same for every word of the paragraph, its propertyState is changed. Before the modification of very, it was DEFAULT_VALUE, afterwards it is AMBIGUOUS_VALUE. You can inquire the propertyState like this

```
oCursor.getPropertyState("CharWeight")
```

This method returns com.sun.star.beans.PropertyState.DEFAULT_VALUE for the default, com.sun.star.beans.PropertyState.AMBIGUOUS_VALUE for ambiguous and com.sun.star.beans.PropertyState.DIRECT_VALUE for a hard format, representing the constants. See the StarOffice reference documentation regarding the interface XPropertyState in com.sun.star.beans.

You can remove hard formatting in a text range with the method setPropertyToDefault(). For example, to make sure that a text range uses the style's font, size and weight, you could use these lines:

```
oCursor.setPropertyToDefault("CharWeight")
oCursor.setPropertyToDefault("CharFontName")
oCursor.setPropertyToDefault("CharHeight")
```

This code does not reset any italic portions to upright (or vice versa), you'd need to reset the property CharPosture for this.

Note - Please note that the strings passed to setPropertyToDefault() are case sensitive, charFontName would not work in the sample code above.

4.1.5 Headers, footers and tab stops

Headers and footers are part of the page styles. They are mostly used in longer documents to display the page number and other information like the title of the current chapter. Before you can define headers and footers, you have to turn them on like this

```
Dim oPageStyles As Object
Dim oStdPage As Object

oPageStyles = oStyleFamilies.getByName("PageStyles")
oStdPage = oPageStyles.getByName("Standard")
oStdPage.HeaderOn = TRUE
oStdPage.FooterOn = TRUE
```

Here we enable headers and footers for the Standard page style. To have the header display some text, you use something like this:

```
Dim oHeader As Object

oHeader = oStdPage.HeaderText
```

```
oHeader.String = "My Header"
```

This is obviously the simplest thing to do; it will display the text "My Header" in the header on every page. If you want some fancy formatting, you have to modify the style used for the header as in the next lines of code:

```
Dim oHeaderText As Object
Dim oHeaderCursor As Object

oHeaderText = oHeader.Text
oHeaderCursor = oHeaderText.createTextCursor()
oParagraphStyles = oStyleFamilies.getByName("ParagraphStyles")
oStyle = oParagraphStyles.getByName(oHeaderCursor.paraStyle)
oStyle.CharPosture = com.sun.star.awt.FontSlant.ITALIC
```

As you can see, it is easy to get at the style used for the header. Its name is found in the `paraStyle` property of the `XTextCursor` interface for the header. Once you know the name of the style (which depends on the local language settings), you can change its properties as shown before. The sample above changes the `CharPosture` property so that the header text appears in italics.

While a static header is appropriate to show the title of a document, there a more dynamic data that might you want to appear in the footer. One of them is obviously the page number. To have it shown in the footer, you can write something like the following:

```
Dim oFooter
Dim oFooterText As Object
Dim oFooterCursor As Object
Dim oPageNumber As Object

oFooter = oStdPage.FooterText
oFooterText = oFooter.Text
oFooterCursor = oFooterText.createTextCursor()
oFooterText.insertString(oFooterCursor,"Page ", FALSE)
oPageNumber = _
 oDocument.createInstance("com.sun.star.text.TextField.PageNumber")
oPageNumber.NumberingType = _
  com.sun.star.style.NumberingType.ARABIC
oFooterText.insertTextContent(oFooterCursor, oPageNumber, FALSE)
```

The first few lines are similar to the previous examples showing the usage of headers. We then insert the string `Page` in the footer and create the text field `pageNumber` which is inserted right afterwards. Its `NumberingType` property is set so that the page numbers are displayed using arabic digits. To learn more about text fields, refer to Section 4.3.7 “Inserting tables, frames, etc.” on page 58.

If you have more than one page in your document, you will notice that the page number inserted with the code above appears always at the left margin. Typically, one would want the even numbers to appear at the left margin and the odd numbers at the right margin. To accomplish this, you have to use a left and a right footer like this:

```
Dim oFooterLeft As Object
Dim oFooterRight As Object
Dim oFooterTextLeft As Object
Dim oFooterTextRight As Object
Dim oFooterCursorLeft As Object
Dim oFooterCursorRight As Object

oStdPage.FooterShareContent=TRUE
oFooterLeft = oStdPage.FooterTextLeft
oFooterTextLeft = oFooterLeft.Text
oFooterCursorLeft = oFooterTextLeft.createTextCursor()
oFooterTextLeft.insertString(oFooterCursorLeft,"Page ", FALSE)
oPageNumber = _
 oDocument.createInstance("com.sun.star.text.TextField.PageNumber")
oPageNumber.NumberingType = com.sun.star.style.NumberingType.ARABIC
oFooterTextLeft.insertTextContent(oFooterCursorLeft, oPageNumber, FALSE)
oFooterRight = oStdPage.FooterTextRight
oFooterTextRight = oFooterRight.Text
oFooterCursorRight = oFooterTextRight.createTextCursor()
oFooterTextRight.insertString(oFooterCursorRight,"Page ", FALSE)
oPageNumber = _
 oDocument.createInstance("com.sun.star.text.TextField.PageNumber")
oPageNumber.NumberingType = com.sun.star.style.NumberingType.ARABIC
oFooterTextRight.insertTextContent(oFooterCursorRight, oPageNumber, FALSE)
```

This does not introduce anything really new. Instead of using the `FooterText` property of the `Standard` page style, you use `FooterTextLeft` and `FooterTextRight`. If you run this code, you'll notice that the page numbers appear twice on each page. To have the `FooterTextRight` appear only on right (odd) pages, you would have to set the page style's `FooterShareContent` property to `FALSE`.

With this value, the page number appears only once on every page. However, it is flush left on even and odd pages, and one would rather want it to appear at the right margin on an odd page. To put the odd page numbers at the right margin, you must use a tab stop:

```
Dim newstops (0) As Object
Dim tabStop As New com.sun.star.style.TabStop
Dim oFooterStyle As Object
Dim h As Long

oFooterStyle = _
 oParagraphStyles.getByName(oFooterCursorRight.paraStyle)
h = oStdPage.Size.Width - oStdPage.LeftMargin - _
    oStdPage.RightMargin
tabStop.position = h
tabStop.alignment = com.sun.star.style.TabAlign.RIGHT
newstops(0) = tabStop
oFooterStyle.paraTabStops = newstops()
```

Tab stops belong to a paragraph style, so we have to get the style used for the right footer first. We then calculate the width of the footer line by subtracting the page's left and right margin from its width and store this value in `h`. The position of the tab stop is set to this value and right alignment is specified for it. The footer style's `paraTabStops` property is then overwritten with an array which contains only the

newly defined tab stop. Any tab stops defined for this paragraph style before will no longer exist afterwards, so make sure you understand the implications of changing the paragraph attributes.

Of course, defining the tab stop is only part of the solution, you have to *jump* to it as well. The code to insert the page number for odd pages looks like this:

```
oFooterTextRight.insertString(oFooterCursorRight,_
 Chr(9)+"Page ", FALSE)
oPageNumber = _
 oDocument.createInstance("com.sun.star.text.TextField.PageNumber")
oPageNumber.NumberingType = _
 com.sun.star.style.NumberingType.ARABIC
oFooterTextRight.insertTextContent(oFooterCursorRight,_
  oPageNumber, FALSE)
```

Note the usage of `Chr(9)` in the call to `insertString()` to insert a literal tab character.

Finally, we will show you how to include the title of the current chapter in the footer of each page. First of all, you have to create a text field that contains the chapter's title:

```
Dim oChapterField As Object

oChapterField = _
 oDocument.createInstance("com.sun.star.text.TextField.Chapter")
oChapterField.Level = 0
oChapterField.chapterFormat = 1
```

The preceding lines create a text field which uses the heading for numbering level 0 (the top level) and includes the heading's text (this is achieved by setting the `chapterFormat` property). StarOffice API has certain default settings associating paragraph styles with numbering levels. To be sure that it uses the style you want, you have to set it like this:

```
Dim oChapterSettings As Object
Dim mLevel As Variant
Dim vProperty As Variant

oChapterSettings = oDocument.ChapterNumberingRules
mLevel = oChapterSettings.getByIndex(0)
For n = LBound(mLevel) To UBound(mLevel)
  vProperty = mLevel(n)
  If (vProperty.Name = "HeadingStyleName") Then
    vProperty.Value = "Heading 1"
  End If
  mLevel(n) = vProperty
Next n
oChapterSettings.replaceByIndex(0,mLevel)
```

This piece of code retrieves the current settings for numbering level 0, which is the one used in the text field above. It then loops over all properties until it finds the one called `HeadingStyleName`. Its value is then set to `Heading 1` and the chapter

settings are updated. The last step is to insert the text field containing the chapter name into the footer:

```
oFooterTextRight.insertTextContent(oFooterCursorRight, _
  oChapterField, FALSE)
oFooterTextRight.insertString(oFooterCursorRight,_
  Chr(9) + "Page ", FALSE)
oPageNumber = _
  oDocument.createInstance("com.sun.star.text.TextField.PageNumber")
oPageNumber.NumberingType = com.sun.star.style.NumberingType.ARABIC
oFooterTextRight.insertTextContent(oFooterCursorRight,_
  oPageNumber, FALSE)
```

This code inserts the text field with the chapter name at the left bottom and the page number after a tab character at the right margin. This is ok for a right (odd) page. You'd have to insert the fields for chapter and page the other way round for the left (even) footer.

4.2 Importing, Exporting and Printing

Regardless of the type of document you are working with, you will occasionally want to import files in other formats, you'll most definitely want to save your work and you will want to print documents. The following sections explain how to achieve each of these goals.

4.2.1 Importing other Formats

Although you'll be working with StarOffice files most of the time, you will occasionally come across a file generated by another product. StarOffice provides many filters for foreign formats permitting you to work with these files, and you can use them from StarOffice API as well. Generally the `ComponentLoader` will select the correct filter automatically, based on the extension of the file. In some cases, however, you might want to force the use of a particular format. This might be necessary if the contents of a file does not match the defaults of StarOffice, for example if `csv.doc` contains comma separated values instead of a Word document as the extension `doc` suggests.

```
Sub import1_sample
  Dim mFileProperties(0) As New com.sun.star.beans.PropertyValue
  Dim sUrl As String

  oDesktop  = createUnoService("com.sun.star.frame.Desktop")
  sUrl = "file:///home/testuser/Office52/work/csv.doc"
  mFileProperties(0).Name = "FilterName"
  mFileProperties(0).Value =_
    "scalc: Text - txt - csv (StarCalc)"
  oDocument = oDesktop.loadComponentFromURL(sUrl,"_blank",_
```

```
   0,mFileProperties())
End Sub
```

As before, we use the `Desktop()` service to open the document. The important part here is the property set `mFileProperties`. It contains only the property "FilterName" with the value `<factory>: <filtername>`. Here, `<factory>` is one of `swriter, scalc, simpress, sdraw, smath, simage`. The `<filtername>` can be found in the file `install.ini`. In the future, you will be able to retrieve it from the `Registry()` service of StarOffice API.

To use a document that contains a foreign file format, you may have to provide the `FilterFlags` property to `loadComponentFromUrl()` to ensure that the document is imported in a certain way. A CSV file, for example, can use commas or semicolons to separate fields or it may use fields of fixed width. `FilterFlags` is needed to specify these details. The value of this property is a single string that contains all required information. For a CSV file, it is made up of five tokens, separated by comma. The tokens are

1. Field separator(s) as ASCII values or the three letters `FIX`. If your fields are separated by commas, this token is `44`, if they are separated by semicolons and tabs, the token would be `59/9`. To treat several consecutive separators as one, append the four letters `/MRG` to this token. If the file contains fixed width fields, use the three letters `FIX` as the token.
2. The text delimiter as ASCII value. Use 34 for double quotes and 39 for single quotes.
3. The character set in the file as string.
4. Number of the first line to convert. Set this to something other then 1 if you want to skip lines at the beginning of the file.
5. This last token is the most complicated one. Its content depends on the value of the first token.

   - If it is `FIX`, this token is of the form `start/format/start/format`. `Start` is the number of the first character for this field, with 0 being the leftmost character in a line. `Format` is explained below.
   - If you are not using fixed fields but separators, the form of the last token is `no/format/no/format` where `no` is the number of the column, 1 being the leftmost column.

   `Format` specifies the content of a field. It is the number 1 for standard, 2 for text, 3, 4, and 5 for date values (MM/DD/YY, DD/MM/YY, YY/MM/DD respectively), 6, 7, and 8 for text and 9 for fields to ignore. 10 is used to indicate that the content of this field is US-English. This is particularly useful if you know that a field contains decimal numbers which are formatted according to the US system (using "." as decimal separator and "," as thousands separator). Using 10 as format specifyer for this field tells StarOffice API to correctly interpret its numerical content even if the decimal and thousands separator in your country are different.

Consider a file that contains four columns: the first one contains text delimited by double quotes, the other columns contain numbers. The columns are separated by commas. The `FilterFlags` for this file would be `44,34,SYSTEM,1,1/1/1/1/1/1/1/1/1`. A complete piece of code looks like this:

```
Sub import2_sample
  Dim mFileProperties(1) As New com.sun.star.beans.PropertyValue
  Dim sUrl As String

  sUrl = "file:///home/ck/ix/ix0399/javaperf/jview.csv"
  mFileProperties(0).Name = "FilterName"
  mFileProperties(0).Value = "scalc: Text - txt - csv (StarCalc)"
  mFileProperties(1).Name = "FilterFlags"
  mFileProperties(1).Value = "44,34,SYSTEM,1,1/1/1/1/1/1/1/1/1"
  oDesktop = createUNOService("com.sun.star.frame.Desktop")
  oDocument = oDesktop.loadComponentFromURL(sUrl,_
    "_blank",0,mFileProperties())
End Sub
```

4.2.2 Saving and exporting documents

If you have changed a document, you will probably want to save these changes. The service `OfficeDocument` provides the interface `XStorable` which contains all the necessary methods to store documents.

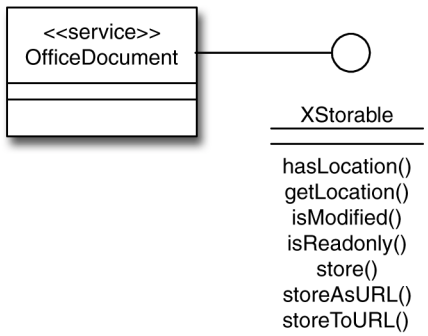

Figure 4–2 Storing documents

The simplest way to do this is to call the `store()` method:

```
oDocument.store()
```

saves the document under its original name. This works only if it existed before you worked on it or if you have already saved a new document. To save a new document for the first time, you'll use

```
Dim mFileProperties(0) As New com.sun.star.beans.PropertyValue
Dim sUrl As String

sUrl = "file:///complete/path/To/New/document"
mFileProperties(0).Name = "Overwrite"
mFileProperties(0).Value = FALSE
oDocument.storeAsURL(sUrl, mFileProperties())
```

This method will save the document in its native StarOffice format, but it will not overwrite a possibly existing file of the same name. To replace existing files, you'll have to set the `overwrite` parameter to `TRUE`. To save the document in a foreign format, you'll have to specify a `FilterName` property as shown above in Section 4.2.1 "Importing other Formats" on page 41.

To make a save-document subroutine more robust, you can use two other properties that help you decide what to do:

```
oDocument = ThisComponent
If (oDocument.isModified) Then
  If (oDocument.hasLocation And (Not oDocument.isReadOnly)) Then
    oDocument.store()
  Else
    oDocument.storeAsURL(sURL, mFileProperties())
  End If
End If
```

This sample code checks first if the current document has been modified - there is no need to save anything if it has not been changed. The sample chooses between `store()` and `storeAsURL()` depending on two criteria: In order to use `store()`, the document must have been saved before and it must be writeable. Only if both conditions are met, `store()` can be used to save the document. In all other cases, a new file is created. You'll have to specify the correct values for the parameters `URL` and `mFileProperties` before you can use this code.

4.2.3 Printing

Printing is not related to one particular type of document but it is still most important for text. The interface `xPrintable` is contained in the service `OfficeDocument` since all office documents have to provide printing.

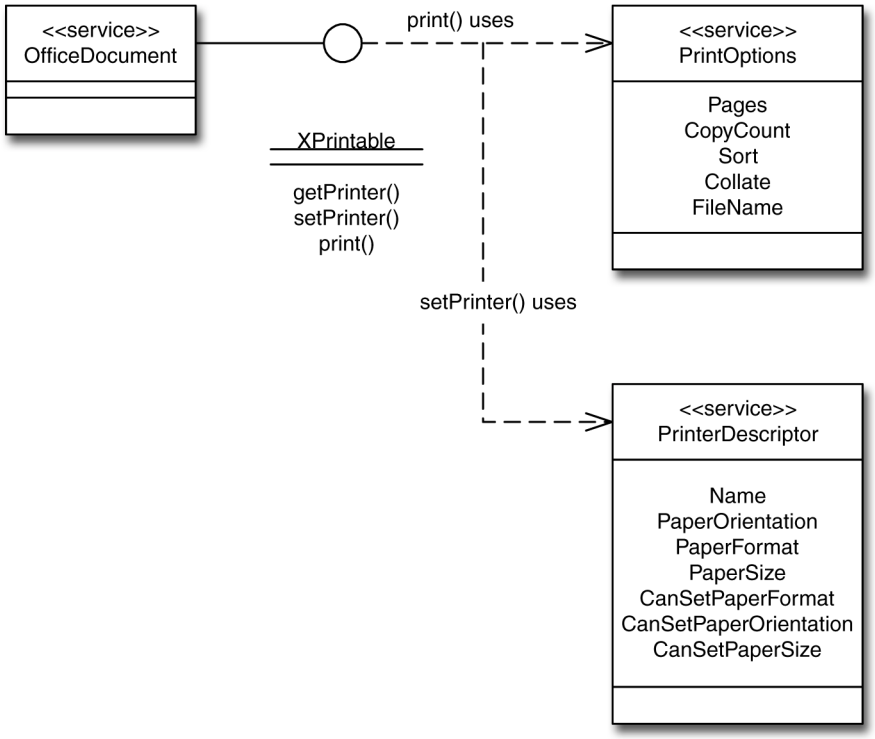

Figure 4–3 Printing documents

Printing a complete document to the default printer is as easy as

```
Dim mPrintopts1()

oDocument.Print(mPrintopts1())
```

Now suppose you wanted to print only the first two pages:

```
Dim mPrintopts2(0) As New com.sun.star.beans.PropertyValue

mPrintopts2(0).Name="Pages"
mPrintopts2(0).Value="1-2"
oDocument.Print(mPrintopts2())
```

As usual, you define a property set and define the property `Pages` in it. To print single pages, separate them with a semicolon:

```
mPrintopts2(0).Name="Pages"
mPrintopts2(0).Value="1-3; 7; 9"
```

prints pages 1, 3, 7, and 9. Other useful properties for printing are

- FileName; set this to the name of a new file if you want to print to a file instead of a printer.
- CopyCount; the number of copies you want to print.
- Sort; set to TRUE to print sorted copies. This is useful only if CopyCount is greater than 2. With Sort set to TRUE, the copies are printed in order.

These properties apply to the print job. There is another set of properties used to configure the printer itself.

To send the document to another printer, you'd have to choose this printer first:

```
Dim mPrinter(0) As New com.sun.star.beans.PropertyValue

mPrinter(0).Name="Name"
mPrinter(0).value="Other printer"
oDocument.Printer = mPrinter()
```

Then you can use print() as before or set print options first. Other useful printer options are

- CanSetPaperFormat; this boolean indicates if you can use the PaperFormat property to change the paper format.
- CanSetPaperOrientation; this boolean indicates if you can use the PaperOrientation property to change the paper format.
- IsBusy; this boolean informs you if the printer is busy.
- PaperFormat; a constant selecting the paper format, use com.sun.star.view.PaperFormat.USER for a user defined format.
- PaperOrientation; an integer selecting the paper orientation, com.sun.star.view.PaperOrientation.PORTRAIT or com.sun.star.view.PaperOrientation.LANDSCAPE
- PaperSize; if a user defined format has been set with the PaperFormat property, specify the desired size here in 100ths of a millimeter. The size is a structure of type com.sun.star.awt.Size with the components Width and Height

The first three properties in this list are read-only - you can not set them for obvious reasons.

4.3 Text

In this section, you'll learn about some basic operations for text: Inserting and retrieving it, navigating through it, searching and replacing text, importing foreign formats and printing text documents.

Text can never exist outside of a container, for example a text document or a spread sheet. You must therefore open an existing document first or create a new one like this:

```
Global oDesktop As Object
Global oDocument As Object
Global oText As Object

Sub textdoc_init
  Dim mNoArgs() REM Empty Sequence
  Dim sUrl As String

  oDesktop  = createUnoService("com.sun.star.frame.Desktop")
  sUrl = "private:factory/swriter"
  REM Or: sUrl = "file:///home/testuser/office52/work/text.sdw"
  oDocument = oDesktop.LoadComponentFromURL(sUrl,"_blank",0,mNoArgs)
  oText = oDocument.Text
End Sub
```

Note that calling `LoadComponentFromURL()` with this `sURL` creates a new document. If you were using the `sURL` which is commented out here, the document `/home/testuser/office52/work/text.sdw` would be opened instead. The variable `oDocument` now provides all interfaces needed for text manipulation. In the following sample code, we'll leave out the part above above and assume that `oDocument` has been created as described.

4.3.1 The structure of text documents

The basic structure of text documents is an enumeration of paragraphs and tables. If you want to walk through the whole document, you should use this enumeration to do so. Furthermore, a text document can contain frames, graphics, fields and other objects providing a `XTextContent` interface. They are always anchored to a part of the document (a page, a paragraph or a character). Because objects are named, you can access them using `getByName()` on the appropriate collection.

