TUHH

Technische Universität Hamburg-Harburg

Bachelor Thesis

Implementation of OpenOffice.org Add-ons for Text Extraction

submitted by

Ky Anh Tuan Tran

Supervisors:

Prof. Dr. Ralf Möller
Dipl.-Ing. Maurice Rosenfeld

Institute of Software, Technology & Systems (STS)
Technical University Hamburg–Harburg
Hamburg, GERMANY

February 2009

Acknowledgements

I would like to express my sincerest gratitude to my supervisor, Dipl.-Ing. Maurice Rosenfeld, who has supported me throughout this thesis with his patience in discussing and giving me lots of helpful criticisms. I am also deeply indebted to Prof. Ralf Möller who has given me guidance and encouragement to proceed with the interesting topic of the thesis.

Declarations

I hereby declare that this thesis is my own original work and effort, conducted under the supervision of Prof. Dr. Ralf Möller and Dipl.-Ing. Maurice Rosenfeld. All sources of information used have been acknowledged in the bibliography part.

Hamburg, February 1st, 2009

Ky Anh Tuan Tran

Contents

1. Introduction