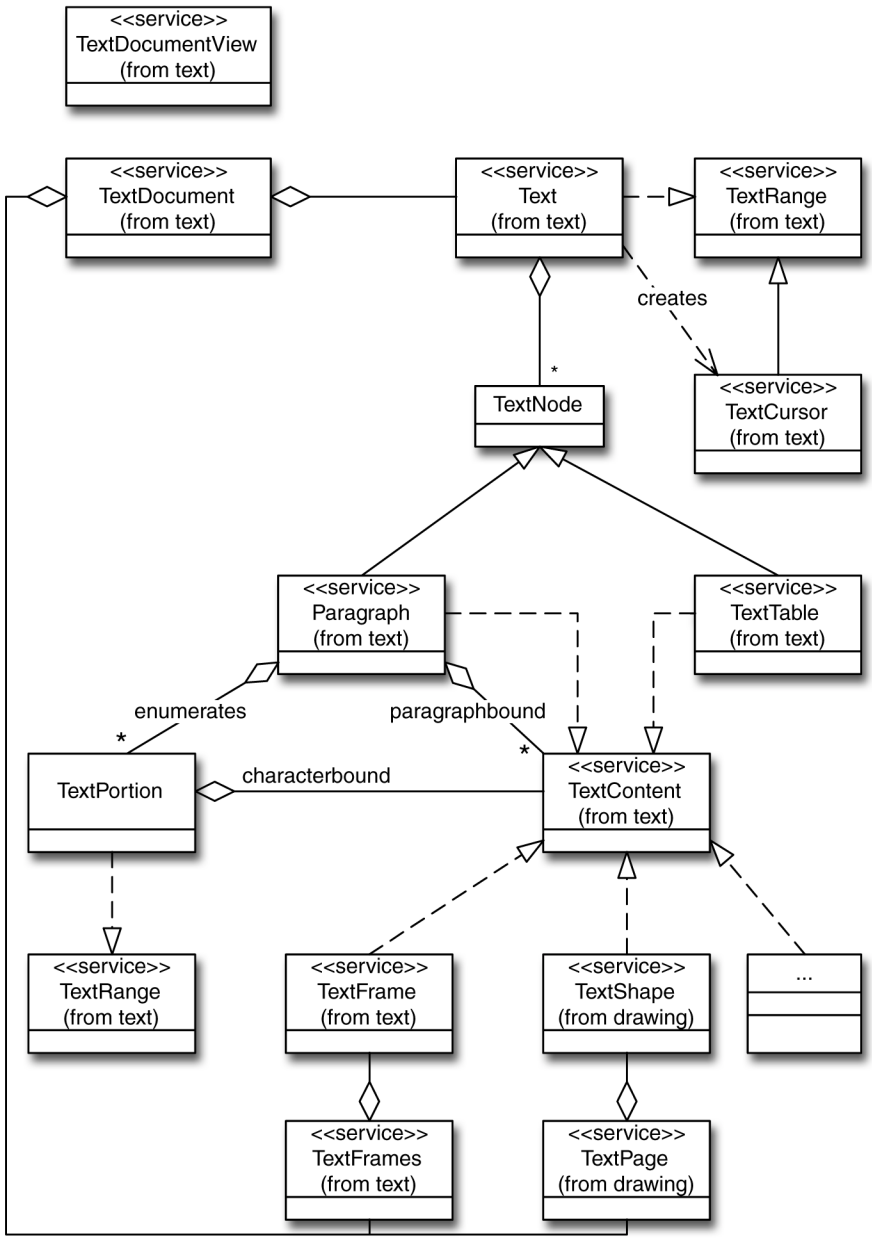

Figure 4–4 Structure of a TextDocument

The paragraphs of a document are neither named nor indexed so that you must access them sequentially starting with the first paragraph. Although tables are available in their own named collection they are always anchored to the preceding

paragraph and belong to the normal flow of text. That's the reason why they are enumerated with the paragraphs.

```
Dim oTextEnum As Object, oTextElement As Object

oTextEnum = oDocument.Text.createEnumeration
While oTextEnum.hasMoreElements()
   oTextElement = oTextEnum.nextElement()
   REM  Do something with the paragraph Or table
Wend
```

This piece of code will move you through *all* paragraphs and tables in the order in which they appear in the document. It creates an enumeration of the objects in the `XText()` interface with `createEnumeration()`. Then it uses `hasMoreElements()` and `nextElement()` to iterate through this numeration. `nextElement()` does two things at the same time: it returns the next element of the enumeration and it moves beyond it. To skip tables, check each element for the `Paragraph()` service like this:

```
If oTextElement.supportsService(_
 "com.sun.star.text.Paragraph") Then
   REM  Code that works With paragraph only
End If
```

Now what is `oTextElement` really? Although we were talking about paragraphs before, it is not always what you perceive as a document's paragraph. Rather, `oTextElement` is another enumeration where each element has the same style attributes. More strictly, `oTextElement` is a `Paragraph()` service. You might think of it as a collection of text portions where one portion ends when a text attribute like font or color changes. You can walk through them like this:

```
Dim oTextPortionEnum As Object
Dim oTextPortion As Object

oTextPortionEnum = oTextElement.createEnumeration()
While oTextPortionEnum.hasMoreElements()
  oTextPortion = oTextPortionEnum.nextElement()
  REM Do something With the Text portion
Wend
```

Some examples:

- If the text of a document's paragraph is formatted with the same attributes (font, font weight, color etc.), the enumeration contains just one text portion which is the complete paragraph.
- If you have one bold word in the middle of a document's paragraph, the enumeration will contain three text portions: First the text up to the bold word, then the bold word itself, and finally the rest of the paragraph.

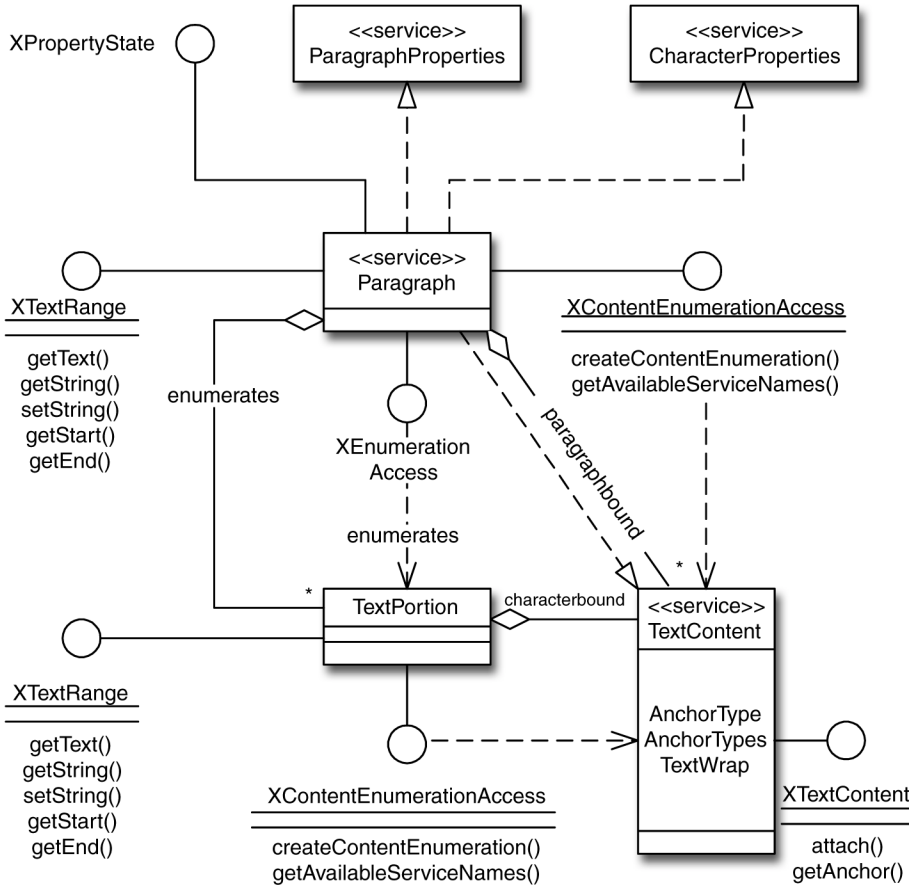

Figure 4–5 Paragraph

You have to be aware of these enumerations in a paragraph if you want to inquire or change its properties. For example: If you are dealing with a single text portion it is sufficient to change the `charFontName` property of `oTextElement` to change the font for the whole paragraph. If you have more than one text portion, you'll have to walk through the paragraph and check the `charFontName` property for each of them:

```
oTextPortionEnum = oTextElement.createEnumeration
While oTextPortionEnum.hasMoreElements
   oTextPortion = oTextPortionEnum.nextElement
   Print oTextPortion.CharFontName
Wend
```

The font name for `oTextElement` is the one of the style used for this paragraph. Each text portion may or may not override this default by specifying its own font name. The actual font name for a portion is stored in its `charFontName` property.

You can use the getPropertyState() method to find out if the font name of a text portion is the one inherited from the paragraph style or if it overrides the paragraph style ones:

```
Dim n As Integer

n = oTextPortion.getPropertyState("CharFontName")
```

n will be com.sun.star.beans.PropertyState.DEFAULT_VALUE which is equal to 1, if the font name has not changed from the style's font name, com.sun.star.beans.PropertyState.DIRECT_VALUE (=2) if it has been overriden by a hard format. A value of com.sun.star.beans.PropertyState.AMBIGUOUS_VALUE (=0) for n indicates that the text portion contains more than one font setting. In this case, the font name is the one that has been set by the portion's style. For more details, you can read the StarOffice API Reference Manual on com.sun.star.beans.XPropertyState.

4.3.2 Moving around

Enumerations are fine if you want to walk through your document from start to end. To move around deliberately, you need a TextCursor() service. Dont't be confused that the cursor on the screen does not move. A TextCursor() acts behind the scenes. You create a TextCursor() like this:

```
Dim oCursor As Object

oText = oDocument.Text
oCursor = oText.createTextCursor()
```

Now you can change this cursor's position in the text:

```
oCursor.goLeft(3,FALSE)
```

moves it three characters to the left,

```
oCursor.goRight(10,FALSE)
```

positions it ten characters to the right etc.

Besides the simple cursor movement by character we've just shown, there are other more powerful interfaces: A page cursor moves between pages, a paragraph cursor between paragraphs, a word cursor jumps from word to word, and a sentence cursor moves between sentences. gotoNextSentence() and gotoPreviousSentence() position the cursor on the next and previous sentence in the document, respectively. Similarly, gotoNextParagraph() and gotoPreviousParagraph() go to the next and previous paragraph; gotoEndOfParagraph() and gotoStartOfParagraph() move to the beginning and end of the current paragraph.

A cursor is a means to move to a certain position in a document. However, it's never *only* a single position, although it may sometimes look like this. A cursor is always a *range* of text, although the beginning and the end of this range may coincide. In this case the range has zero length and is positioned between two characters. We will show you what you can do with ranges in the next section.

4.3.3 Inserting and Changing text

Three basic steps are required to insert new text into a document:

1. create a cursor
2. select the position where you want to put the new text
3. insert the text.

The following examples will explain these steps in more detail.

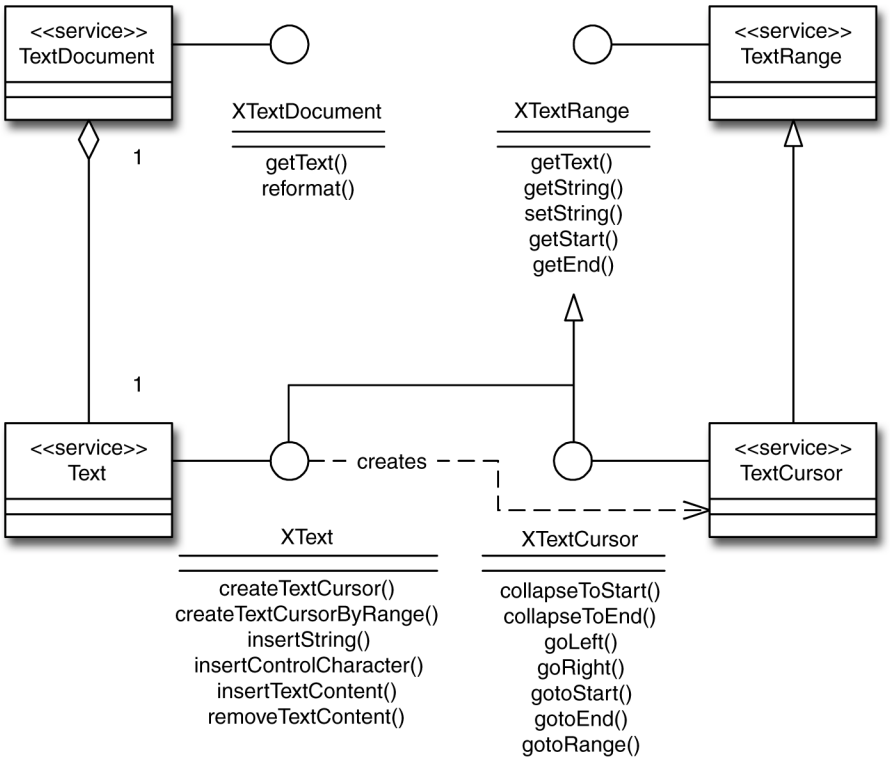

Figure 4–6 Inserting and changing text

You might be wondering what the FALSE parameter is doing in the cursor function calls above - why can't you simply say cursor.goRight(10)() to move ten characters forward? The reason for this is simple. Since moving in a document is almost always done with the purpose to *do* something with the text, StarOffice API cursors are prepared to work. As long as you set the last parameter of the movement functions to FALSE, you can simply insert text at the current cursor position with

```
oText.insertString(oCursor,_
 "New Text at current oCursor position.",_
 FALSE)
```

In these cases, the cursor position is the text range specified by a single character.

You can move around in the document and change the existing text by replacing it. To do so, you must eventually set the last parameter of the movement functions to TRUE, like in the following example.

```
oText = oDocument.Text
oCursor = oText.createTextCursor()
oCursor.gotoStart(FALSE)
oText.insertString(oCursor,"A piece of New Text And another one.",_
 FALSE)
oCursor.gotoStart(FALSE)
oCursor.gotoNextWord(FALSE)
oCursor.gotoEndOfWord(TRUE)
oText.insertString(oCursor,"chunk",TRUE)
```

The current position is first set to the beginning of the document and text is inserted. The cursor is then moved to the beginning of the next word. Now we "span" the textRange with oCursor.gotoEndOfWord(TRUE)(), so that it extends from the last position (the first character of "piece") to the current one (the last character of "piece"). Finally, this span of text is replaced with the word "chunk". insertString() replaces the text span, because its last parameter is TRUE. If you wanted to insert text after the end of the spanned region, you'd set this value to FALSE.

A region spanned by a cursor survives a call to insertString(). To switch back to normal cursor movement (i.e. withouth spanning a region), you have to use collapseToEnd() or collapseToStart(). The first function sets the current cursor position to the end of the range, the second function sets it to the start.

4.3.4 Inserting paragraph breaks, special characters, and page breaks

Documents hardly ever consist of one long piece of text but of several paragraphs. You create a paragraph by inserting a "control character" after the text that makes up the paragraph.

```
oText.insertControlCharacter(oCursor,_
com.sun.star.text.ControlCharacter.PARAGRAPH_BREAK,FALSE)
```

would start a new paragraph at the end of the current `textRange`. Again, you must move the range *beyond* the newly inserted text with `c.collapseToEnd()`, if you want to insert text into the new paragraph.

Similarly, to insert a line break, you'd use

```
oText.insertControlCharacter(oCursor,_
 com.sun.star.text.ControlCharacter.LINE_BREAK,FALSE)
```

The following control characters are defined:

`PARAGRAPH_BREAK`: A new paragraph

`LINE_BREAK`: Start a new line in the current paragraph.

`HARD_HYPHEN`: A fixed hyphenation character that hyphenation cannot remove.

`SOFT_HYPHEN`: Indicating a possible hyphenation point, visible in the document only if actual hyphenation occurs at this point.

`HARD_SPACE`: A fixed space that can neither be stretched nor compressed by justification.

The control characters are actually constants that can be inserted in your program code by prepending `com.sun.star.text.ControlCharacter.` to the name of the character to insert, as shown above in the short examples. The following code segment illustrates the use of all control characters.

```
oCursor = oText.createTextCursor()
oCursor.gotoStart(FALSE)
oText.insertString(oCursor,"My first piece of Text",FALSE)
oText.insertControlCharacter(oCursor,_
  com.sun.star.text.ControlCharacter.PARAGRAPH_BREAK,FALSE)
oText.insertString(oCursor,_
 "second Line In first paragraph", FALSE)
oText.insertControlCharacter(oCursor,_
 com.sun.star.text.ControlCharacter.LINE_BREAK,FALSE)
oText.insertString(oCursor,"Second paragraph", FALSE)
oText.insertControlCharacter(oCursor,_
 com.sun.star.text.ControlCharacter.PARAGRAPH_BREAK,  FALSE)
oText.insertString(oCursor,_
 "First Line In 3rd paragraph,", FALSE)
oText.insertString(oCursor," a fixed hyphen", FALSE)
oText.insertControlCharacter(oCursor,_
 com.sun.star.text.ControlCharacter.HARD_HYPHEN,FALSE)
oText.insertString(oCursor,", a soft hyphen,", FALSE)
oText.insertControlCharacter(oCursor,_
 com.sun.star.text.ControlCharacter.SOFT_HYPHEN,FALSE)
oText.insertString(oCursor," And a fixed space", FALSE)
oText.insertControlCharacter(oCursor,_
 com.sun.star.text.ControlCharacter.HARD_SPACE,FALSE)
oText.insertString(oCursor,"between two words", FALSE)
oText.insertControlCharacter(oCursor, _
 com.sun.star.text.ControlCharacter.PARAGRAPH_BREAK,FALSE)
oText.insertString(oCursor,"New page starts here",FALSE)
oCursor.gotoStartOfParagraph(TRUE)
oCursor.breakType = com.sun.star.style.BreakType.PAGE_BEFORE
```

A page break is inserted at the end of this example. You may have noticed that there is no control character provided for page breaks. Text can be used in different contexts, for example in a paragraph or in the cell of a table. While the control characters mentioned so far have reasonable uses even in a cell, a page break is meaningful only in some circumstances (and not in a table cell). It can thus not be inserted with `insertControlCharacter()`.

There are two methods to force a page break in the text.

1. Set the `PageDescName` property of the paragraph to the name of the page style you want the next page to use. This sets the paragraph's `breakType` property to `PAGE_BEFORE` automatically. If you want to remove this page break, you have to set `PageDescName` to an empty string again.
2. Set the `breakType` property of the paragraph to `PAGE_AFTER`, `PAGE_BOTH`, `COLUMN_BEFORE`, `COLUMN_AFTER`, or `COLUMN_BOTH`. In these cases, you can't specify a page style to use for several reasons. First, if you specify a column break, a page format makes no sense. Second, if you use a page break that is *not* `PAGE_BEFORE`, the page break would have to occur after the paragraph for which you specify the page style to use with the *next* paragraph. This would make the paragraph style quite incomprehensible.

As in `insertString()`, the last parameter of `insertControlCharacter()` determines if the text in the current text range is replaced (`TRUE`) or not (`FALSE`).

4.3.5 Searching and replacing

We have shown you how you can replace text in a document by creating a spanning cursor. This approach is fine as long as you can move to the text to be replaced with the cursor. If you are working on a document that you didn't create yourself, this is generally not possible. In this case, you'll have to search for the correct position first. Search and replace descriptors are the objects you need to achieve this.

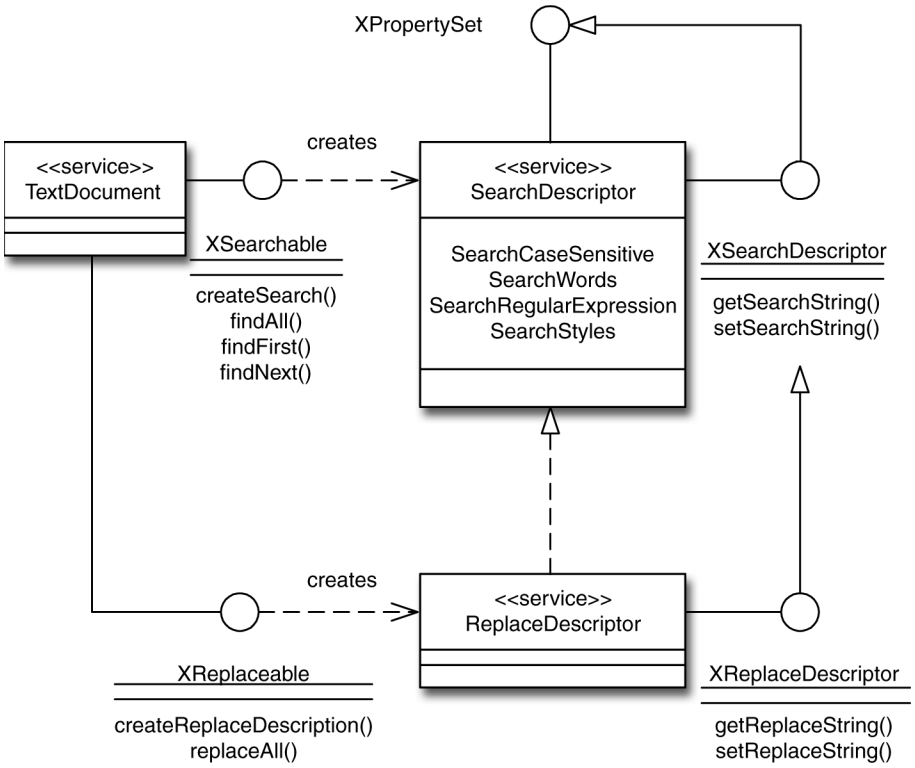

Figure 4–7 Searching and replacing

```
oSearchDesc = oDocument.createSearchDescriptor
oSearchDesc.searchString="find"
```

This piece of code creates a `SearchDescriptor` and then sets the string to search for to the word `find`. You have to start searching like this:

```
oFound = oDocument.findFirst( oSearchDesc )
Do While mFound
 REM  Do something With the Text
   oFound = oDocument.findNext( oFound.End, aSearch)
Loop
```

`findNext()` returns the next range of text that contains the search string. It starts at the text range specified in its first parameter. We use `oFound.End` here, which is the end of the last text found. `End` is a property of the `XTextRange()` interface. Since `findNext()` returns an `XTextRange()` interface, you can use all properties and methods available with it. That makes it easy to change attributes. To change all appearances of the word "Bauhaus" in a document so that it is displayed in the font with the same name, you'd use something like this:

```
oSearch = oDocument.createSearch
oSearch.searchString="Bauhaus"
oSearch.SearchWords = TRUE
oSearch.SearchCaseSensitive = TRUE
oFound = oDocument.findFirst( oSearch )
Do While NOT isNull(oFound)
   oFound.charFontName="Bauhaus"
   oFound = oDocument.findNext( oFound.End, oSearch)
Loop
```

Two additional properties of the search descriptor ensure that "Bauhaus" is changed only when it appears on its own and when the letter case match exactly (i.e. "bauhaus" would not be changed).

Instead of walking through the whole document with `findNext()`, you can use `findAll()` to get a sequence of text ranges where the search string occurs:

```
oFound = oDocument.findAll( oSearch )
For n = 0 To oFound.Count - 1
   REM  Do something With the oFound.getByIndex(n)
Next n
```

This approach is not fundamentally different from the first one.

4.3.6 Using regular expressions

To find strings in a certain context, you should use regular expressions. We will not explain them in detail here since the StarOffice documentation contains all necessary information. Suppose that in a bibliography each entry starts with the title followed by the author. You cannot search for only the lines in the document containing these entries with a simple string, since there is no common *word* for all entries. However, they share a common *structure*- each line starts with a number contained in brackets. Using a regular expression, you can search for this structure. The following program fragment exchanges title and author:

```
Dim oSearchDesc As Object, oCursor As Object
Dim oFoundAll As Object, oFound As Object
Dim nLen As Integer, n As Integer
Dim nEndAuthor As Integer
Dim nStartTitle As Integer, nEndTitle As Integer
Dim sAuthor As String, sTitle As String
Dim s As String, sRest As String

oSearchDesc = oDocument.createSearchDescriptor()
oSearchDesc.SearchString = "^\[[0-9]+\]"
oSearchDesc.SearchRegularExpression = TRUE
oFoundAll = oDocument.findAll( oSearchDesc )
For n = 0 To oFoundAll.Count - 1
   oFound = oFoundAll(n)
   REM Or: oFound=oFoundAll.getByIndex(n)
   oCursor = oText.createTextCursorByRange(oFound)
   oCursor.gotoNextWord(FALSE)
   oCursor.gotoEndOfParagraph(TRUE)
```

```
    s = oCursor.String
    nEndAuthor = InStr(1, s, ";") -1
    nLen = nEndAuthor
    sAuthor = Left(s, nLen)
    nStartTitle = nEndAuthor + 2
    nEndTitle = InStr(nStartTitle, s, ";") -1
    nLen = nEndTitle - nStartTitle + 1
    sTitle = Mid(s, nStartTitle, nLen)
    nLen = Len(s) - nEndTitle
    sRest = Right(s, nLen)
    oCursor.String = sTitle + " " + sAuthor + sRest
Next n
```

After creating the search descriptor, its `SearchString` is set to `^\[[0-9]+\]`. In plain words this means: "Find all lines beginning with a left bracket. This bracket must be followed by at least one digit and a closing bracket.". The caret (^) stands for "beginning of the line", "[0-9]" denotes all digits, and the following plus sign means "at least one". We'll not go into more detail here, the StarOffice API Reference Manual contains more information on regular expressions. The search descriptor's `searchRegularExpression` attribute must be set to `TRUE` to let it know that it should not look for the literal string "^\[[0-9]+\]".

All lines matching the regular expression are then found with `findAll()`. It returns an `XTextRange()` interface which is passed to `createTextCursorByRange()`. The new cursor is thus automatically positioned at the matching regular expression. By moving it to the next word and then to the end of the paragraph, we span a text range that contains the whole bibliography entry but the leading number. This string is finally modified with some lines of pure StarBasic. They simply find the semicolons separating the author and the title and re-assemble the string from these parts.

4.3.7 Inserting tables, frames, etc.

As mentioned in Section 4.3.1 "The structure of text documents" on page 47, a text document can contain tables and other elements besides paragraphs of text. We'll show you how to insert tables and frames in your text document in this section.

A table is a special object that you can insert into a text document. You can think of it as a simple spreadsheet - a very simple one, in fact. First of all, you will see how to insert a simple table:

```
Dim oTable As Object

oCursor = oText.createTextCursor()
oCursor.gotoStart(FALSE)
oTable = oDocument.createInstance("com.sun.star.text.TextTable")
oTable.initialize(5,9)
oText.insertTextContent(oCursor, oTable, FALSE)
```

The table is created by calling `createInstance()` and then initialized to contain five rows and nine columns. It is then placed in the current document with a call to

`insertTextContent()`. This methods expects a `textCursor` as its first element. In our example, this cursor is simply placed at the beginning of the document.

Your document now contains an empty table. Since you would probably want to place text or other things inside this table, we'll show you how to do this next:

```
Dim oTableCursor As Object
Dim sCellName As String
Dim oCell As Object

oTableCursor = oTable.createCursorByCellName(oTable.CellNames(0))
oTableCursor.gotoStart(FALSE)

Dim mTableHeaders(8) As String

mTableHeaders(0) = "Field"
mTableHeaders(1) = "Format"
mTableHeaders(2) = "AutoIncrement"
mTableHeaders(3) = "Descending"
mTableHeaders(4) = "PartOfPrimaryKey"
mTableHeaders(5) = "Required"
mTableHeaders(6) = "Scale"
mTableHeaders(7) = "Size"
mTableHeaders(8) = "Type"
For n = 0 To 8
   sCellName = oTableCursor.getRangeName()
   oCell = oTable.getCellByName(sCellName)
   oCell.String=mTableHeaders(n)
   oTableCursor.goRight(1,FALSE)
Next n
```

To move around in a table, you have to create a `tableCursor` first. The sample code above uses the method `createCursorByCellName()` for this and then moves the cursor to the top left cell of the table. After initializing the array `tableHeaders` with the column headers of the table, it moves the cursor to each of the cells in the first row with `goRight()`. The content of the cell is then set by assigning to its `String` property.

If you want to insert numbers into a table, you'll probably not want them to be displayed left justified, which is the default for all cells. To change the justification for a cell depending on its content, you can use this code snippet:

```
s=oCell.String
If IsNumeric(s) Then
   oCellCursor = oCell.createTextCursor()
   oCellCursor.paraAdjust = com.sun.star.style.ParagraphAdjust.RIGHT
End If
```

We are using a text cursor here to hard format a cell if it contains a number. Alternatively, you might define a special style for numbers and assign it to the cell:

```
oCellCursor.paraStyle="myNumberStyle"
```

4.3.7.1 Inserting a frame

Frames can serve several purposes. You can use them to display graphics in a text document, to attach an icon to a certain word, and frames can be used to implement columns, because you can link them together. If the text overflows the first frame, it is automatically continued in the second one. We'll first show you how to create a simple frame that contains text:

```
Dim oFrame As Object
oCursor = oText.createTextCursor()
oText.insertString(oCursor,_
 "First paragraph", FALSE)
oText.insertControlCharacter(oCursor,_
 com.sun.star.text.ControlCharacter.PARAGRAPH_BREAK, _
 FALSE)
oText.insertString(oCursor,_
 "Second paragraph" , FALSE)
oCursor.gotoStart(FALSE)
oCursor.gotoNextWord(FALSE)
oFrame = oDocument.createInstance(_
 "com.sun.star.text.TextFrame")
Dim aSize As New com.sun.star.awt.Size

aSize.width = 2000
aSize.height = 1000
oFrame.Size = aSize
oFrame.AnchorType = _
 com.sun.star.text.TextContentAnchorType.AS_CHARACTER
oFrame.TopMargin = 0
oFrame.BottomMargin = 0
oFrame.LeftMargin = 0
oFrame.RightMargin = 0
oFrame.BorderDistance = 0
oFrame.HoriOrient = _
 com.sun.star.text.HoriOrientation.NONE
oFrame.VertOrient = _
 com.sun.star.text.VertOrientation.LINE_TOP
oText.insertTextContent(oCursor, oFrame, FALSE)
```

Here we begin with two simple text paragraphs and position the text cursor at the beginning of the first `paragraph`. Then the frame is created and some properties are set to reasonable values. In particular, the `anchorType` is specified as `AS_CHARACTER`, meaning that the frame is treated as if it were a character. Consequently, the line spacing is adjusted if the frame grows vertically. Another `anchorType` value is `AT_CHARACTER`, which simply anchors the frame at a specific character. The other properties ensure that the new frame is properly placed with respect to the text line.

```
oCursor = oFrame.createTextCursor
oCursor.charWeight = com.sun.star.awt.FontWeight.BOLD
oCursor.paraAdjust = com.sun.star.style.ParagraphAdjust.CENTER
oFrame.insertString(oCursor, "New", TRUE)
```

These four lines insert the centered word `New` in bold characters into the frame. The important thing here is to create a text cursor at the frame and to use the `insertString()` method of the frame as well.

4.3.7.2 Inserting textfields

The best known applications for textfields are probably the date and page numbers automatically inserted in a document. We'll show you how to insert the current date in a document first:

```
Dim oDateTime As Object

oCursor = oText.createTextCursor()
oText.insertString(oCursor,"Today is the ", FALSE)
oDateTime = oDocument.createInstance("com.sun.star.text.TextField.DateTime")
oDateTime.Fix = FALSE
oText.insertTextContent(oCursor,oDateTime,FALSE)
oText.insertString(oCursor," ",FALSE)
```

This code creates a single line showing the words `Today is the` followed by the current date. You reate the text field with `createInstance()` and insert it at the cursor position with `insertTextContent()`. The only property set here ensures that the date is not fixed. If you save the document and open it two days later, it will show the actual date.

The current page number is most helpful in the header or footer of a page. In the next sample, you'll see how to insert it into the footer of a page style.

```
Dim oStyleFamlies As Object
Dim oPageStyles As Object, oStdPage As Object
Dim oFooterLeft As Object, oFooterCursor As Object
Dim oFooterText As Object
Dim oPageNumber As Object

oPageNumber = oDocument.createInstance(_
 "com.sun.star.text.TextField.PageNumber")
oPageNumber.numberingType = _
 com.sun.star.style.NumberingType.ARABIC
oStyleFamilies = oDocument.StyleFamilies
oPageStyles = oStyleFamilies.getByName("PageStyles")
oStdPage = oPageStyles("Standard")
oStdPage.FooterOn = TRUE
oFooterLeft = oStdPage.FooterTextLeft
oFooterText = oFooterLeft.Text
oFooterCursor = oFooterText.createTextCursor()
oFooterText.insertString(oFooterCursor,_
 "Page ", FALSE)
oFooterText.insertTextContent(oFooterCursor, _
 oPageNumber, FALSE)
```

As before, the text field for the page number is created with `createInstance()`. The footer is then turned on for the page style `Standard` - you must do that before you can access the style's `FooterTextLeft` property. Since this is the `xText()` interface of the footer, it provides all the well known methods to insert text. We use `insertString()` to put the word `Page` before the number, which is added to the footer with `insertTextContent()`.

Field name	Meaning
PageNumber	The current page number
PageCount	Total number of pages
FileName	Name of the document
DateTime	A date, possibly automatically changed to the current date whenever the document is opened.
SetExpression	Definition of a formula or variable
GetExpression	Value of a formula or variable
Author	The author of the document, possibly updated whenever the document is modified.
Chapter	Name and/or number of the current chapter.
GetReference	Crossreference entry
ConditionalText	Text inserted depending on a condition.
Input	An entry field
HiddenParagraph	A paragraph that is only visible when a condition yields true
DocumentInfo	The information available for this document
TemplateName	Name of the template file used for this document
Database	Complete name of a field in a database.
DatabaseName	Name of a database and a table.
ParagraphCount	Total number of paragraphs
WordCount	Total number of words
CharacterCount	Total number of characters
TableCount	Total number of tables
GraphicObjectCount	Total number of graphic objects
EmbeddedObjectCount	Total number of embedded objects

4.3.7.3 Inserting graphics

You can insert graphic objects as well as frames, tables and text fields. Since this requires another StarOffice API module besides `Text()`, the process is a bit more complicated. Note that the function `createRect()` is defined later in Section 4.5.2 “Making things easier” on page 90

```
Dim oRectangleShape As Object

oCursor = oText.createTextCursor()
oText.insertString(oCursor,"First paragraph", FALSE)
oText.insertControlCharacter(oCursor,_
 com.sun.star.text.ControlCharacter.PARAGRAPH_BREAK, _
 FALSE)
oText.insertString(oCursor,"Second paragraph" , FALSE)
oCursor.gotoStart(FALSE)
oCursor.gotoNextWord(FALSE)
oRectangleShape = createRect(1000,1000,2000,500)
oRectangleShape.TextRange = oCursor.start
oRectangleShape.fillColor = RGB(255,0,0)
oRectangleShape.anchorType = _
 com.sun.star.text.TextContentAnchorType.AS_CHARACTER
oRectangleShape.HoriOrient = _
 com.sun.star.text.HoriOrientation.NONE
oRectangleShape.VertOrient = _
 com.sun.star.text.VertOrientation.LINE_TOP
oText.insertTextContent(oCursor,oRectangleShape,FALSE)
```

As before, we create two dummy paragraphs and position the text cursor at the beginning of the first word `paragraph`. The rectangle is then created using a function which will be introduced later (see Section 4.5.2 “Making things easier” on page 90). The differences begin now: We could insert tables, frames and text fields directly using `insertTextContent()`. The rectangle, however, is created on the `DrawPage` associated with the current page of the text document. To create it, you have to provide its position and dimensions in 100ths of a millimeter. Since you can’t deduce the position from your text document, you simply specify a dummy position and link the graphic object to the correct position by setting its `TextRange` property to the start position of `oCursor`. The rectangle is then made visible and attached to the cursor position by inserting it as before with the `insertTextContent()` method.

4.3.8 Locating text content

As you have seen in the preceding section, text documents can contain other content besides normal text. If you want to modify this content, you have to be able to locate it first. This is fairly easy, because StarOffice API’s `text` module contains appropriate interfaces for each type of content. For example, to get an array of all tables in a document, you could use this:

```
Dim oTextTables As Object
Dim oTable As Object
Dim n As Integer

oTextTables = oDocument.getTextTables()
For n = 0 to oTextTables.count - 1
  oTable = oTextTables(n)
   REM Do something with oTable
Next n
```

To get the frames, you use `getTextFrames()`, for text fields you employ `getTextFields()`, and for graphics you call `getGraphicObjects()`. Once you have found the content you are looking for, either by index or by name access, you can inquire and change its properties.

4.4 Sheet

Sheet is the module that contains spreadsheet services. It is used like the `text` service, since it needs a document to work with:

```
Global oDesktop As Object
Global oDocument As Object

Sub sheetdoc_init
   Dim mNoArgs() REM Empty Sequence
   Dim sUrl As String

   oDesktop  = createUnoService("com.sun.star.frame.Desktop")
   sUrl = "private:factory/scalc"
   REM Or: sUrl = "file:///home/testuser/Office52/work/table.sdc"
   oDocument = oDesktop.LoadComponentFromURL(sUrl,"_blank",0,mNoArgs)
End Sub
```

In the following examples, we will assume you have opened the document as described. You'll learn how to address cells and ranges of cells, how to navigate through sheets, and how to draw a chart from sheet data.

4.4.1 Using Cells and Ranges

Most spreadsheet documents contain several sheets. When you create an empty one, it always comes with three sheets. The sheets consist of cells, each of which can be empty or contain text, a value, or a formula. Every cell has an address which contains the number of the sheet, the row, and the column number.

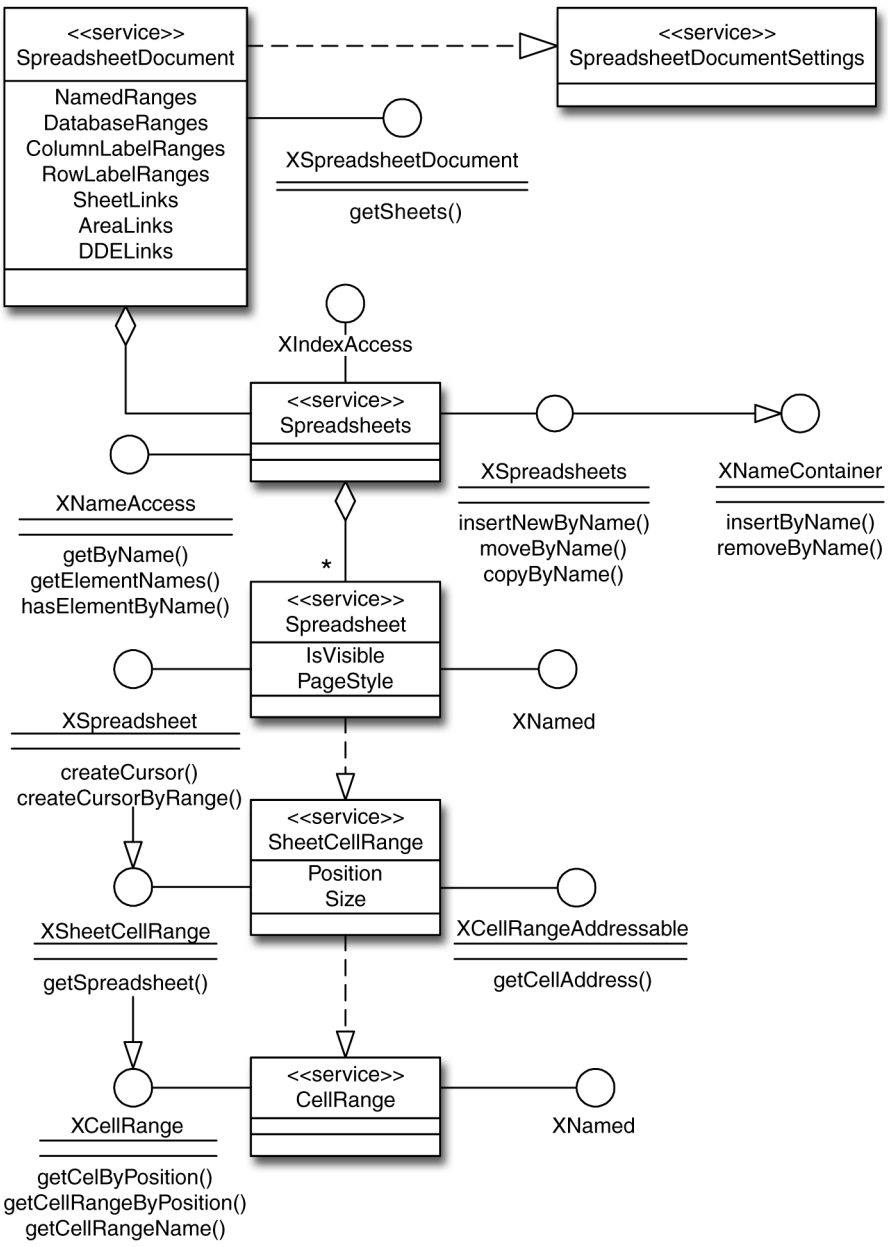

Figure 4–8 Structure of a SpreadsheetDocument

Rows and columns can be specified numerically, starting with 0, or as strings. In the latter case, the cell in the upper left corner of a sheet has the address `A1`. The next cell to the right is named `A2`, the one below is `B1`. A rectangular region of cells is

called a range. To address cells or ranges of cells in a spreadsheet, you first must select the sheet before you can do anything else.

```
oSheet1 = oDocument.Sheets(0)
```

As usual, the first sheet is numbered zero. If you don't know how many sheets your document has, use `count()`:

```
nSheetCount = oDocument.Sheets.Count
```

A single cell is adressed like this:

```
Function GetCell (oDocument As Object, _
                  nSheet As Long, nColumn As Long ,_
                  nRow As Long) As com.sun.star.table.XCell

  Dim oSheets As Object
  Dim oSheet As Object

  oSheets = oDocument.Sheets()
  oSheet = oSheets.getByIndex(nSheet)
  GetCell = oSheet.getCellByPosition (nColumn , nRow)
End Function
```

This function returns an `XCell` interface of the `SheetCell` service. A range of cells can be specified like this:

```
sTopLeft = "b2"
sBottomRight = "d8"
oCellRange = oSheet1.getCellRangeByName(sTopLeft + _
 ":" + sBottomRight)
```

Alternatively, you can use a function to obtain a cell range:

```
Function GetCellRange (oDocument As Object, _
          nSheet As Long, _
   nCol1 As Long , nRow1 As Long, _
   nCol2 As Long , nRow2 As Long) _
   As com.sun.star.table.XCellRange

   Dim nLeft As Long
   Dim nRight As Long
   Dim nTop As Long
   Dim nBottom As Long
   Dim oSheets As Object
   Dim oSheet As Object

   nLeft   = iif(nCol1 < nCol2, nCol1, nCol2)
   nRight  = iif(nCol1 < nCol2, nCol2, nCol1)
   nTop    = iif(nRow1 < nRow2, nRow1, nRow2)
   nBottom = iif(nRow1 < nRow2, nRow2, nRow1)
   oSheets = oDocument.Sheets()
   oSheet  = oSheets.getByIndex(nSheet)
   GetCellRange = _
    oSheet.getCellRangeByPosition(nLeft, nTop, nRight, nBottom)
End Function
```

This function accepts the corners of the range in any order, it determines the top, left, bottom, and right cells to pass to `getCellRangeByPosition()`.

If you have a range of cells, you might want to work with a single cell. This can be achieved with `getCellByPosition()`:

```
oCell = oCellRange.getCellByPosition(0,0)
```

would return the cell at the top left corner of the range. All values passed to `getCellByPosition()` are relative to the range. To get the right bottom cell, you'd use

```
nCols = oCellRange.Columns.Count
nRows = oCellRange.Rows.Count
oCell = oCellRange.getCellByPosition(nCols - 1, nRows -1)
```

Note that we subtract 1 from the number of rows and columns here, because the numbering starts at zero.

4.4.1.1 Calculations

You can perform some simple calculations on ranges without using a formula.

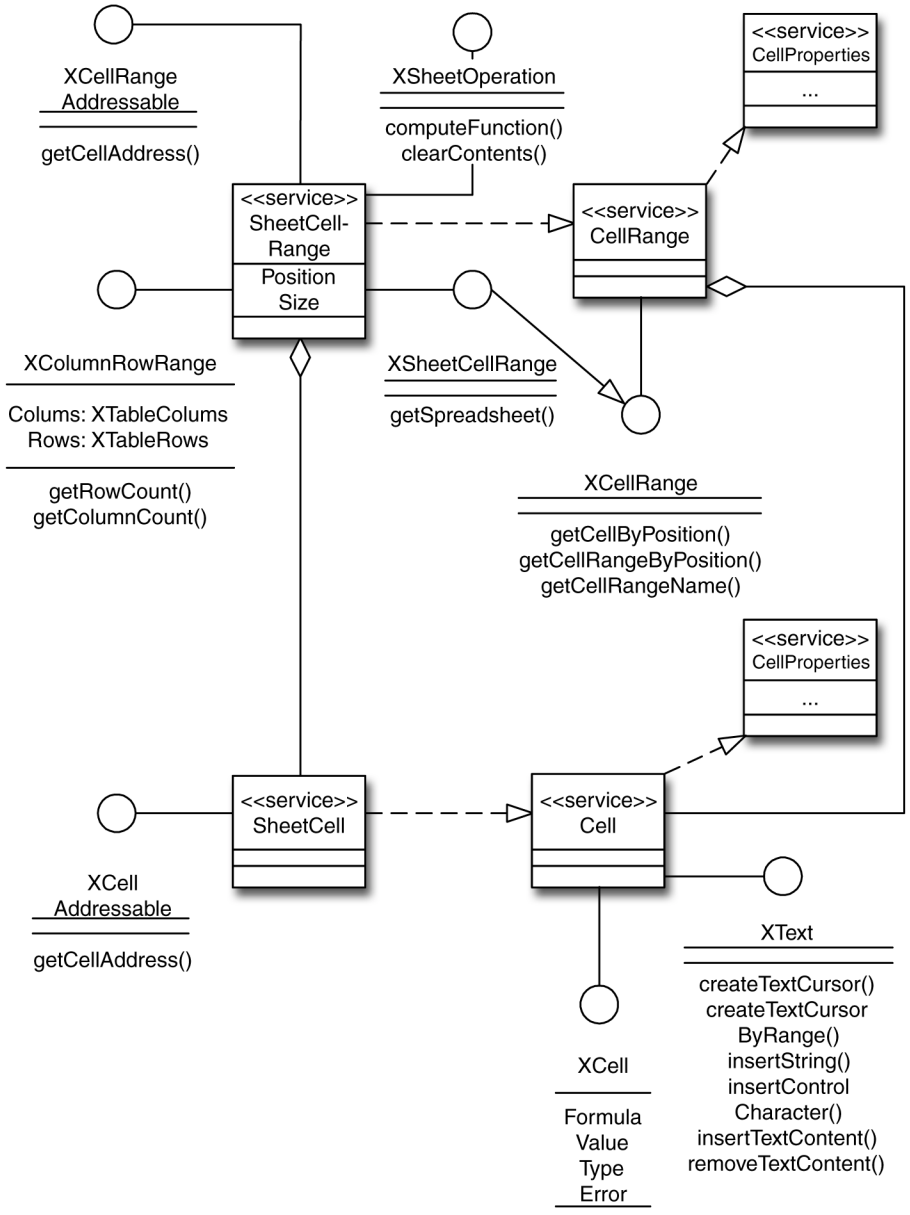

Figure 4–9 Calculations using XSheetOperation interface

The following sample code illustrates this as well as the way to enter data into cells. It uses the function `getCellRange()` defined above.

```
Dim oCell As Object, oRange As Object
Dim nCols As Long, nRows As Long
Dim oCellSum As Object, oCellAvg As Object
Dim eSumFunc As Long, eAvgFunc As Long
Dim nSum As Long, nAvg As Long
Dim n As Long

oRange = GetCellRange(oDocument,0, 1, 0, 1, 5)
nCols = oRange.Columns.Count
nRows = oRange.Rows.Count
For n = 0 To nRows-1
  oCell = oRange.getCellByPosition(0,n)
  oCell.Value = n
Next n
eSumFunc = _
 com.sun.star.sheet.GeneralFunction.SUM
eAvgFunc = _
 com.sun.star.sheet.GeneralFunction.AVERAGE
nSum = oRange.computeFunction(eSumFunc)
nAvg = oRange.computeFunction(eAvgFunc)
oCell = GetCell(oDocument,0,0,nRows)
oCell.String = "Sum"
oCell = GetCell(oDocument,0,0,nRows +1)
oCell.String = "Average"
oCellSum = GetCell(oDocument,0,1,nRows)
oCellSum.Value = nSum
oCellAvg = GetCell(oDocument,0,1,nRows + 1)
oCellAvg.Value = nAvg
```

This program fills all cells in the range B1 through B6 with the numbers 0 through 5 (see `for` loop). It then computes the sum and the average over these values. Cells A7 and A8 are labeled with `Sum` and `Average`, respectively and the calculated values are then written into the cells B7 and B8. Although the values are as correct as they can be, this calculation method has a serious disadvantage: Whenever one of the values used changes, you must recalculate the sum and average yourself. Since this is precisely the job of a spreadsheet, you should use a formula for this purpose:

```
oCellAvg.Formula = "=SUM(B1:B6)/"+ CStr(nRows)
```

This line changes the content of cell B7 so that it contains a formula instead of the value.

To summarize: To set the content of a cell to text, you use the `String` property, to enter a value in a cell, you set the `Value` property. If you want to put a formula in a cell, you assign it (including the equals sign) to the `Formula` property. Note that function names must be English if you use the `Formula` property. To use functions in your local language, you must use the `FormulaLocal` property. If you want to know what a cell contains, you can retrieve its `Type` property:

```
Sub Printinfo (oCell As Object)
  Dim eType as Long

  eType = oCell.Type
  If eType = com.sun.star.table.CellContentType.VALUE Then
    Print CStr(oCell.Value)
  Elseif eType = com.sun.star.table.CellContentType.TEXT Then
```

```
    Print oCell.String
  Elseif eType <> com.sun.star.table.CellContentType.EMPTY Then
    Print oCell.Formula + "..." + oCell.FormulaLocal
  Else
    Print "Cell Is empty"
  End If
End Sub
```

This piece of code simply outputs the content of a cell as a string. If it is a formula, it is shown in the English and the local variant.

4.4.2 Formatting cells

One way to format cells are styles. With them you can specify *how* something is formatted. We are not going to explain this here in more detail, you can use the techniques explained in Section 4.1 “Styles” on page 31. The more interesting formatting tasks in stylesheets are those dealing with the representation of cell content - how do you ensure that a value is shown as percentage or as a date? The following examples will show you how to influence *what* is displayed.

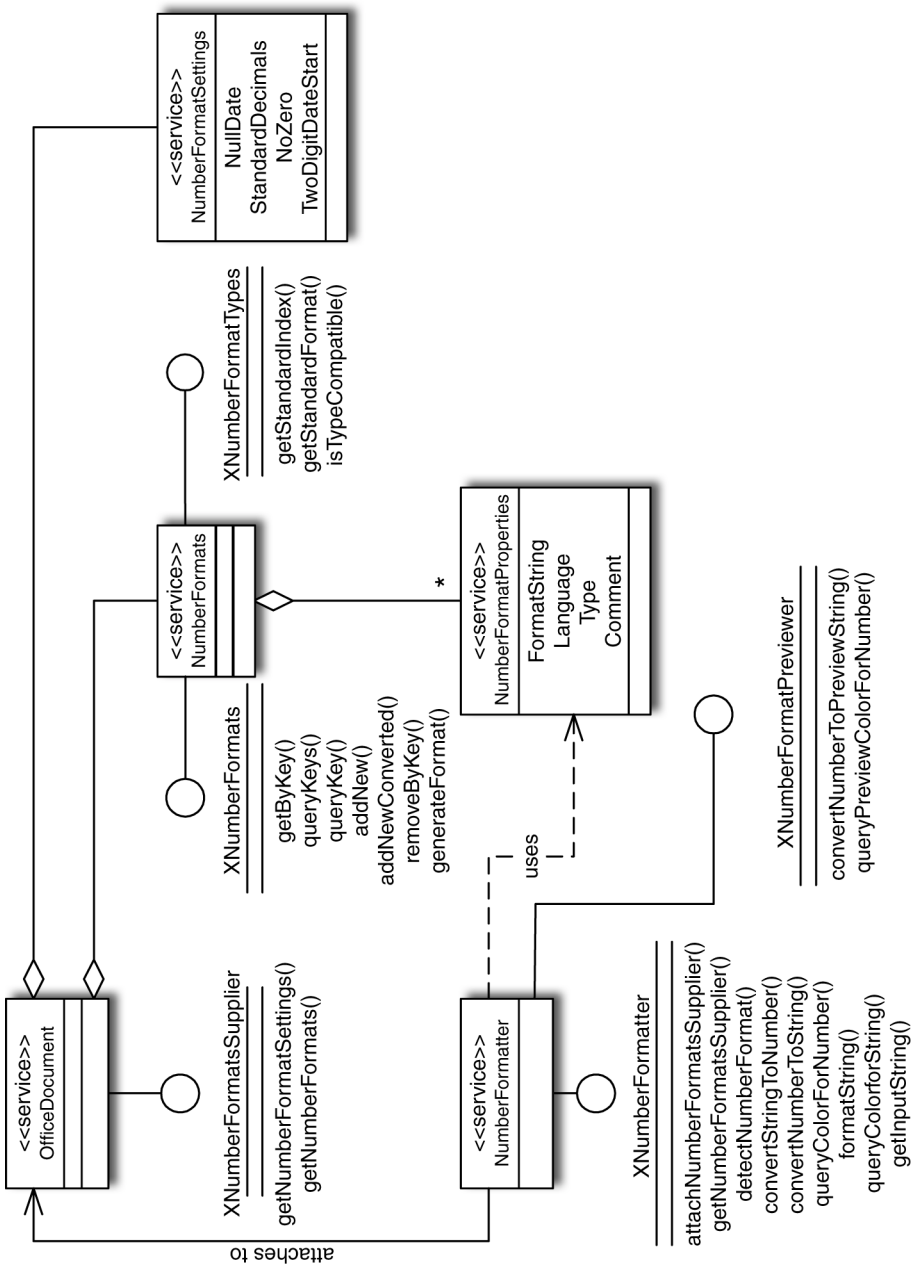

Figure 4–10 Formatting cells using NumberFormatter service

The key concept here is the `NumberFormatter()` interface. It provides access to all formats available in the sheet cells and permits you to define new formats as well. StarOffice API comes with a huge collection of predefined number formats from

which you can choose. We will first show you how to find an appropriate format and then how to define your own number format.

Predefined formats can be accessed by language and category. As long as you don't want to use a language different from the default, you should always use 0 as value for the language parameter.

```
Dim oFormats As Object
Dim mKeys As Variant
Dim oLocale As Object

oFormats = oDocument.NumberFormats
mKeys = oFormats.queryKeys(0, oLocale, FALSE)
```

These two lines return the list of all available formats, which is usually more then you really want. To get just the currency formats defined for your language, you could use

```
oFormats = oDocument.NumberFormats
mKeys = oFormats.queryKeys(_
 com.sun.star.util.NumberFormat.CURRENCY, oLocale, FALSE)
```

`mKeys` is now a sequence of numbers, each of which addresses a particular format definition. To find a format which displays all negative amounts in red, inserts a separator after every third digit ("thousands separator"), and shows no leading zeros, you could use the following lines of code:

```
Dim nFoundFormat As Long
Dim nKey As Long
Dim mProperties As Variant
Dim vProperty As Variant
Dim nI As Long, nJ As Long
Dim nLZ As Long
Dim bNR As Boolean
Dim bTS As Boolean
Dim sNewFormat As String
nFoundFormat = -1
For nI = LBound(mKeys) To UBound(mKeys)
  nKey = mKeys(nI)
  mProperties = oFormats.getByKey(nKey).getPropertyValues()
  For nJ = LBound(mProperties) To UBound(mProperties)
     vProperty = mProperties(nJ)
     If vProperty.Name = "LeadingZeros" Then
       nLZ = vProperty.Value
     End If
     If vProperty.Name = "NegativeRed" Then
       bNR = vProperty.Value
     End If
     If vProperty.Name = "ThousandsSeparator" Then
       bTS = vProperty.Value
     End If
  Next nJ
  If nLZ = 0 And bNR = TRUE And bTS = TRUE Then
    nFoundFormat = nKey
  End If
Next nI
```

```
If nFoundFormat < 0 Then
  sNewFormat = oFormats.generateFormat( mKeys(0), _
    oLocale, TRUE,TRUE, 2, 0 )
  nKey = oFormats.queryKey( sNewFormat, oLocale, TRUE )
  If nKey=-1 Then
    nKey = oFormats.addNew( sNewFormat, oLocale )
  End If
  nFoundFormat = nKey
End If
```

First of all, we initialize the variable `nFoundFormat` to -1. This is never a valid format key. If the value is still -1 after we have searched through all formats, we know that none of them suited our needs. We then loop over the `keys` array. The method `getByKey()` returns an array of properties associated with each format. The next loop iterates over these properties and copies the values of `LeadingZeros`, `NegativeRed`, and `ThousandsSeparator` in the variables `nLZ`, `bNR`, and `bTS`, respectively. If these variables have the desired values after the loop is terminated, the current format's key is saved in `nFoundFormat`. When we have looked at all currency formats, we check whether `nFoundFormat` is positive. If not, there was no matching format found and we have to define one ourselves. The method `generateFormat()` takes care of the next most important step. Despite its name, it doesn't generate a new format. Instead, it translates a format definition into a format string. We pass in all the attributes we want the new format to have as well as the key of a base format from which it inherits all the non-specified attributes. We can then use the format string returned by `generateFormat()` to ask `queryKey()` for a format which matches the string. If it returns -1, no such format exists and we add a new one by passing the format string to `addNew()`. The next few lines will show you how to apply the new format:

```
Dim oRange As Object
Dim oCell As Object

oRange = oDocument.Sheets(0).getCellRangeByPosition(0,0,0,2)
oRange.numberformat = nFoundFormat
oCell = oRange.getCellByPosition(0,0)
oCell.Value = 10000.2693
oCell = oRange.getCellByPosition(0,1)
oCell.Value = 0.3
oCell = oRange.getCellByPosition(0,2)
oCell.Value = -20
```

This code first creates a range spanning the top three cells in the first row. It then sets the `numberFormat` property of these cells to the key of the just found or created format. Finally, three numbers are inserted in the three cells. The first illustrates the insertion of a thousands separator, the second one will show up without a leading zero (which is not particularly useful for a currency format), and the last one will be displayed in red.

4.4.3 Drawing a Chart from Data

StarOffice can produce diagrams (called "charts") from data in a spread sheet. The details of charts are handled by a different module, `com.sun.star.chart`. Charts are "embedded" into spreadsheets and are accessible through a named collection from the spreadsheet.

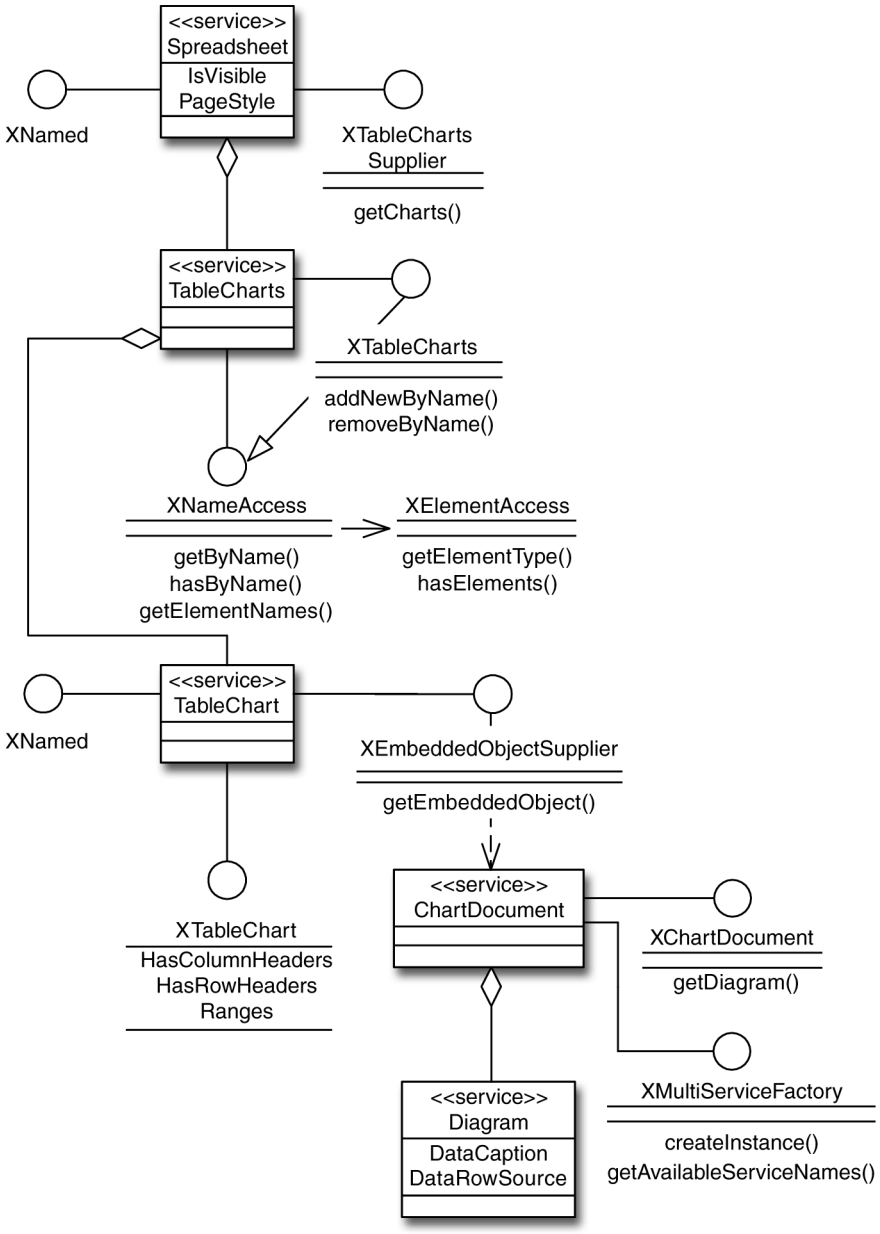

Figure 4–11 Drawing a chart diagram from spreadsheet data

Three steps are required to create a chart from spreadsheet data:

1. Define the data to display in the chart and the rectangular area in the sheet where it is to be drawn.

2. Add the chart to the named collection.
3. Set all chart properties to the desired values.

In its most simple form, a chart takes just a rectangular region of the sheet containing the data and another rectangle defining the size of the chart. For the following examples, we will assume that the upper left corner of the first sheet contains this data:

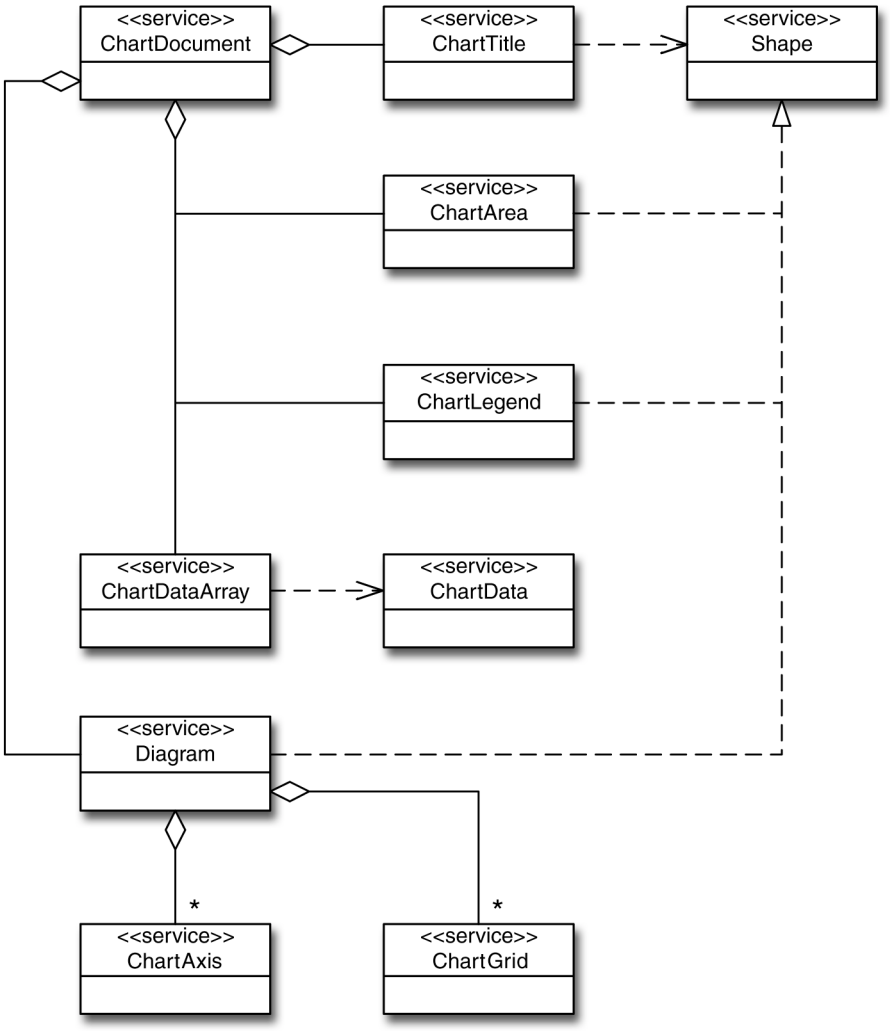

Figure 4–12 Structure of a ChartDocument

Year	Prod. A	Prod. B
2000	15	15
2001	8	10
2002	5	20

2003	10	15
2004	20	25

We will assume that the numbers are sales for two products.

4.4.3.1 Creating barcharts

```
Global aRect As New com.sun.star.awt.Rectangle
Global mRangeAddress(0) As New com.sun.star.table.CellRangeAddress
Global oCharts As Object
Sub chart_init
  sheetdoc_init()
  aRect.X = 8000
  aRect.Y = 1000
  aRect.Width = 10000
  aRect.Height = 10000
  mRangeAddress(0).Sheet = 0
  mRangeAddress(0).StartColumn = 0
  mRangeAddress(0).StartRow = 0
  mRangeAddress(0).EndColumn = 2
  mRangeAddress(0).EndRow = 5
  oCharts = oDocument.Sheets(0).Charts
End Sub


Sub chart1_sample
  chart_init()
  oCharts.addNewByName("BarChart",aRect,mRangeAddress(),TRUE, TRUE)
End Sub
```

The `rect` used in this example is the area on the physical page where the diagram is to be drawn. Its position and size are given in 100ths of a millimeter. The rectangle is thus positioned eight centimeters from the left and one centimeter from the upper margin, it is 10 centimeters high and wide. There is no direct relationship between the rectangle in which the diagram is drawn and the position of the cells in the sheet.

The data used to draw the chart is found in the range specified by `rangeAddress`. It starts at the upper left corner of the sheet (cell A1) and extends to the cell in the third column, sixth row (C6). To indicate that the first row and column of this range contain the labeling for the chart, the last two parameters of `addNewByName()` are set to TRUE.

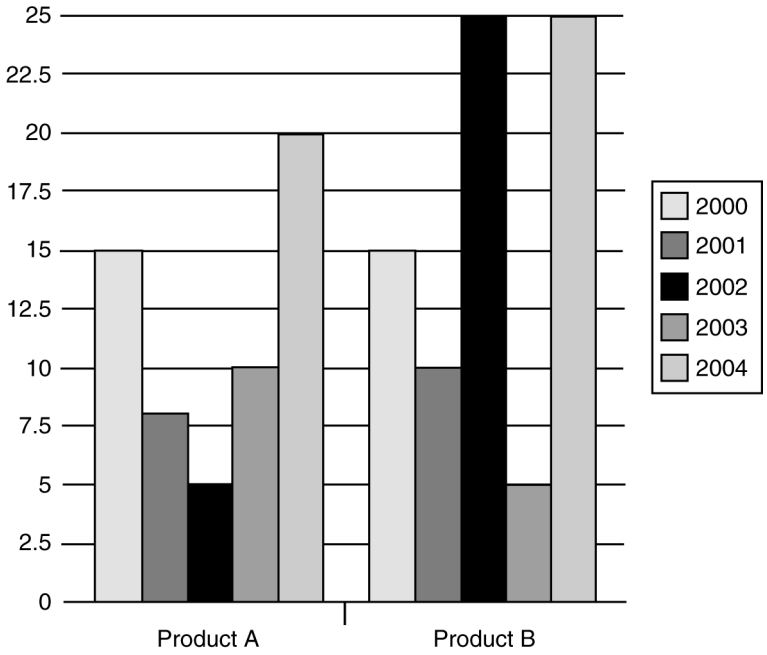

Figure 4–13 BarChart sample 1

This example creates a bar diagram interpreting the table rows as the data pairs to plot. Therefore, you see two groups of bars, one for each product. Every group contains one bar for each of the years 2000 through 2004. To show the two products side by side for each year you must tell StarOffice API to interpret the data by columns:

```
Sub chart2_sample
  chart_init()
  oCharts.addNewByName("BarChart",aRect,mRangeAddress(),TRUE, TRUE)

  Dim oChart As Object

  oChart = oCharts.getByName("BarChart").embeddedObject
  oChart.diagram.DataRowSource = com.sun.star.chart.ChartDataRowSource.COLUMNS
End Sub
```

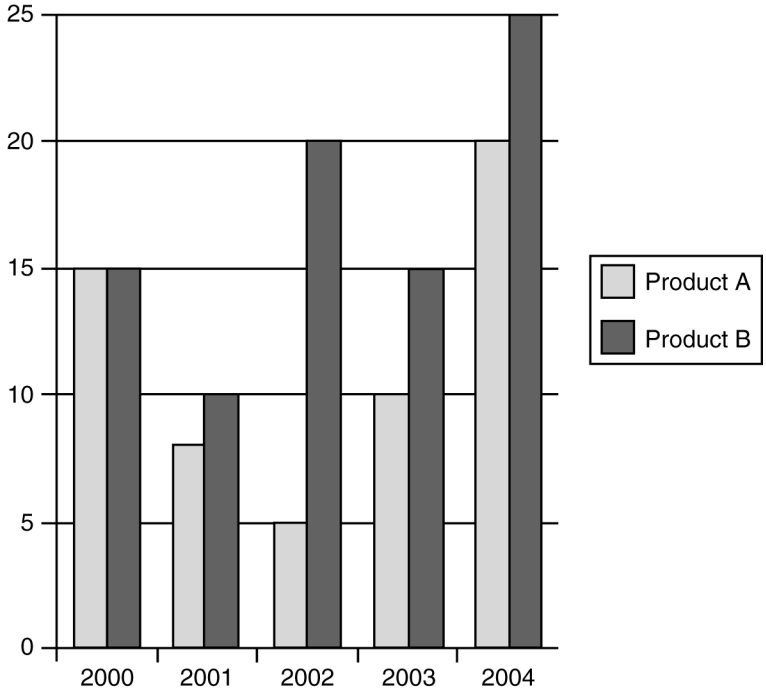

Figure 4–14 BarChart sample 2

Setting the `DataRowSource` property to `com.sun.star.chart.ChartDataRowSource.COLUMNS` is all that is needed to change the chart. Note that you must do that *after* you have added the new chart with `addNewByName()`.

4.4.3.2 Creating linecharts

Although a barchart illustrates the development of the two products over the years, a line diagram would show the trends more clearly. To create a line diagram instead of a bar chart, you can use these lines:

```
Sub chart3_sample
  Dim oChart As Object

  chart_init()
  oCharts.addNewByName("LineDiagram",aRect,mRangeAddress(),TRUE, TRUE)
  oChart = oCharts.getByName("LineDiagram").embeddedObject
  oChart.diagram =_
   oChart.createInstance("com.sun.star.chart.LineDiagram")
  oChart.diagram.DataRowSource = com.sun.star.chart.ChartDataRowSource.COLUMNS
End Sub
```

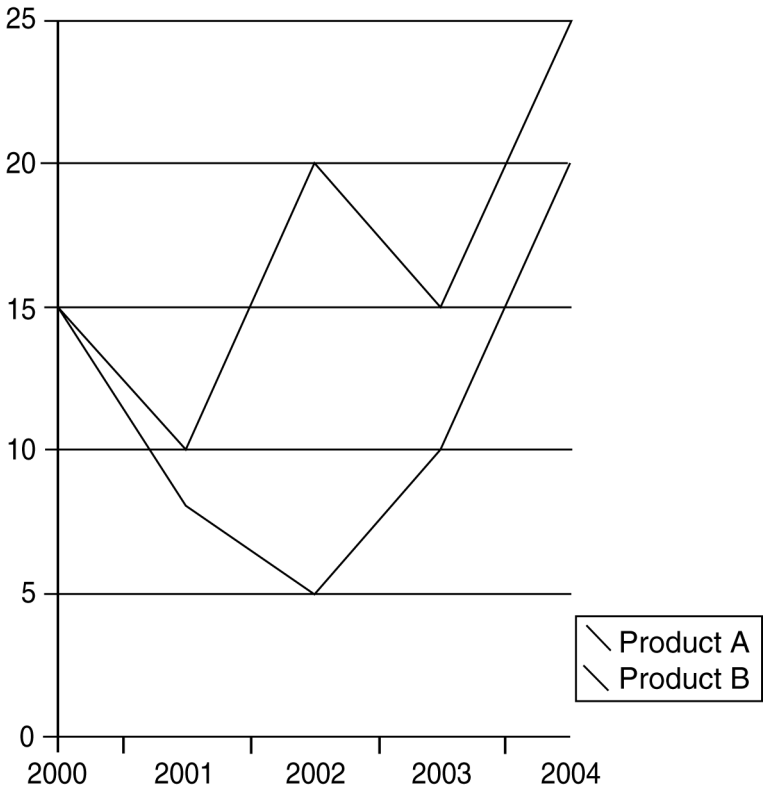

Figure 4–15 LineChart sample

This chart displays one line for each product. You can thus clearly see the overall tendency and the sharp fall for product A in 2002.

4.4.3.3 Creating pie charts

Suppose A and B were the only products of a company and you wanted to see how each of them contributed to the total revenues in every year. One way to visualize this is to use pie charts like this:

```
Sub chart4_sample
   Dim oCell As Object
   Dim sChartName As String
   Dim sYear As String
   Dim n As Long
   Dim oRange As Object
   Dim oChart As Object
   Dim mRangeAddress(1) As New com.sun.star.table.CellRangeAddress
   Dim aRect As New com.sun.star.awt.Rectangle

REM see section "Importing-other-formats"
REM   import1_sample()
   aRect.X = 8000
```

```
   aRect.Y = 1000
   aRect.Width = 3000
   aRect.Height = 3000
   mRangeAddress(0).Sheet = 0
   mRangeAddress(0).StartColumn = 0
   mRangeAddress(0).StartRow = 0
   mRangeAddress(0).EndColumn = 2
   mRangeAddress(0).EndRow = 0
   oRange = GetCellRange(oDocument,0, 0, 1, 0, 5)
   For n = 0 To 4
      oCell = oRange.getCellByPosition(0,n)
      sYear = CStr(oCell.value)
      mRangeAddress(1).Sheet = 0
      mRangeAddress(1).StartColumn = 0
      mRangeAddress(1).StartRow = n + 1
      mRangeAddress(1).EndColumn = 2
      mRangeAddress(1).EndRow = n +1
      oCharts = oDocument.Sheets(0).Charts
      sChartName = "PieChart" + sYear
      oCharts.addNewByName(sChartName,_
       aRect,mRangeAddress(),TRUE, TRUE)
      oChart = oCharts.getByName(sChartName).EmbeddedObject
      oChart.diagram = _
       oChart.createInstance("com.sun.star.chart.PieDiagram")
      oChart.Title.String = sYear
       aRect.Y = aRect.Y + 4000
   Next n
End Sub
```

This gives you one row of pie charts to the right side of the table - unless you have changed the default table settings. In this case, the charts may well end up obscuring part or all of your table. The reason for this is that we are using a fixed starting point (8000/1000) for the first chart and draw every other chart just below the preceding one.

The sample code above illustrates one thing we have not mentioned before: Your data doesn't have to be contained in consecutive rows or columns. As you can see above, the second element of `rangeAddress` changes for each pie, because we need one pie per row. The first element of `rangeAddress` remains the same, because it contains the strings for the legend.

To make chart placement more reasonable, you should figure out the dimensions of your table and decide on the position of your chart depending on them. To illustrate this, we will use the same data as before and decide on the chart placement based on the table's dimension.

```
REM ... same code As before
oRange = GetCellRange(oDocument,0,0,0,2,5)
nTableWidth = oRange.Size.Width + oRange.Position.X
nPieRadius = 1500
aRect.X = oRange.Position.X
aRect.Y = oRange.Position.Y + oRange.Size.Height * 1.1
aRect.Width = nPieRadius * 2
aRect.Height = nPieRadius * 2
```

After specifying a cell range, we use its `Position` and `Size` properties to specify the position for the first pie chart. It is placed so that its left margin coincides with the table range and its top margin is just a bit below the last cell. The variable `pieRadius` is used to make the following code more legible.

```
If aRect.X + 3500 + nPieRadius*2 > nTableWidth Then
 aRect.X = 0
  aRect.Y = rect.Y + 3500
Else
  aRect.X = rect.X + 3500
End If
```

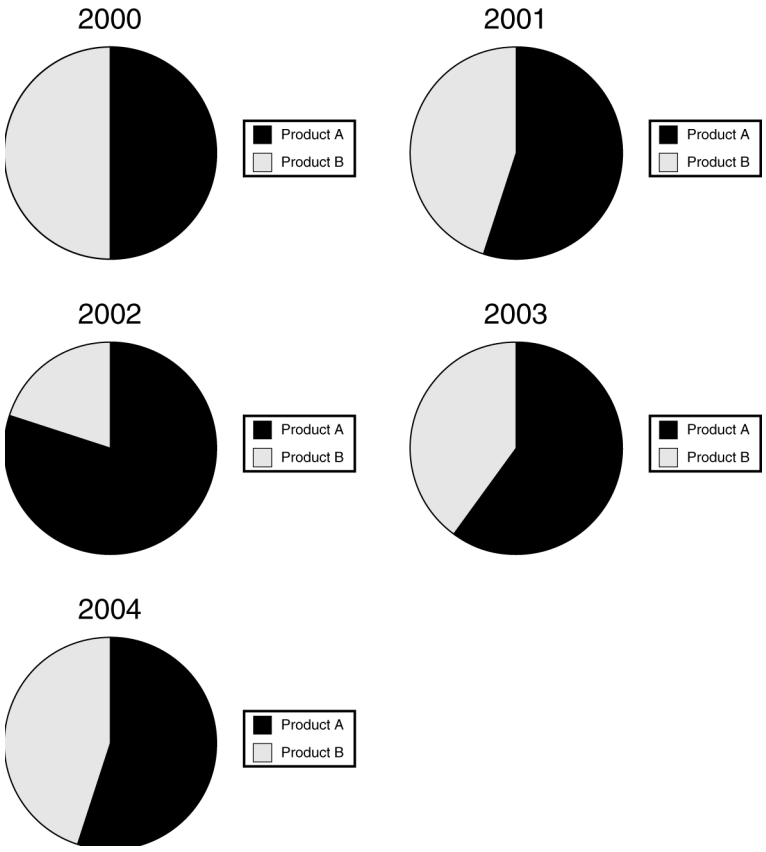

Figure 4–16 PieChart sample

These lines make sure that the next pie is positioned to the right of the previous one only if this would not exceed the width of the table. If it does, the chart is placed flush at the left margin under the last row of charts. You can use a method like this to place your charts near the table they belong to.

4.5 Drawing

StarOffice contains a module to create vector graphics (lines, rectangles, circles etc.). These drawings can then be included in other documents like text.

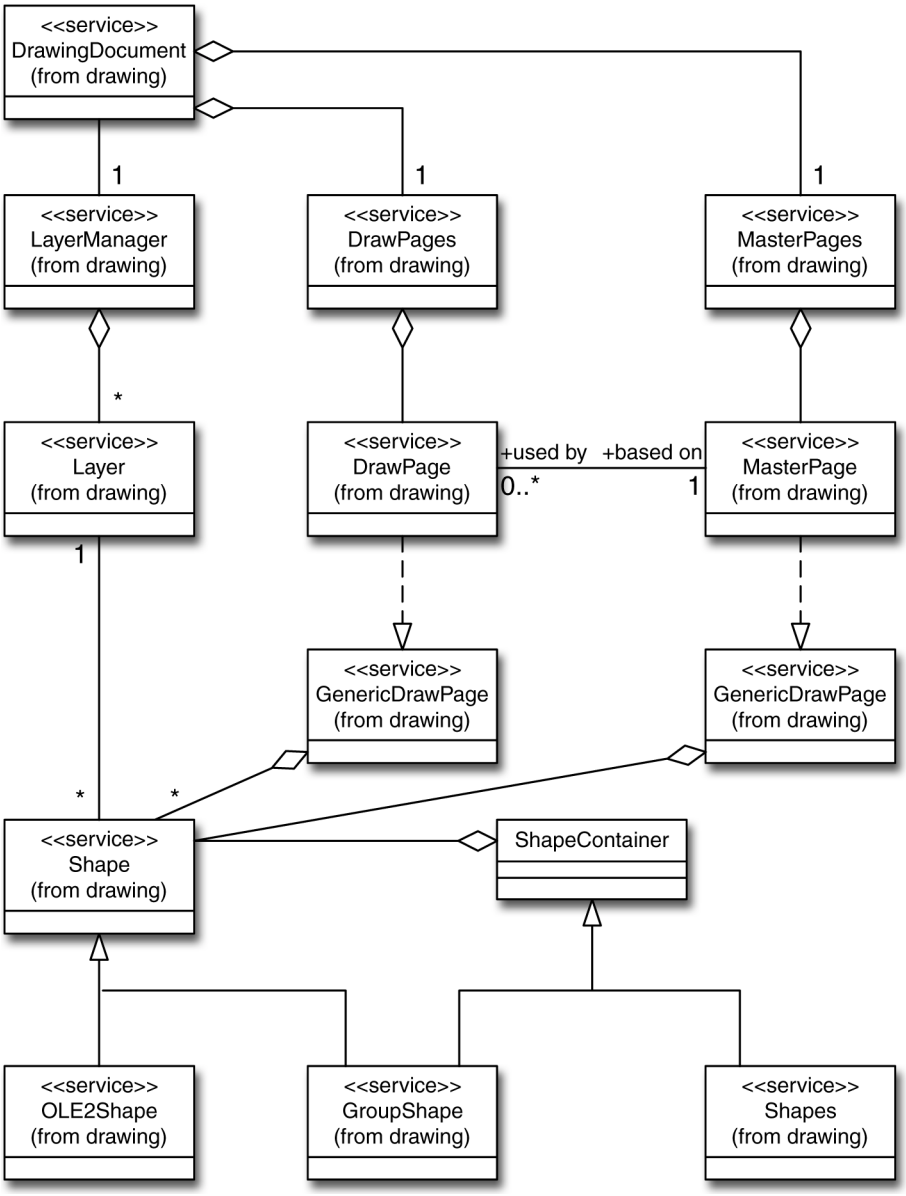

Figure 4–17 Structure of DrawingDocument

In this section we'll show you how to create graphics, how to group graphics objects and how to change their attributes.

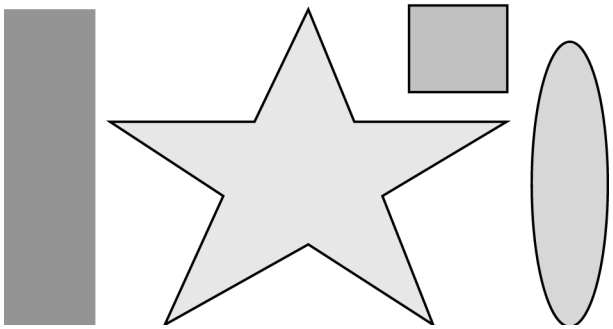

Figure 4–18 Shapes

Drawing is handled by a service that works only in documents. You have thus first to open an existing document or to create a new one like this.

```
Global oDesktop As Object
Global oDocument As Object
Global oPage As Object
Sub drawdoc_init
  Dim mNoArgs()
  Dim sUrl As String

  oDesktop  = createUnoService("com.sun.star.frame.Desktop")
  sUrl = "private:factory/sdraw"
  REM Or:  sUrl = "file:///home/testuser/office52/work/image.sdd"
  oDocument = oDesktop.LoadComponentFromURL(sUrl,_
     "_blank",0,mNoArgs())
  oPage = oDocument.drawPages(0)
End Sub
```

Note that the first URL creates a new document, while the second one (after the "or" comment) opens an existing document. `oDocument` now provides the `XDrawPages()` interface which in turn is the main entry point for all drawing functionality. We use the first drawing page here and assume in the following examples that you have opened or created a document as described.

The draw module differs from other StarOffice API modules we have seen so far in that in can actually draw on *any* type of document. It is therefore not necessary to create an `sdraw` document or open an `sdd` file. You could as well create a text document or open a spreadsheet. The important point is that you get the document's drawing page, because all graphics objects are attached to it.

The main concepts in the drawing module are the drawing page and shapes. Shapes are the actual graphic objects which you see. They are always attached to a drawing page which is responsible for displaying them. A shape remains invisible as long as it is not added to a drawing page.

4.5.1 Creating simple shapes

Shapes are rectangles, circles etc. They make heavy use of some basic structures like `com.sun.star.awt.Point` and `com.sun.star.awt.Size` to specify coordinates and dimensions. All coordinates and dimensions are given in 100ths of a millimeter.

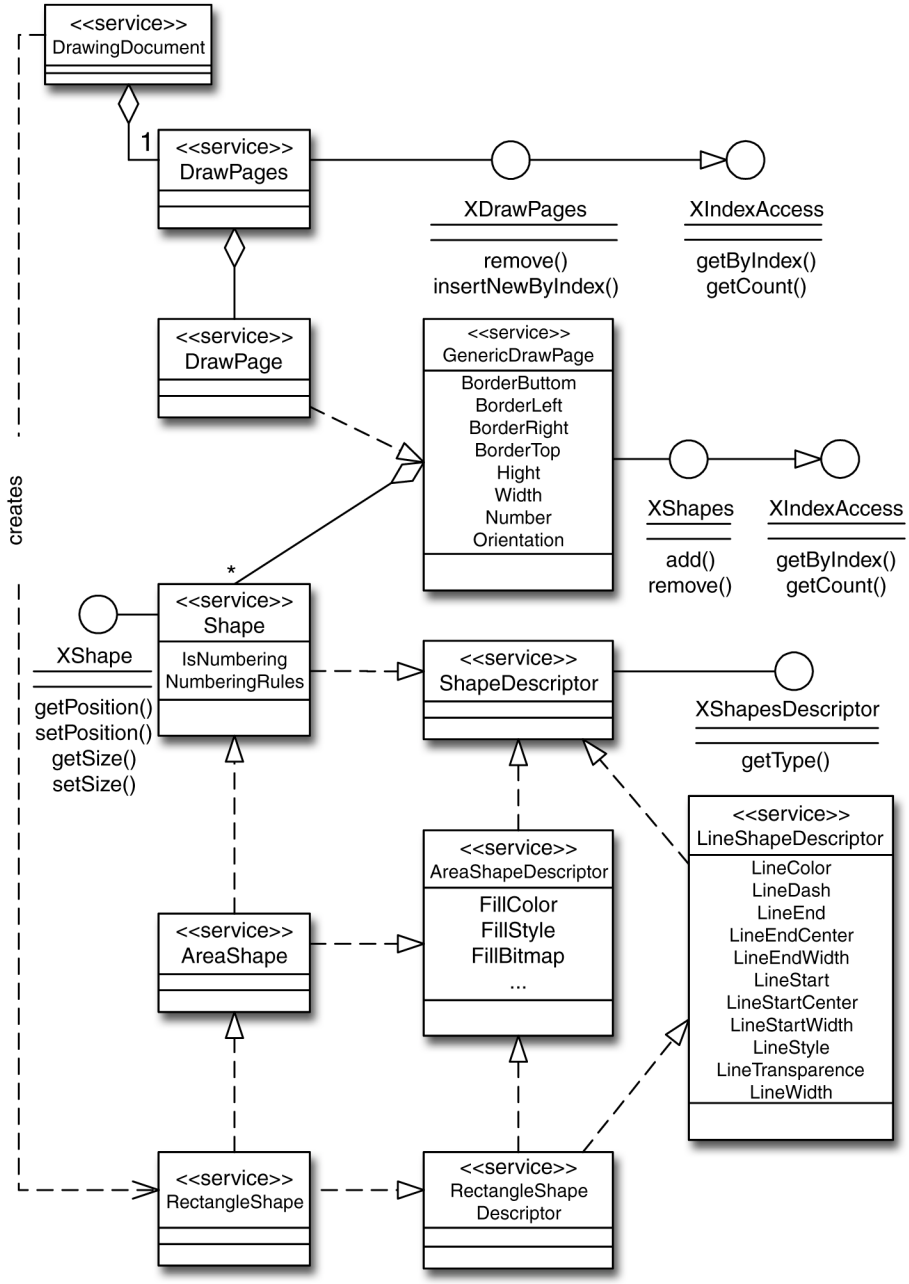

Figure 4–19 Creating simple shapes

```
Dim aPoint As New com.sun.star.awt.Point
Dim aSize As New com.sun.star.awt.Size
```

```
aPoint.x = 1000
aPoint.y = 1000
aSize.Width = 10000
aSize.Height = 10000

Dim oRectangleShape As Object

oRectangleShape = oDocument.createInstance("com.sun.star.drawing.RectangleShape")
oRectangleShape.Size = aSize
oRectangleShape.Position = aPoint
oRectangleShape.FillColor = RGB(255,0,0)
oPage.add(oRectangleShape)

Dim oEllipseShape As Object

oEllipseShape = oDocument.createInstance("com.sun.star.drawing.EllipseShape")
oEllipseShape.Position = aPoint
oEllipseShape.Size = aSize
oEllipseShape.FillColor = RGB(0,255,0)
oPage.add(oEllipseShape)

Dim oTextShape As Object
oTextShape = oDocument.createInstance("com.sun.star.drawing.TextShape")
aPoint.x = 11000
aPoint.y = 11000
oTextShape.Position = aPoint
oTextShape.Size=aSize
oPage.add(oTextShape)
oTextShape.String="Hello World"
```

This example creates a red square whose side is 10 centimeters long (`aSize`). Its upper left corner is one centimeter right and down from the upper left corner of the page (`aPoint`). The same values are used for the green circle. In general, the `Size` property of a shape defines the bounding box in which it is contained. The `Position` property specifies the position of the bounding box's upper left corner on the page. Coordinates on a page start at the upper left with (0,0) and extend to the right and the bottom.

You can learn several important points from the preceding example:

1. All shapes are first created by `createInstance()` which expects the name of the shape as its parameter. Then all properties are set, and finally the shapes are made visible with a call to the draw page's `add()` method.
2. StarOffice API does not provide a circle shape. Instead, you use an ellipse with a `size` property that specifies a square.
3. The string of a text shape can be set only after it has been added to the page. In general, you can only set those properties that are provided directly by the service `ShapeDescriptor` and indirectly by the interface `XShape`.

If you want to change the visual appearance of a shape after you have created it, you simply modify the corresponding property. This works as long as you didn't add this shape to a group. Shapes forming a group can be modified separately using an index access method. You can apply geometric transformations to the whole group.

4.5.2 Making things easier

To facilitate working with the `drawing` Module, we'll present some functions which you can use in your programs to create objects. Each function returns the object it has created, it expects the current document as its first parameter. We'll not explain these functions further, they just do what the name implies.

```
Function createSquare (nX As Long, nY As Long,_
                       nWidth As Long) As Object
REM Creates a square with upper left
REM at (nX,nY) with sides = nWidth

  createSquare = createRect(nX,nY,nWidth,nWidth)
End Function

Function createRect (nX As Long, nY As Long,_
                     nWidth As Long,_
                     nHeight) As Object
REM Creates a rectangle with upper left
REM at (nX,nY), width = nWidth, height = nHeight

  Dim aPoint As New com.sun.star.awt.Point
  Dim aSize As New com.sun.star.awt.Size
  Dim oRectangleShape As Object

  aPoint.X = nX
  aPoint.Y = nY
  aSize.Width = nWidth
  aSize.Height = nHeight
  oRectangleShape = oDocument.createInstance("com.sun.star.drawing.RectangleShape")
  oRectangleShape.Size = aSize
  oRectangleShape.Position = aPoint
  createRect = oRectangleShape
End Function

Function createCircle (nX As Long, nY As Long,_
                       nRadius As Long) As Object
REM Creates a circle with center
REM at (nX,nY), radius = nRadius

  Dim aPoint As New com.sun.star.awt.Point
  Dim aSize As New com.sun.star.awt.Size
  Dim oCircle As Object

  aPoint.X = nX - nRadius
  aPoint.X = nY - nRadius
  aSize.Width = nRadius * 2
  aSize.Height = nRadius * 2
  oCircle = oDocument.createInstance("com.sun.star.drawing.EllipseShape")
  oCircle.Size = aSize
  oCircle.Position = aPoint
  createCircle = oCircle
End Function
```

4.5.3 Grouping objects

Several shapes can be assembled in one group. This allows to move, scale, rotate etc. a set of shapes in one step. Furthermore, it ensures that shapes belonging together

are manipulated together. The following sample code uses the functions introduced in the last section.

```
Dim oSquare1 As Object, oSquare2 As Object

oSquare1 = createSquare(1000, 1000, 3000)
oSquare1.FillColor = RGB(255,128,128) ' light red
oPage.add(oSquare1)
oSquare2 = createSquare(1000, 7000, 3000)
oSquare2.FillColor = RGB(255,64,64) ' darker red
oPage.add(oSquare2)

Dim oCircle As Object

oCircle = createCircle(2500, 8500, 1500)
oCircle.FillColor = RGB(0,255,0)
oPage.add(oCircle)

Dim oShapes As Object

oShapes = createUnoService("com.sun.star.drawing.ShapeCollection")
oShapes.add(oSquare2)
oShapes.add(oCircle)

Dim oGroup As Object

oGroup = oPage.group(oShapes)

Dim aNewPos As New com.sun.star.awt.Point
Dim nHeight As Long
Dim nWidth As Long

nHeight = oPage.Height
nWidth = oPage.Width
aNewPos.X = nWidth / 2
aNewPos.Y = nHeight / 2
nHeight = oGroup.Size.Height
nWidth = oGroup.Size.Width
aNewPos.X = aNewPos.X - nWidth/2
aNewPos.Y = aNewPos.Y - nHeight/2
oGroup.Position = aNewPos
```

As you can see, you have to create and `add()` shapes before you can group them. The example draws two squares (one filled with a light red and the other one filled with a dark red) and one circle. The circle is drawn on top of the bottom square. These two shapes are then merged into one group in three steps:

1. Create a new shape collection `oGroup`with `createUnoService()`
2. Add existing shapes to this collection with `aGroup.add()`
3. Merge all shapes in the collection with `group()`

After you have created the group, you can no longer change the visual appearance of the shapes it consists of. That means that `oCircle.fillColor=RGB(0,0,255)()` will not change the circle's color to blue after you have added it to the shape collection.

The last lines of the example are just there to prove that the circle and the bottom square do actually form a group: They change the position of the group so that it's moved to the center of the page. This part of the code shows you how to determine the current position and size of a group.

4.5.4 Creating sophisticated shapes

There are many pictures you can draw with rectangles, circles and squares. And there are many more where you need other shapes, for example lines, triangles or curves. We'll show you how to produce some of these shapes here, but the following samples are by no means complete.

4.5.4.1 Creating triangles

Let's start with a simple triangle. This is a special form of what StarOffice API calls "PolyPolygonShape". A PolyPolygonShape is a shape that consists of one or more (hence the first "Poly") polygons, and a polygon is simply a filled area with a fixed number of vertices. In other words, a PolyPolygon is a set of polygons with at least one element. To create a triangle, you'd use something like

```
Dim aPoint As New com.sun.star.awt.Point
Dim aSize As New com.sun.star.awt.Size
Dim oPolyPolygonShape As Object
Dim vPolyPolygon As Variant
Dim mCoordinates(2) As New com.sun.star.awt.Point

oPolyPolygonShape = oDocument.createInstance("com.sun.star.drawing.PolyPolygonShape")
aPoint.X = 5000
aPoint.Y = 3000
aSize.Width = 5000
aSize.Height = 5000
oPolyPolygonShape.Size = aSize
oPolyPolygonShape.Position = aPoint
oPage.add(oPolyPolygonShape)
mCoordinates(0).x = 5000
mCoordinates(1).x = 7500
mCoordinates(2).x = 10000
mCoordinates(0).y = 5000
mCoordinates(1).y = 7500
mCoordinates(2).y = 5000
vPolyPolygon = oPolyPolygonShape.PolyPolygon
vPolyPolygon = Array(mCoordinates())
oPolyPolygonShape.PolyPolygon = vPolyPolygon
```

As before, you create a shape by calling `createInstance()` with the appropriate name of the shape and set its position and size after you have created it. You must than add it to the current page *before* you can specify the points of the polygon. The points are provided in an array which is assigned to the the `PolyPolygon` property of the shape. This property is a sequence of `PointSequence`, which is in turn a

sequence of com.sun.star.awt.Point. In StarBasic, this translates to an array of array of points.

4.5.4.2 Creating figures with holes

All polygons, i.e. figures with three and more corners can be created as shown in the triangle example. A polygon with a hole is simply a sequence of two polygons, as shown below.

```
Dim oPolyPolygonShape As Object
Dim vPolyPolygon As Variant
Dim aPoint As New com.sun.star.awt.Point
Dim aSize As New com.sun.star.awt.Size
Dim aSquare1(3) As New com.sun.star.awt.Point
Dim aSquare2(3) As New com.sun.star.awt.Point

oPolyPolygonShape = oDocument.createInstance("com.sun.star.drawing.PolyPolygonShape")
aPoint.x = 5000
aPoint.y = 3000
aSize.width = 5000
aSize.height = 5000
oPolyPolygonShape.Size = aSize
oPolyPolygonShape.Position = aPoint
oPage.add(oPolyPolygonShape)
aSquare1(0).x = 5000
aSquare1(1).x = 10000
aSquare1(2).x = 10000
aSquare1(3).x = 5000
aSquare1(0).y = 5000
aSquare1(1).y = 5000
aSquare1(2).y = 10000
aSquare1(3).y = 10000
aSquare2(0).x = 6500
aSquare2(1).x = 8500
aSquare2(2).x = 8500
aSquare2(3).x = 6500
aSquare2(0).y = 6500
aSquare2(1).y = 6500
aSquare2(2).y = 8500
aSquare2(3).y = 8500
vPolyPolygon = oPolyPolygonShape.PolyPolygon
vPolyPolygon = Array(aSquare1(),aSquare2())
oPolyPolygonShape.PolyPolygon = vPolyPolygon
```

This example creates a PolyPolygon which consists of two squares. The coordinates of the first square are stored in Square1, those of the second square are stored in Square2. These two arrays are then assembled to an array of arrays which is assigned to oPolyPolygonShape.

You might ask why one would ever want to have PolyPolygons. One possible answer is the picture drawn when you execute the program above. It shows a square that seems to have a square hole in the center. What *appears* to be a hole, though, is in fact the second square. It is drawn as a hole because StarOffice API applies the so-called even-odd rule when filling multiple polygons.

The even-odd rule considers a point to be inside a polygon when any line drawn from it towards infinity crosses the polygon an odd number of times. Conversely, a point is considered to be outside the polygon when the line crosses the polygon an even number of times. Consider the inner square that lies completely inside the first one. If you draw a line from one of its inner points to the outside, it crosses the inner square first and then the outer square. Two intersections are an even number, so the points in the second square are *outside* the polygon. Since the fill color is only applied to the polygon's inside, the second square is not filled and appears as a hole in the first square. The picture below shows this.

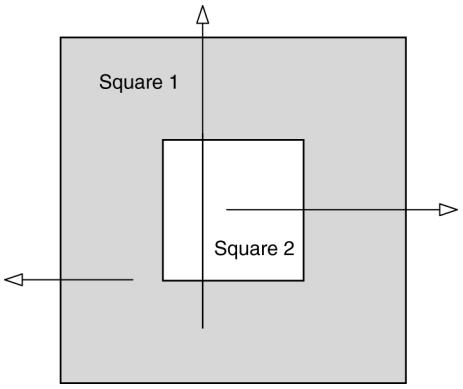

Figure 4–20 Even-odd rule

4.5.4.3 Creating self-intersecting Polygons

A second example will illustrate the even-odd rule. It will display one single polygon that intersects itself, thereby creating a hole.

```
Dim oPolyPolygonShape As Object
Dim vPolyPolygon As Variant
Dim aPoint As New com.sun.star.awt.Point
Dim aSize As New com.sun.star.awt.Size
Dim aSquare1(3) As New com.sun.star.awt.Point
Dim aSquare2(3) As New com.sun.star.awt.Point

oPolyPolygonShape = oDocument.createInstance("com.sun.star.drawing.PolyPolygonShape")
aPoint.x = 5000
aPoint.y = 3000
aSize.width = 5000
aSize.height = 5000
oPolyPolygonShape.Size = aSize
oPolyPolygonShape.Position = aPoint
oPage.add(oPolyPolygonShape)

Dim mStar(4) As New com.sun.star.awt.Point

mStar(0).x = 8000
mStar(0).y = 3000
```

```
mStar(1).x = 10000
mStar(1).y = 8000

mStar(2).x = 4800
mStar(2).y = 4800
mStar(3).x = 11200
mStar(3).y = 4800

mStar(4).x = 5700
mStar(4).y = 8000

vPolyPolygon = oPolyPolygonShape.PolyPolygon
vPolyPolygon = Array(mStar())
oPolyPolygonShape.PolyPolygon = vPolyPolygon
```

Again, a line from any point in the center of the star would have to cross the polygon an even number of times.

To draw lines instead of filled polygons, you'd use the same approach as presented. Instead of `com.sun.star.drawing.PolyPolygonShape` you create an instance of `com.sun.star.drawing.PolyLineShape` or `com.sun.star.drawing.LineShape`. The first one allows you to draw several unconnected line segments in one step while the second one draws a single line segment. To draw a simple line connecting two points, you might want to use a subroutine like this:

```
Function createLine (oDocument,page,x1,y1,x2,y2) As Object
REM Creates a Line from (x1,y1) To (x2,y2)
Dim aPoint As New com.sun.star.awt.Point
Dim aSize As New com.sun.star.awt.Size

aPoint.x = x1
aPoint.y = y1
aSize.Width = x2-x1
aSize.Height = y2-y1
oLine = oDocument.createInstance("com.sun.star.drawing.LineShape")
oLine.Size = aSize
oLine.Position = aPoint
page.add(oLine)

Dim points(1) As New com.sun.star.awt.Point

points(0).x = x1
points(0).y = y1
points(1).x = x2
points(1).y = y2
aLineShape = oLine.PolyPolygon
aLineShape = Array(points())
oLine.PolyPolygon = aLineShape
createLine = oLine
End Function
```

4.5.5 Manipulating shapes

There are several things you may want to do with shapes after you have created them. The changes fall in one of two groups:

- Geometric modifications
- Graphic modifications

Geometric modifications comprise all kinds of transformations like moving (aka translating), rotating, scaling and shearing of objects. Graphic modifications don't change the size, form or position of the object, but aspects of its visual appearance like the fill color. As mentioned before, graphic modifications don't work directly with objects that have been added to a group.

4.5.5.1 Moving and scaling shapes

To move a single shape or a group of shapes, you canset its position to a new value like this:

```
Dim aNewPos As New com.sun.star.awt.Point

aNewPos.x = 10000
aNewPos.y = 15000
oShape.Position = aNewPos
```

Note in StarBasic you cannot simply assign values to `oShape.Position.x` or `oShape.Position.y` to move the shape.

You scale shapes like this:

```
aSize = oShape.Size
aSize.Width = aSize.Width * 0.5
aSize.Height = aSize.Height * 0.5
oShape.Size = aSize
```

This code would halve the size of `aShape`. If you have ever done graphics programming before, you are probably aware of the "fixed point" problem: When you scale an object, its coordinates are multiplied by a constant value. This not only changes its size but also its position, unless you take special caution to avoid this effect. The fixed point of an object is the one point that doesn't move when you scale it. This fixed point is given by the `Position` property.

To scale an object using another fixed point, you could use this subroutine:

```
Sub scale(oShape, nFx, nFy, nScaleX, nScaleY)
  p = oShape.Position
  px = p.X
  py = p.Y
  dx = px - nFx
  dy = py - nFy
  p.X = nFx
  p.Y = nFy
  oShape.Position = p
  aSize = oShape.Size
  aSize.Width = aSize.Width * nScaleX
  aSize.Height = aSize.Height * nScaleY
  oShape.Size = aSize
  p.X = p.X + dx * nScaleX
  p.Y = p.Y + dy * nScaleY
```

```
  oShape.Position = p
End Sub
```

The `scale()` subroutine first moves the object's `Position` so that it coincides with the fixed point `fx/fy`. It then scales the object as described before and finally moves it back to its original position, taking into account the object's new size.

4.5.5.2 Rotating shapes

Rotating objects is similar to moving or scaling them, but StarOffice API takes care of the fixed point for you.

```
Sub rotate(oShape, nFx, nFy, nAngle)
  Dim nAngleI

  nAngleI = nAngle * 100
  oShape.RotationPointX = nFx
  oShape.RotationPointY = nFy
  oShape.RotateAngle = nAngleI
End Sub
```

This subroutine rotates the object `oShape` around the point `nFx/nFy` by `nAngle` degrees counter-clockwise. All it needs to do is to set three properties that determine the rotation angle and fixed point. Since StarOffice API expects the rotation angle in 100ths of a degree, the subroutine scales its last parameter accordingly.

CHAPTER 5

Code Complete

This chapter presents complete StarOffice API programs written in StarBasic. Each of them is designed to solve a single problem or perform a single task. In most cases, the complete program is printed with annotations explaining it. Some trivial code is occasionally left out to save space.

Although the samples are arranged by modules like `Text`, `Data` etc. this grouping is somewhat arbitrary as most of the examples use functions from several modules. We have tried to organize these programs by sections that deal with similar problems.

5.1 Text

The examples in this section concentrate on automated treatment of text. You will see how to change attributes, how to enforce spelling and stylistic rules, and how to generate an index automatically. The two most used interfaces here are `XSearchDescriptor()` and `XTextCursor()`.

5.1.1 Modifying text automatically

If you are writing or editing text documents, some automation can help you to perform your work more efficiently. The next examples will give you some ideas about the kind of automated text processing you can achieve with StarOffice API. They use simple search descriptors and regular expressions and demonstrate how to change styles and text.

The overall structure of these examples is quite similar:

- Create a search or replace descriptor.

- Call the `findAll()` or `replaceAll()` method on this descriptor. We use the method pair `findFirst()/findNext()` occasionally instead of `findAll()`.
- Perform the necessary changes for each text range returned by this method. These changes are modifications of the style, to make the text range stand out or insertion of bookmarks enabling direct navigation to them.

5.1.1.1 Changing appearance

To change the style used for certain words, you can start with the following example.

```
Sub Main
  Dim oDocument As Object
  Dim oSearch As Object, oResult As Object
  Dim oFound As Object, oFoundCursor As Object
  Dim n As Long

  oDocument = ThisComponent
  oSearch = oDocument.createSearchDescriptor
  oSearch.SearchString = "the[a-z]"
  oSearch.SearchRegularExpression = TRUE
  oResult = oDocument.findAll(oSearch)
  For n = 0 To oResult.count - 1
     oFound = oResult(n)
     oFoundCursor = oFound.Text.createTextCursorByRange(oFound)
     oFoundCursor.CharWeight = com.sun.star.awt.FontWeight.BOLD
  Next n
  oSearch.SearchString = "all[a-z]"
  oFound = oDocument.findFirst(oSearch)
  While NOT IsNull(oFound)
     oFoundCursor = oFound.Text.createTextCursorByRange(oFound)
     oFoundCursor.CharPosture = com.sun.star.awt.FontSlant.ITALIC
     oFound = oDocument.findNext(oFound, oSearch)
  Wend
End Sub
```

This code searches for the regular expression `the[a-z]` which stands for the text portion `the` followed by exactly one lowercase letter. All occurrences of these four letters are then changed to be displayed in bold characters. The same happens in the next part of the program, this time changing the appearance of `all[a-z]` to italic. In order for this example to work, you must execute it from an open text document.

5.1.1.2 Replacing text

If you regularly receive documents from other people for editing, you might want to make sure that certain words are always written the same. The next example illustrates this by forcing certain words to be spelled in American English.

```
Sub Main
  Dim mBritishWords(5) As String
  Dim mUSWords(5) As String
```

```
  Dim n As Long
  Dim oDocument As Object
  Dim oReplace As Object

  mBritishWords() = Array("colour", "neighbour", "centre", _
     "behaviour", "metre", "through")
  mUSWords() = Array("color", "neighbor", "center", _
     "behavior", "meter", "thru")
  oDocument = ThisComponent
  oReplace = oDocument.createReplaceDescriptor
  For n = lbound(mBritishWords()) To ubound(mBritishWords())
     oReplace.SearchString = mBritishWords(n)
     oReplace.ReplaceString = mUSWords(n)
     oDocument.replaceAll(oReplace)
  Next n
End Sub
```

In order for this example to work, you must execute it from an open text document. For a real world application, you'd probably want to read the words from an external file.

5.1.1.3 Using regular expressions

Another application of automatic text modification is related to stylistic questions. Suppose your company's policy is to avoid the use of certain words. You want to replace these words, but you can't do that automatically, because you have to find the appropriate replacement which depends on the context. So instead of deleting or replacing the offending words automatically, you change their color to make them stand out during a subsequent manual review process.

The following example handles two kinds of unwanted wordings: those which are absolutely forbidden and must be replaced by something else, and those which are considered bad style. A subroutine is responsible for the changes. It can be used to make all words in a list appear in a certain color in the text document. To keep the lists short, we are using regular expressions here which provide for the variants of the words (plural, adjective etc.).

```
Sub Main
  Dim mOffending(3) As String
  Dim mBad(3) As String
  Dim nOffendColor As Long
  Dim nBadColor As Long

  mOffending() = Array("negro(e|es)?","bor(ed|ing)?", _
    "bloody?", "bleed(ing)?")
  mBad() = Array("possib(le|ilit(y|ies))", _
    "real(ly)+", "brilliant", "\<[a-z]+n\'t\>")
  nOffendColor = RGB(255,0,0)
  nBadColor = RGB(255,128,0)
  colorList(mOffending(), nOffendColor)
  colorList(mBad(), nBadColor)
End Sub
```

```
Sub colorList(mList As Variant, nColor As Long)
  Dim n As Long
  Dim oDocument As Object
  Dim oSearch As Object, oFound As Object
  Dim oFoundCursor As Object

  oDocument = ThisComponent
  oSearch = oDocument.createSearchDescriptor
  oSearch.SearchRegularExpression = TRUE
  For n = LBound(mList()) To UBound(mList())
     oSearch.SearchString = mList(n)
     oFound = oDocument.findFirst(oSearch)
     While NOT IsNull(oFound)
       oFoundCursor = _ oFound.Text.createTextCursorByRange(oFound)
       oFoundCursor.CharColor = nColor
       oFound = oDocument.findNext(oFound,oSearch)
     Wend
  Next n
End Sub
```

Here, we use two lists of regular expressions `mOffending` and `mBad`. Those words that match one of the expressions in `mOffending` are marked red in the text, because we have to get rid of them. Those in `mBad` are marked orange, as they are considered bad style only. The regular expressions here are not overly complex, but two of them might merit a closer look: `possib(le|ilit(y|ies))` matches all occurrences of `possible`, `possibility`, and `possibilities`. It looks for `possib` followed by either `le` or by `ilit(y|ies)`, which is `ility` or `ilities`. The other interesting regular expression is `\<[a-z]+n\'t\>`. It finds all abbreviated negations like `don't`, `can't` etc. by looking for the start of a word (`\<`), at least one lowercase letter (`[a-z]+`)) followed by `n't` at the end of the word (`\>`).

Most of the work is done by the subroutine `colorList()`. It creates a search descriptor `oSearch` and sets its `SearchRegularExpression` property to `TRUE`. It then loops over all strings in `mList`, changing the search descriptor's `SearchString` and finding all occurrences of them in the document. The color of these words is then set to `nColor`.

5.1.1.4 Inserting bookmarks

The next example does something very similar. This time, however, we do not change the color of the words but insert a bookmark at each of them. You can thus use the StarOffice navigator to jump directly from word to word. Bookmarks have first to be created using `createInstance()`. They are then inserted with `insertTextContent()` at the current text range.

```
Sub Main
  Dim mOffending(3) As String
  Dim mBad(3) As String
  Dim sOffendPrefix As String
  Dim sBadPrefix As String

  mOffending() = Array("negro(e|es)?","bor(ed|ing)?", _
    "bloody?", "bleed(ing)?")
```

```
  mBad() = Array("possib(le|ilit(y|ies))",  _
    "real(ly)", "brilliant", "\<[a-z]+n\'t\>")
  sOffendPrefix = "Offending"
  sBadPrefix = "BadStyle"
  markList(mOffending(), sOffendPrefix)
  markList(mBad(), sBadPrefix)
End Sub

Sub markList(mList As Variant, sPrefix As String)
  Dim n As Long, nCount As Long
  Dim oDocument As Object
  Dim oSearch As Object, oFound As Object
  Dim oFoundCursor As Object
  Dim oBookmark As Object

  oDocument = ThisComponent
  oSearch = oDocument.createSearchDescriptor
  oSearch.SearchRegularExpression = TRUE
  nCount=0
  For n = LBound(mList()) To UBound(mList())
    oSearch.SearchString = mList(n)
    oFound = oDocument.findFirst(oSearch)
    While NOT IsNull(oFound)
       nCount=nCount+1
       oFoundCursor = _
       oFound.Text.createTextCursorByRange(oFound)
       oBookmark = _
       oDocument.createInstance("com.sun.star.text.Bookmark")
       oBookmark.Name=sPrefix + CStr(nCount)
       oDocument.Text.insertTextContent(oFoundCursor,_
          oBookmark,TRUE)
       oFound = oDocument.findNext(oFound, oSearch)
    Wend
  Next n
End Sub
```

The main difference to the preceding example is the `For` loop in `markList()`. Instead of changing the color of the current word, it creates a new bookmark, `oBookmark`, whose name is the current word with an integer appended. It then inserts this bookmark at the word.

5.1.2 Creating an index

Indices for text documents can be created manually in StarWriter by clicking on the words that shall appear in the index. If the document is large or if you have to generate indices for several documents, this task should be automated. The following program consists of three main parts:

- It first opens an external file containing the words to appear in the index.
- It loops over the file content and marks every occurrence of each word in the text document using a text content of type `documentIndexMark`. We use a search descriptor here again.
- When the file is exhausted, the program appends the index to the document.

```
REM sUrl URL to indexlist.txt
Sub Main
  ApplyIndexList(sUrl, ThisComponent)
End Sub
Sub ApplyIndexList(sIndexFilePath As String, oDocument As Object)
  Dim oText As Object
  Dim nFileNumber As Long
  Dim sLineText As String

  oText = oDocument.Text
  nFileNumber = FreeFile
  Open sIndexFilePath For Input As #nFileNumber
  While Not eof(nFileNumber)
     Line Input #nFileNumber, sLineText
     If sLineText <> "" Then
       SearchWordInTextAndInsertIndexEntry(oDocument, sLineText)
     End If
  Wend
  Close #nFileNumber
  InsertDocumentIndex(oDocument)
End Sub
```

The main part of the program opens the external file `sIndexFilePath` and reads it line by line. For the sake of simplicity, each line is supposed to contain exactly one of the index terms. The words are then passed to `SearchWordInTextAndInsertIndexEntry()`. When the file is exhausted, the subroutine `InsertDocumentIndex()` is called to add the index to the document.

```
Sub SearchWordInTextAndInsertIndexEntry(oDocument As Object,_
    sEntry As String)
  oSearch = oDocument.createSearchDescriptor
  oSearch.SearchString = sEntry
  oFound = oDocument.findAll(oSearch)
  For n = 0 To oFound.Count - 1
     oFoundPos = oFound(n)
     oIndexEntry = _
       oDocument.createInstance("com.sun.star.text.DocumentIndexMark")
     oFoundPos.text.insertTextContent(oFoundPos, oIndexEntry, TRUE)
  Next n
End Sub
```

The subroutine `SearchWordInTextAndInsertIndexEntry()` creates the search descriptor `oSearch` and sets its `SearchString` property to the word to look for (`sEntry`). It then loops over all occurrences of this word and inserts the special text content `DocumentIndexMark` at this position.

```
Sub InsertDocumentIndex(oDocument As Object)
 oText = oDocument.Text
 oCursor = oText.createTextCursor
 oCursor.GotoEnd(FALSE)
 oText.insertControlCharacter(oCursor,_
     com.sun.star.text.ControlCharacter.PARAGRAPH_BREAK, FALSE)
 oText.insertControlCharacter(oCursor,_
     com.sun.star.text.ControlCharacter.PARAGRAPH_BREAK, FALSE)
 oCursor.goLeft(1, FALSE)
 oIndex = oDocument.createInstance("com.sun.star.text.DocumentIndex")
 oIndex.UseCombinedEntries = TRUE
```

```
 oIndex.Title = "Index"
 oIndex.UsePP = TRUE
 oText.insertTextContent(oCursor, oIndex, FALSE)
 oIndex.update()
End Sub
```

Finally, the subroutine `InsertDocumentIndex()` inserts the complete index. It creates a text cursor (`oCursor`) and uses it to move to the end of the text, where it adds two paragraph breaks. The index (`oIndex`) is then created with `createInstance()`. Its title is set to `Index`, and the style is defined so that for each entry only the first page number is shown, possibly with an appended `ff` if the entry occurs on other pages as well. `UsePP` specifies the format for the page numbers. If it is set to `FALSE`, the first and last page would be printed instead of only the first one. If `CombineEntries` is `FALSE`, all occurrences of each entry is listed.

5.2 Sheet

5.2.1 Adapting to Euroland

Most of the member of the European Union will abandon their old currency in favor of the new Euro in 2001. This requires modifications to all programs using the old currencies. Since the exchange rates for the old currencies have been fixed at the end of 1999, one can already convert old data. The following program does this for all values in a table that are formatted with the currency string `DM`. Three steps are necessary for this modification:

- First, it finds all cells using a number format of type `CURRENCY` and containing `DM` as currency string. We use an enumeration to loop over all cells here.
- A new number format is then defined for these cells and applied to them.
- Finally, the cell's values are corrected to show the new amount in Euro.

```
Sub Main
  Dim oDocument As Object

  oDocument = ThisComponent
  oDocument.addActionLock
  Convert( oDocument.Sheets(0), oDocument.NumberFormats, "DM", _
    "EUR", 1.95583 )
  oDocument.removeActionLock
End Sub
```

The main program just calls the subroutine `Convert()` which does nearly all the work. It receives the sheet, the `NumberFormats` interface, the names of the old and the new currency, and the conversion factor as parameters. The call is embedded in `addActionLock()` and `removeActionLock()` calls, which delays update of the

table view until Convert() has terminated. This speeds up the function considerably, since the table is only redrawn once.

```
Sub Convert( oSheet As Object, oFormats As Object,_
             sOldSymbol As String, sNewSymbol As String,_
             fFaktor As Double )
  Dim nSimpleKey As Long
  Dim aLanguage As New com.sun.star.lang.Locale
  Dim sSimple As String
  Dim oFormat As Object
  Dim oRanges As Object, oRange As Object
  Dim n As Long
  Dim sNew As String
  Dim oValues As Object
  Dim oCells As Object, oCell As Object

  aLanguage.Country = "de"
  aLanguage.Language = "de"
  sSimple = "0 [$"+sNewSymbol+"]"
  nSimpleKey = NumberFormat( oFormats, sSimple, aLanguage )
  oRanges = oSheet.CellFormatRanges.createEnumeration
```

The first few lines of Convert() determine the key of a simple format using the new currency symbol (nSimpleKey) and the function NumberFormat() (see below). Then a collection of ranges is retrieved from the sheet. Every element in this collection is a range whose cells are using the same format. Convert() will now iterate over these ranges.

```
    While oRanges.hasMoreElements
      oRange = oRanges.nextElement
      oFormat = oFormats.getByKey( oRange.NumberFormat )
```

Every range (oRange) is characterized by the fact that all its cells use the same format. Its properties are retrieved here with the format's getByKey() method.

```
       If ( oFormat.Type AND com.sun.star.util.NumberFormat.CURRENCY )_
              And ( oFormat.CurrencySymbol = sOldSymbol ) Then
```

The line above will be executed only if the current format is of a type that encompasses currency formats and if the currency string is identical to the old one (sOldSymbol). The condition in the if statement first calculates the integer AND of oFormat.Type and com.sun.star.util.NumberFormat.CURRENCY. This expression yields TRUE only if CURRENCY is a part of oFormat.Type. You cannot simply check for equality here because the number formats in StarOffice API can be combinations of several base formats.

```
  If ( oFormat.Type AND com.sun.star.util.NumberFormat.CURRENCY )_
           And ( oFormat.CurrencySymbol = sOldSymbol ) Then
    sNew = oFormats.generateFormat( nSimpleKey, oFormat.Locale, _
                                    oFormat.ThousandsSeparator, _
                                    oFormat.NegativeRed, _
                                    oFormat.Decimals, oFormat.LeadingZeros )
    oRange.NumberFormat = NumberFormat( oFormats, sNew, oFormat.Locale )
    oValues = oRange.queryContentCells( com.sun.star.sheet.CellFlags.VALUE )
    If oValues.Count > 0 Then
```

```
      oCells = oValues.Cells.createEnumeration
      While oCells.hasMoreElements
        oCell = oCells.nextElement
        oCell.Value = oCell.Value / fFaktor
      Wend
    End If
  End If
Wend
End Sub
```

The function `generateFormat()` create a new format string based on the simple format (`nSimpleKey`) generated at the very beginning of the subroutine. All we really do here is to change the currency's name, the other properties of the current format remain untouched. This string is passed to the function `NumberFormat()` which returns a suitable format key. The key is then assigned to the `NumberFormat` property of the current range, which effectively sets its format to use the new currency name. Finally, we loop over all cells in the current range which contain a constant value and adjust this value according to the currency exchange rate.

```
Function NumberFormat( oFormats As Object,_
                       sFormat As String , aLanguage As Variant ) As Long
   Dim nRetKey As Long

   nRetKey = oFormats.queryKey( sFormat, aLanguage, true )
     If nRetKey = -1 Then
        nRetKey = oFormats.addNew( sFormat, aLanguage )
        If nRetKey = -1 Then nRetKey = 0
     End If
     NumberFormat = nRetKey
End Function
```

The function `NumberFormat()` searches for a given format string in the list of formats using `queryKey()`. This method returns either the key of the existing format string or -1 if the string didn't exist previously. In this case, the function creates a new format using `addNew()`. In any case, the key of the format string is returned.

5.3 Drawing

The following example converts a textual form of a directory tree into a graphics.

5.3.1 Import/Export of ASCII files

You can use the drawing facilities of StarOffice API to generate a picture from ASCII input. One application would be a hierarchical representation of a directory listing, based on textual data which looks like this:

```
Explorer
   Workspace
      Bookmarks
      Bugtracker
      Databases
      Macros
      Starmath
      StarOffice API
      ToDo
      Translations
       Chinese
       German
       English
       French
       Italian
       Dutch
       Portuguese
       Swedish
       Spanish
   Examples
      Databases
         Files
```

This is an extract from a directory listing. Every directory is indented three spaces from the preceding one. The complete path to the file `Chinese` is therefore `Explorer/Workspace/Translations/Chinese`. This data would be displayed as shown in the picture below. Every file and directory name is written in a text box. Directories on the same level are connected by a vertical line. Files and subdirectories in a directory are connected by a horizontal line to this vertical line.

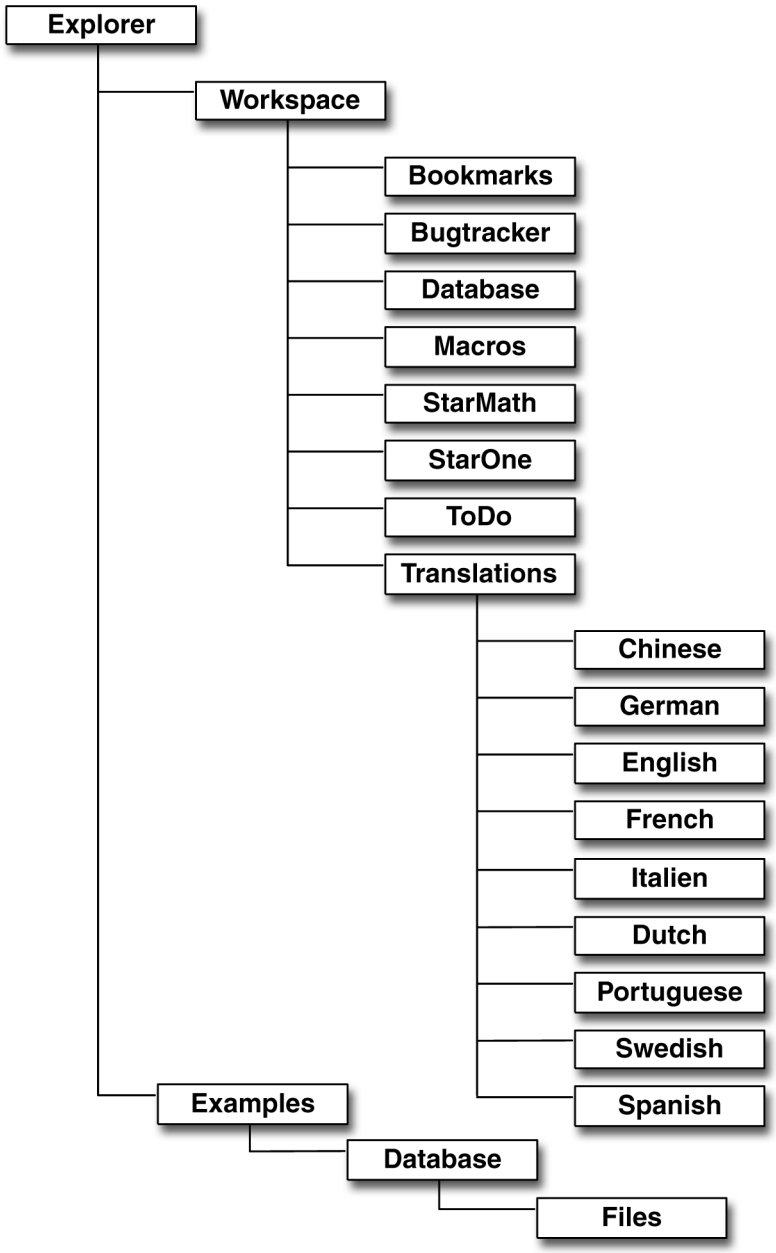

Figure 5–1 Directory tree

The following application reads such a file and produces a graphical representation of its contents. It performs several steps:

- Open the data file and read one line at a time.

- Determine the indentation of this line relative to the previous one.
- Create a graphics text from the line and calculate its dimensions.
- Add a new page if the text doesn't fit on the current one.
- Draw the lines connecting the current item to the one on on the previous level.

5.3.1.1 Setting up

Before the real code starts, we define some constants that permit us to write down the rest of the program more clearly. Since they represent values used in many places in the code it's reasonable to define them only once thus allowing for easier modification.

```
Const SBBASEWIDTH = 8000
Const SBBASEHEIGHT = 1000
Const SBPAGEX = 800
Const SBPAGEY = 800
Const SBBASECHARHEIGHT = 12
Const SBRELDIST = 1.1
```

The constants above are used throughout the whole program. The following table lists them together with their meaning.

Name	Meaning
SBASEWIDTH	Original width of the text fields in 100th mm
SBASEHEIGHT	Original height of the text fields in 100th mm
SBPAGEX	Left page margin in 100th mm
SBPAGEY	Top/bottom page margin in 100th mm
SBBASECHARHEIGHT	Character height in points
SBRELDIST	Factor to increase spacing between entries

```
Const SBBASEX = 0
Const SBBASEY = 1
Const SBOLDSTARTX = 2
Const SBOLDSTARTY = 3
Const SBOLDENDX = 4
Const SBOLDENDY = 5
Const SBNEWSTARTX = 6
Const SBNEWSTARTY = 7
Const SBNEWENDX = 8
Const SBNEWENDY = 9
Dim nActLevel as Integer
Dim nConnectLevel as Integer
Dim mLevelPos(10,9) As Integer
```

The preceding constants are used to specify the second index of the two-dimensional array `mLevelPos` which records positions for every level. Their meaning is listed in the next table.

Name	Meaning
SBBASEX	X coordinate of the starting point for the level.
SBBASEY	Y coordinate of the last text drawn in this level.
SBOLDSTARTX	X start coordinate for vertical connecting lines.
SBOLDSTARTY	Y start coordinate for vertical connecting lines.
SBOLDENDX	X end coordinate for vertical connecting lines.
SBOLDENDY	Y end coordinate for vertical connecting lines.
SBNEWSTARTX	X start coordinate for next vertical connecting line.
SBNEWSTARTY	Y start coordinate for next vertical connecting lines.
SBNEWENDX	X end coordinate for next vertical connecting line.
SBNEWENDY	Y end coordinate for next vertical connecting line.

5.3.1.2 Initialization

We start by initializing some global variables and acquiring the desktop service. We then request the `drawPage` of the current document and start reading the file containing the textual representation of the tree.

```
Sub TreeInfo(sFilePath As String)
  REM Variable declarations ommitted
  bStartUpRun  = TRUE
  nOldHeight = 200
  nOldY = SBPAGEY
  nOldX = SBPAGEX
  nOldWidth = SBPAGEX
  nActPage = 0
  oDocument = ThisComponent
  oPage = oDocument.drawPages(nActPage)
  nFileHandle = FreeFile
  Open sFilePath For Input As nFileHandle
```

The preceding lines open the file `sFilePath` for input. This filename is the only parameter of the subroutine. Note that we have omitted variable declarations for the sake of brevity.

```
While Not Eof(nFileHandle)
 Line Input #nFileHandle, sActDir
 nBaseLength = Len(sActDir)
 nModLength = Len(LTrim(sActDir))
 sActDir = LTrim(sActDir)
 If nBaseLength - nModLength > 0 Then
   nActLevel = (nBaseLength - nModLength) / 3
   nConnectLevel = nActLevel -1
 Else
   nActLevel = 0
   nConnectLevel = 0
 End If
```

The `While` loop iterates over all lines of the input file. It reads the next file or directory name into the variable `sActDir` and figures out the indentation level using `nBaseLength` and `nModLength`. The first variable is the complete length of the line, possibly including leading spaces. The second one is just the name of the file or directory with leading spaces removed. The difference of these two variables divided by three gives the indentation level for the current entry, stored in `nActLevel`. The variable `nConnectLevel` indicates the number of the preceding level - it is thus `nActLevel-1`.

5.3.1.3 Drawing an item

Before we can actually draw the current entry, we have to make sure that it fits completely on the current page. If it does not, we'll add a new draw page and reset some of the global variables. The text is then drawn and the current positions are updated.

```
  If nOldY + (nOldHeight + SBBASECHARHEIGHT)* 1.5 > _
oPage.Height - SBPAGEY Then
   nActPage = nActPage + 1
   oDocument.getDrawPages.InsertNewbyIndex(nActPage)
   Set oOldPage = oPage
   oPage = oDocument.drawPages(nActPage)
   For n = 0 To nConnectLevel
   mLevelPos(n,SBNEWENDY) = oOldPage.Height - SBPAGEY
   oOldLeavingLine = DrawLine(n, SBNEWSTARTX, SBNEWSTARTY,_
         SBNEWSTARTXX, SBNEWENDY)
   oOldPage.Add(oOldLeavingLine)
  Next
  For n = 0 To nConnectLevel
  mLevelPos(n,SBNEWSTARTY) = SBPAGEY
  Next
  nOldY = SBPAGEY
End If
```

This chunk of code takes care of page breaks. If the next entry would overflow the current draw page, `nActPage` is incremented and a new draw page inserted in the

document by calling `InsertNewbyIndex()`. The condition to determine possible overflow is based on some heuristics, since we can't be sure about the size of the bounding box *before* having drawn it. The program then draws the vertical connecting lines to the next page. To do so, we have to update the `SBNEWENDY` element of `mLevelPos` so that it contains the lowest y coordinate of a page. The function `DrawLine()` then creates a line from `mLevelPos(n,SBNEWSTARTX)/ mLevelPos(n,SBNEWSTARTY)` to `mLevelPos(n,SBNEWSTARTX)/ mLevelPos(n,SBNEWENDY)` for every level from 0 to `nConnectLevel`. Finally the `mLevelPos(n,SBNEWSTARTY)` entries and `nOldY` are set to the top margin of the page.

```
oActText = CreateText()
oPage.add(oActText)
oActText.LineStyle = 1
oActText.Charheight = SBBASECHARHEIGHT
oActText.TextAutoGrowWidth = TRUE
oActText.TextAutoGrowHeight = TRUE
oActText.String = sActDir
aPoint.x = mLevelPos(nActLevel,SBBASEX)
oActText.Position = aPoint
aSize.Height = SBRELDIST * oActText.CharHeight
aSize.Width = SBRELDIST * oActText.Size.Width
oActText.Size = aSize
mLevelPos(nActLevel,SBBASEY) = oActText.Position.Y
```

These lines add the entry for the current text using the function `CreateText()`. The shape created by it is added to the current draw page and some properties are set to ensure that the surrounding box grows with the text. To increase the margin of the text, the default box dimensions are increased by multiplying them with `SBRELDIST`.

```
If bStartUpRun = FALSE Then
  If nActLevel <> 0 Then
  mLevelPos(nActLevel,SBOLDSTARTX) =_
    mLevelPos(nConnectLevel,SBNEWSTARTX)
  mLevelPos(nActLevel,SBOLDSTARTY) =_
    oActText.Position.Y + 0.5 * oActText.Size.Height
  mLevelPos(nActLevel,SBOLDENDX) =_
    mLevelPos(nActLevel,SBBASEX)
  mLevelPos(nActLevel,SBOLDENDY) =_
    mLevelPos(nActLevel,SBOLDSTARTY)
  oOldArrivingLine = DrawLine(nActLevel, SBOLDSTARTX,_
        SBOLDSTARTY, SBOLDENDX,_
        SBOLDENDY)
  oPage.Add(oOldArrivingLine)
  mLevelPos(nConnectLevel,SBNEWENDX) =_
    mLevelPos(nConnectLevel,SBNEWSTARTX)
  mLevelPos(nConnectLevel,SBNEWENDY) =_
    oActText.Position.Y + 0.5 * oActText.Size.Height
  Else
  mLevelPos(nConnectLevel,SBNEWENDY) = oActText.Position.Y
  mLevelPos(nConnectLevel,SBNEWENDX) =_
      mLevelPos(nConnectLevel,SBNEWSTARTX)
  End If
  oOldLeavingLine = DrawLine(nConnectLevel, SBNEWSTARTX,_
        SBNEWSTARTY, SBNEWENDX,_
        SBNEWENDY)
```

```
    oPage.Add(oOldLeavingLine)
  Else
    bStartUpRun = FALSE
  End If
```

5.3.1.4 Connecting to the including item

Now we have to connect the current entry with the vertical line descending from the current directory. If the current level is 0 (as indicated by `nActLevel`)), we simply set the `SBNEWENDX` and `SBNEWENDY` parts in `mLevelPos` to the `SBNEWSTARTX` entry and the current text position (`oActText.Position.Y`), respectively. This sets up `mLevelPos` so that the next `DrawLine()` call will create a vertical line (`oOldLeavingLine`). If the current level is not 0, we have to add a horizontal line connecting the text box to the descending line from the previous level's vertical line. This is achieved in the first part of the second `If` statement above by creating `oOldArrivingLine`.

Horizontal and/or vertical lines are only drawn if `bStartUpRun` is `FALSE`, which is the case for every entry but the very first one. If `bStartUpRun` is `TRUE`, it is simply set to `FALSE`.

```
    mLevelPos(nActLevel,SBNEWSTARTX) =_
     mLevelPos(nActLevel,SBBASEX) + 0.5 * oActText.Size.Width
     mLevelPos(nActLevel,SBNEWSTARTY) =_
          mLevelPos(nActLevel,SBBASEY) + oActText.Size.Height
     nOldHeight = oActText.Size.Height
     nOldX = oActText.Position.X
     nOldWidth = oActText.Size.Width
     nOldLevel = nActLevel
  Wend
  Close #nFileHandle
Exit Sub
```

The rest of the main part updates two entries in `mLevelPos` and a couple of global variables to reflect the positions of the text and line(s) just drawn. The first two assignments to `mLevelPos` update the start coordinates of the line leaving from the current entry.

5.3.1.5 Utility functions

Some functions take care of repetitive tasks like drawing the actual text and calculating the current X position.

```
Function CreateText()
  Dim oText As Object

  aSize.Width = SBBASEWIDTH
  aSize.Height = SBBASEHEIGHT
  aPoint.x = CalculateXPoint()
  aPoint.y = nOldY + SBRELDIST * nOldHeight
  nOldY  = aPoint.y
  oText = oDocument.createInstance("com.sun.star.drawing.TextShape")
  oText.Size = aSize
```

```
  oText.position = aPoint
  CreateText = oText
End Function
```

`CreateText()` creates a text shape of size `SBBASEWIDTH` by `SBBASEHEIGHT` and inserts it at the x coordinate returned by `CalculateXPoint()`. The y coordinate of the text position is calculated by adding a bit more than the last text height to the last y coordinate (stored in `nOldY`). The text shape is then returned.

```
Function CalculateXPoint()
  If nActLevel = 0 Then
     mLevelPos(nActLevel,SBBASEX) = SBPAGEX
  Elseif nActLevel > nOldLevel Then
     mLevelPos(nActLevel,SBBASEX) =_
     mLevelPos(nActLevel-1,SBBASEX) + nOldWidth + 100
  End If
  CalculateXPoint = mLevelPos(nActLevel,SBBASEX)
End Function
```

`CalculateXPoint()` figures out the x coordinate of the current text box and stores it in `mLevelPos(nActLevel, SBBASEX)`. If the current level is 0, this is simply the left page margin. If the current entry is the first one in its level (`nActLevel > nOldlevel`) the x position is that of the last level incremented by the value of `nOldWidth` plus a small margin of 1 millimeter, thereby increasing the indentation.

If the current level is less then the previous one, there is nothing to be done, because the x values are already calculated

```
Function DrawLine(nLevel As Integer,_
      nStartX As Integer, nStartY As Integer,_
      nEndX As Integer, nEndY As Integer)
  Dim oConnect As Object

  aPoint.X = mLevelPos(nLevel,nStartX)
  aPoint.Y = mLevelPos(nLevel,nStartY)
  aSize.Width = mLevelPos(nLevel,nEndX) - mLevelPos(nLevel,nStartX)
  aSize.Height = mLevelPos(nLevel,nEndY) - mLevelPos(nLevel,nStartY)
  oConnect = oDocument.createInstance("com.sun.star.drawing.LineShape")
  oConnect.Position = aPoint
  oConnect.Size = aSize
  DrawLine = oConnect
End Function
```

`DrawLine()` creates a line shape. It uses the `nLevel` and the `nStartX/nStartY`, `nEndX/nEndY` pairs to retrieve the coordinates of the line from `mLevelPos`. The newly created line shape is then returned to the caller.

5.4 Stock quotes updater

If you want to display stock charts for certain companies, you can fire up your browser every day, go to Yahoo, look up the quote and copy it by hand into a table. Or you can use a program that does all this automatically.

The following example relies on the sheet module. It uses URLs to obtain the current stock quotes. The quotes are displayed in sheets, one for each company. We show a line diagram and the numerical values for this company on every sheet. The functionality is hidden in the three subroutines `GetValue()`, `UpdateValue()`, and `UpdateChart()`.

```
Option Explicit
Const STOCK_COLUMN = 2
Const STOCK_ROW = 6
Sub UpdateAll
  Dim sName As String
  Dim oCells As Object
  Dim fDate As Double, fValue As Double
  Dim oDocument As Object, oInputSheet As Object
  Dim oColumn As Object, oRanges As Object

  oDocument = ThisComponent
  oDocument.addActionLock
  oInputSheet = oDocument.Sheets(0)
  oColumn = oInputSheet.Columns(STOCK_COLUMN)
  oRanges = _
   oDocument.createInstance("com.sun.star.sheet.SheetCellRanges")
  oRanges.insertByName("", oColumn)
  oCells = oRanges.Cells.createEnumeration
  oCells.nextElement
  While oCells.hasMoreElements
     sName = oCells.nextElement.String
     If GetValue( oDocument, sName, fDate, fValue ) Then
       UpdateValue( oDocument, sName, fDate, fValue )
       UpdateChart( oDocument, sName )
     Else
       MsgBox sName + " Is Not available"
     End If
  Wend
  oDocument.removeActionLock
End Sub
```

This macro should be run from a table document that contains the NASDAQ names of the companies to look for in the column `STOCK_COLUMN`. The sample document provided on the CD-ROM already contains a table and two buttons you can use to insert new columns or to update the stock quotes. The program creates an unnamed range for this column and then steps through all cells until it finds an emtpy one using `nextElement()`. The first cell is skipped because it contains no valid NASDAQ name but just a label for the column. The function `GetValue()` is then used to retrieve the current quote. If the value can't be found, a message is

displayed. Otherwise, subroutines `UpdateValue()` and `UpdateChart()` are used to update the value and chart for this company.

5.4.1 Retrieving URLs

We use Yahoo here to obtain the current stock quote. This requires retrieving a nURL via StarOffice API and retrieving a part of its contents.

```
Function GetValue( oDocument As Object,_
       sName As String,_
       fDate As Double, fValue As Double )
  Dim sUrl As String, sFilter As String
  Dim sOptions As String
  Dim oSheets As Object, oSheet As Object

  oSheets = oDocument.Sheets
  If oSheets.hasByName("Link") Then
    oSheet = oSheets.getByName("Link")
  Else
    oSheet = _
    oDocument.createInstance("com.sun.star.sheet.Spreadsheet")
    oSheets.insertByName("Link", oSheet)
    oSheet.IsVisible = False
  End If
  sUrl = "http://quote.yahoo.com/d/quotes.csv?s=" +_
  sName + "&f=sl1d1t1c1ohgv&e=.csv"
  sFilter = "Text - txt - csv (StarCalc)"
  sOptions = _
    "44,34,SYSTEM,1,1/10/2/10/3/10/4/10/5/10/6/10/7/10/8/10/9/10"
  oSheet.link(sUrl, "", sFilter, sOptions, _
     com.sun.star.sheet.SheetLinkMode.NORMAL )
  fDate = oSheet.getCellByPosition(2,0).Value
  fValue = oSheet.getCellByPosition(1,0).Value
  GetValue = (fDate <> 0)
End Function
```

The function `GetValue()` first checks if a table with the name `Link` exists and creates one if it doesn't. Since this table is just used to store temporary results, we turn its visibility off. The next line builds the URL needed to retrieve the quotes from Yahoo and stores it in `sUrl`. Since we retrieve the data in CSV format, we have to specify this in `sFilter` and provide the correct options so that StarOffice API can parse the returned data. For more information on the CSV options, see Section 4.2.1 "Importing other Formats" on page 41. URL, filter and options are then passed to the sheet's `link()` method which causes StarOffice API to retrieve the data and store it in the sheet. Afterwards, the third cell in the first row contains the current date and the second cell the quote for this date. These values are returned in `fDate` and `fValue`, respectively. The function itself returns `TRUE` if it could retrieve the quote and false otherwise.

5.4.2 Updating the tables

When the current value has been determined, we have to update the table belonging to that value. The following subroutine takes care of this.

```
Sub UpdateValue( oDocument As Object, sName As String,_
     fDate As Double, fValue As Double )
  Dim oSheets As Object, oSheet As Object
  Dim nColumn As Long, nRow As Long
  Dim oCursor As Object, oRanges As Object
  Dim aAddress As Variant
  Dim oCells As Object, oCell As Object
  Dim aLanguage as New com.sun.star.lang.Locale

  aLanguage.Country = "de"
  aLanguage.Language = "de"
  oSheets = oDocument.Sheets
  sName = SheetNameFor(sName)
  If oSheets.hasByName(sName) Then
    oSheet = oSheets.getByName(sName)
  Else
    oSheet =_
    oDocument.createInstance("com.sun.star.sheet.Spreadsheet")
    oSheets.insertByName(sName, oSheet)
  End If
  oRanges =_
    oDocument.createInstance("com.sun.star.sheet.SheetCellRanges")
  oRanges.insertByName("", oSheet)
  oCells = oRanges.Cells
  If oCells.hasElements Then
     oCell = oCells.createEnumeration.nextElement
     oCursor = oSheet.createCursorByRange(oCell)
     oCursor.collapseToCurrentRegion
     aAddress = oCursor.RangeAddress
     nColumn = aAddress.StartColumn
     nRow = aAddress.EndRow
     If oSheet.getCellByPosition(nColumn,nRow).Value <> fDate Then
       nRow = nRow + 1
     End If
  Else
     oSheet.getCellByPosition(1,18).String = "Date"
     oSheet.getCellByPosition(2,18).String = sName
     oSheet.getCellRangeByPosition(1,18,2,18).CharWeight = 150
     nColumn = 1
     nRow = 19
  End If
  oCell = oSheet.getCellByPosition(nColumn,nRow)
  oCell.Value = fDate
  oCell.NumberFormat =_
  oDocument.NumberFormats.getStandardFormat(_
    com.sun.star.util.NumberFormat.DATE, aLanguage )
  oSheet.getCellByPosition(nColumn+1,nRow).Value = fValue
End Sub
```

Subroutine `UpdateValue()` receives the NASDAQ name (`sName`), the current date (`fDate`) and the quote (`fValue`) as parameters. First, it verifies if a sheet named `sName` exists and if not, creates it. The subroutine then creates a cell range for this sheet. If the program has been run before, this range contains non-emtpy cells, and

the `if` branch finds the last of these cells. If this cell's value is different from `fDate`, the variable `nRow` is incremented by one. This ensures that a later quote for the same day overwrites an earlier one and that a quote for a later date is inserted in the next row.

If this is the first time that a qoute for this company is retrieved, the cell range is empty and we simply insert the strings `Date` and `sName` in cells B19 and C19, respectively. The variables `nColumn` and `nRow` are then set to 1 and 19, which addresses cell B20.

Finally, `updateValue()` inserts the values `fDate` and `fValue` in the cells at `nColumn` and `nColumn+1` in row `nRow`. The update for this company's sheet is thus complete.

5.4.3 Updating the chart

Finally we must update or create the line diagram. The following function `UpdateChart()` first checks if a diagram exists already and possibly creates it. It then adjusts the minimum and maximum values of its x and y axis according to the values in the sheet making sure that no values are cut off.

```
Sub UpdateChart( oDocument As Object, sName As String)
  Dim oSheet As Object
  Dim oCell As Object, oCursor As Object
  Dim oRanges As Object
  Dim oDataRanges As Object, oChartRange As Object
  Dim oEmbeddedChart As Object, oCharts As Object
  Dim oChart As Object, oDiagram As Object
  Dim oYAxis As Object, oXAxis As Object
  Dim fMin As Double, fMax As Double
  Dim nDateFormat As Long
  Dim aPos As Variant
  Dim aSize As Variant
  Dim aLanguage as New com.sun.star.lang.Locale

  aLanguage.Country = "de"
  aLanguage.Language = "de"
  oSheet = oDocument.Sheets.getByName( SheetNameFor(sName) )
  oCharts = oSheet.Charts
  oRanges = oDocument.createInstance("com.sun.star.sheet.SheetCellRanges")
  oRanges.insertByName("", oSheet)
  oCell = oRanges.Cells.createEnumeration.nextElement
  oCursor = oSheet.createCursorByRange(oCell)
  oCursor.collapseToCurrentRegion
  oDataRanges = oDocument.createInstance("com.sun.star.sheet.SheetCellRanges")
  oDataRanges.insertByName("", oCursor)
```

Subroutine `UpdateChart()` takes care of displaying and updating the stock quotes chart for one company. It begins with the creation of cursor which points to the first of the cells showing the dates and quotes. The cursor is then collapsed to this region, thereby spanning a rectangle which contains only the date and quote values. Using

createInstance() and insertByName(), the cursor's rectangular region is then used to define a range.

```
If Not oCharts.hasElements Then
  oChartRange = oSheet.getCellRangeByPosition(1,1,7,16)
  aPos = oChartRange.Position
  aSize = oChartRange.Size

  Dim oRectangleShape As New com.sun.star.awt.Rectangle

  oRectangleShape.X = aPos.X
  oRectangleShape.Y = aPos.Y
  oRectangleShape.Width = aSize.Width
  oRectangleShape.Height = aSize.Height
  oCharts.addNewByName( sName, oRectangleShape,_
  oDataRanges.RangeAddresses, true, false )
  oChart = oCharts.getByName(sName).EmbeddedObject
  oChart.diagram = oChart.createInstance("com.sun.star.chart.XYDiagram")
  oChart.Title.String = sName
  oDiagram = oChart.Diagram
  oDiagram.DataRowSource = com.sun.star.chart.ChartDataRowSource.COLUMNS
  oChart.Area.LineStyle = com.sun.star.drawing.LineStyle.SOLID
  oXAxis = oDiagram.XAxis
  oXAxis.StepMain = 1
  oXAxis.StepHelp = 1
  oXAxis.AutoStepMain = false
  oXAxis.AutoStepHelp = false
  oXAxis.TextBreak = false
  oYAxis = oDiagram.getYAxis()
  oYAxis.AutoOrigin = true
  nDateFormat =_
  oXAxis.NumberFormats.getStandardFormat(_
    com.sun.star.util.NumberFormat.DATE, aLanguage )
  oXAxis.NumberFormat = nDateFormat
```

If the charts collection (oCharts) of this sheet is empty, we have to create the chart first. It is positioned just above the cells displaying the data. We use the Position and Size properties of the range B2:H17 to find out the size and position of the rectangle to contain the chart. As the first cell of the sheet is located at B20, this rectangle lies above the stock quote values. The chart is then added to the charts collection and its diagram type is set to XYDiagram, which ensures that it uses lines to display the evolution of the quotes. The rest of the code just sets some of the diagram properties.

```
Else
   oChart = oCharts(0)
   oChart.Ranges = oDataRanges.RangeAddresses
   oChart.HasRowHeaders = false
   oEmbeddedChart = oChart.EmbeddedObject
   oDiagram = oEmbeddedChart.Diagram
   oXAxis = oDiagram.XAxis
End If
```

The next lines are executed if the program has already been executed and the chart therefore exists. They simply change its data range so that it contains all dates and quotes and set the oXAxis variable.

```
  fMin = oCursor.getCellByPosition(0,1).Value
  fMax = oCursor.getCellByPosition(0,oCursor.Rows.Count-1).Value
  oXAxis.Min = fMin
  oXAxis.Max = fMax
  oXAxis.AutoMin = false
  oXAxis.AutoMax = false
End Sub
```

This variable is needed because the minimum and maximum of the x axis have to be updated each time the quotes are retrieved. The lines above simply get the first and last date for the current quote (`fMin` and `fMax`) and adjust the x axis accordingly. When you have run this program several times, the chart looks similar to the one shown below.

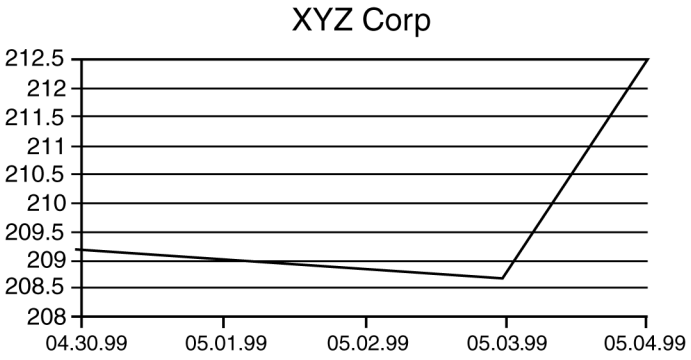

Figure 5–2 Stock quotes

5.5 Troubleshooting

5.5.1 Debugging

Most programs you write will not work right from the beginning. Some errors like misspelled or missing keywords are caught by the StarBasic interpreter before it even starts to execute your program. These are easy to correct. Others are a bit more difficult to catch because they happen at runtime.

StarOffice API provides three functions that permit you to inspect objects at runtime:

- `DBG_properties()` returns all properties defined for a class. Note that some properties need not be defined for the particular object you are inquiring. Before accessing this property, you should check its existence with `isNull()`.
- `DBG_methods()` returns all methods defined for a class.
- `DBG_supportedInterfaces()` returns all interfaces supported by the current object.

5.5.2 Displaying the object details

Sometimes it is not enough to know which properties an object has - you want to see their values. Although this is not something easily accomplished in Basic, we will show you some subroutines that should get you started. They expect an object as input and generate a text document containing the names and values of all properties in a table.

```
Sub GeneratePropertiesTable(oObject)
  Dim vVariant as Variant

  mProperties = oObject.PropertySetInfo.Properties
  nCount = UBound(mProperties)-LBound(mProperties) + 2
  oDocument = CreateTextDocument()
  oText = oDocument.Text
  oCursor = oText.createTextCursor()
  oCursor.gotoStart(FALSE)
  oTable = CreateTable()
  oTable.initialize(nCount,2)
  oText.insertTextContent(oCursor, oTable, FALSE)
  oTableCursor = oTable.createCursorByCellName(oTable.CellNames(0))
  oTableCursor.gotoStart(FALSE)
  InsertNextItem("Name", oTableCursor, oTable)
  InsertNextItem("Value", oTableCursor, oTable)
  For i% = LBound(mProperties) To UBound(mProperties)
     p = mProperties(i%)
     n$ = p.name
     vVariant = oObject.getPropertyValue(n$)
     InsertNextItem(n$, oTableCursor, oTable)
     If IsNull(vVariant) Then
       sString = "NULL-Value"
     ElseIf IsObject(vVariant) Then
       sString = vVariant.dbg_properties
     ElseIf IsArray(vVariant) Then
       tmp$ = ""
       For j% = LBound(vVariant) To UBound(vVariant)
       vItem = vVariant(j%)
       If IsNull(vItem) Then
         sItemDescription = "NULL-Value"
       ElseIf IsObject(vItem) Then
         sItemDescription = vItem.dbg_properties
       Else
         sItemDescription = cstr(vItem)
       End If
       tmp$ = tmp$ + sItemDescription + "/"
       Next j%
       sString = tmp$
     Else
       sString = cstr(vVariant)
     End If
     InsertNextItem(sString, oTableCursor, oTable)
  Next i%
End Sub
```

The main entry point is the subroutine `GeneratePropertiesTable()`. It creates an empty text document (`CreateTextDocument()`) and inserts a table with two columns in it (`createTable()`). The subroutine `insertNext()` is used to insert a string into the current table cell and advance the table cursor to the next field. It is first employed to add the headings to the two columns. We then loop over all properties (`for i% = :..`) and first insert the name of each of them. We then try to figure out its value and convert it to a suitable string. If the property's value is itself an object, we use its `dbg_Properties()` method to display at least the names of the object's properties. If the property is a sequence, we loop over all of its elements and perform the same value steps as for single values. Note, however, that we can't display arrays of arrays with this approach.

```
Sub InsertNextItem(what, oCursor, oTable)
  Dim oCell As Object

  sName = oCursor.getRangeName()
  oCell = oTable.getCellByName(sName)
  oCell.String = what
  oCursor.goRight(1,FALSE)
End Sub
```

Subroutine `InsertNextItem()` expects a string (`what`), a table cursor (`cursor`) and a table. It inserts the string at the current cursor position, changing the adjustment to right if the string is a number. It then moves the table cursor one cell to the right. If the current cell was the last one in this row, this moves the cursor to the first cell on the next row.

```
Function CreateTable() as Object
  oDocument = ThisComponent
  oTextTable = oDocument.createInstance("com.sun.star.text.TextTable")
  CreateTable = oTextTable
End Function
Function CreateTextDocument() as object
  dim noargs()
  CreateTextDocument = StarDesktop.loadComponentFromURL(url,"_blank", 0, noargs)
End Function
```

The subroutines `CreateTable()` and `CreateTextDocument()` do exactly what their names suggest. We are mainly using subroutines here to keep the main routine somewhat shorter.

APPENDIX **A**

The UML Class Diagrams

Class diagrams show the static structure of the model, in particular, the things that exist (such as classes and types), their internal structure, and their relationships to other things. Class diagrams do not show temporal or behavioral information. All class diagrams shown below are based on the UML 1.1 notation.

A.1 UML

The Unified Modeling Language (UML) is a language for specifying, visualizing and documenting the artifacts of software systems. It represents a collection of the best engineering practices that have proven successful in the modeling of large and complex systems. The UML 1.1 specification was adopted by the Object Management Group (OMG) as an official standard. The websites of Rational Software Corporation and the OMG provide detailed information concerning the specification of UML.

A.2 Stereotypes used by StarOffice API

UML allows you to extend the model by making use of stereotypes. A stereotype is, in effect, a new class of modeling elements that is introduced at modeling time. It represents a subclass of an existing modeling element with the same form but with a different intent. Generally a stereotype represents a usage distinction.

There are two stereotypes of classes, which are important to StarOffice API. Whereas *interface*s are already part of the UML model, the stereotype *service* has been added

by StarOffice API to reflect the definition of StarOffice API `services` described in this book.

In the figure below you see both an interface and a service. XSearchDescriptor provides operations to set and retrieve a string to search for. This set of operations might be useful to various services. The service SearchDescriptor specifies such an application. It describes a piece of software which supports the operations of interface XSearchDescriptor and is able to handle properties like SearchCaseSensitiv or SearchRegularExpression. These properties hold further information about the search string and how to interpret its content.

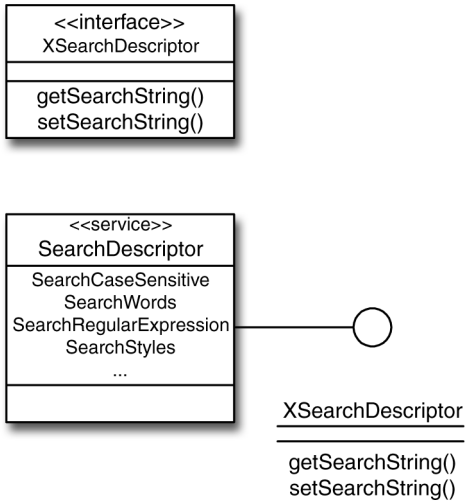

Figure A–1 StarOffice API stereotypes *service* and *interface*

A.3 Interface

An *interface* is a specifier for the externally-visible operations of a class, component, or other entity without specification of internal structure. Interfaces do not have implementation; they lack attributes, states, or associations; they only have operations. An interface is formally equivalent to an abstract class with no attributes and no methods and only abstract operations, but Interface is a peer of Class within the UML metamodel; both are Classifiers.

A.4 Service

A *service* is a subclass of the UML Classifier Class. Like an interface, it is an abstract class and does not have an implementation, but it does have an internal structure, attributes and associations. Services are designed for a more specific purpose than interfaces and gain their external behavior through properties and the interfaces they support.

A.5 Relations used in the Class Diagrams

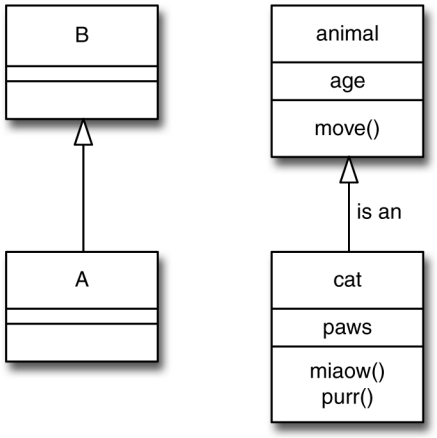

Figure A–2 Generalization

Generalization is the relationship between a more general element and a more specific element that is fully consistent with the first element and that adds additional information. Generalization is shown as a solid-line path from the more specific element to the more general element, with a large hollow triangle at the end of the path where it meets the more general element. The arrow from A to B is read as "A inherits from B." This includes "Only A knows B, B does not need to know anything about A." In every-day-language, this relationship would be expressed as "is a": "A is an B" or "a cat is an animal" .

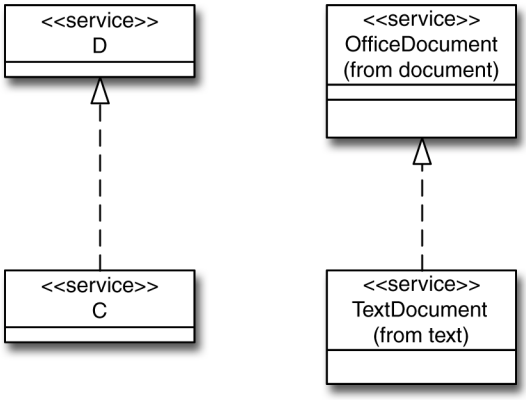

Figure A–3 Realizes relationship

The *realizes* relationship from a class to an interface that it supports is shown by a dashed line with a solid triangular arrowhead (a "dashed generalization symbol"). This is the same notation used to indicate realization of a type by an implementation class. In fact, this symbol can be used between any two classifier symbols, with the meaning that the client (the one at the tail of the arrow) supports at least all of the operations defined in the supplier (the one at the head of the arrow). StarOffice API diagrams make use of this relationship to indicate that a client service extends another service by the interfaces and properties of the supplier. In the example the arrow from C to D is read as "Service C realizes or extends service D." Service TextDocument offers the basic functionality common to all documents , descibed as service OfficeDocument, by properties and interfaces, which are specific to text documents.

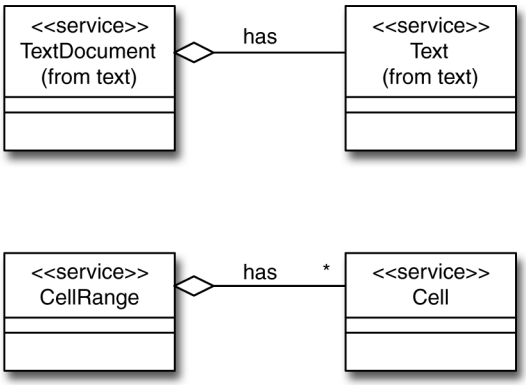

Figure A–4 Binary association

A *binary association* is an association among exactly two classes (including the possibility of a reflexive association from a class to itself). it is drawn as a solid path connecting two class symbols. A hollow diamond may be attached to the end of the path to indicate aggregation. The diamond is attached to the class that is the

aggregate. In every-day-language,this relationship would be expressed as "is part of" or "has": "A TextDocument has Text" or "Cells are part of a CellRange".

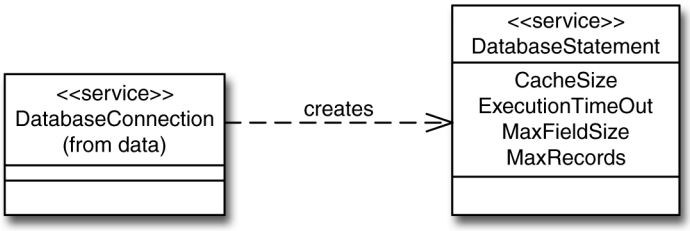

Figure A–5 Dependency relationship

A *dependency* indicates a semantic relationship between two or more model elements. It relates the model elements themselves. In most cases this relationship indicates that one object uses another. (e.g. instantiating an object of another class). A dependency is shown as a dashed arrow between two model elements. The model element at the tail of the arrow depends on the model element at the arrowhead. In the figure above you read the association as "A DatabaseConnection creates a DatabaseStatement". In case you are in need to create a service DatabaseStatement, ask a DatabaseConnection, it will create one for you.

Glossary

BASIC	Beginners all purpose simple instruction code: One of the oldest and easiest to learn programming languages.
Cell	The smallest addressable part of a spreadsheet or a table. A cell can contain text, a number or a formula.
CSV Comma Separated Value	A format used to store tabular data in an independent form. The columns need not be separated by commas anymore, most programs permit to choose the separator freely.
Cursor	Object that permits movement in a document. Depending on the document's type, different cursors are available.
Event	Something that happens inside or outside of a program outside of the normal flow of control. Events can not be foreseen, programs wanting to react to them have to install event handlers that are triggered when an event is received. Typical events are a key press or the movement of the mouse.
GUI Graphical User Interface	The menus, dialogs, buttons etc. used to tell a program what it should do.
MIME Multipurpose Internet Mail Extension	Textual definitions first used in email to permit the inclusion of binary documents. For example, "application/ vnd.sun.staroffice.writer" indicates that the following document is to be used with Starwriter. Note that not all MIME types are standardized.
Property	Characteristic of an object. Properties have a name by which their value can be inquired and possibly set.
Range	Part of a document that can be spanned by a cursor.

Service	Abstract concept providing interfaces and properties. All implementations of the same service provide the same interfaces.
Shape	The abstract concept of a graphic object. Different types of shapes are rectangles, ellipses, text, polygons, etc.
SQLStructured Query Language	A language used to insert, update, delete and retrieve data from relational databases. Although SQL is normed, each database implements its own extensions.
Style	The collection of attributes applied to a particular part of a document. For example, a paragraph style specifies the font, the linespacing, the justification, the indentation and other aspects used to format a paragraph.
TextRange	A part of a text described by a text cursor. The range can consist only of a text position if the start and end position of the cursor coincide.
UNO Universal Network Objects	Basic services such as the ServiceManager, Property APIs etc used by StarOffice API.
URL Universal Resource Locator	String describing an object on any computer in the internet. `http://www.sun.com/index.html` is an URL pointing to the file `index.html` on the computer `www.sun.com`. `http` indicates the service used to access the object, other services are `ftp` or `news`.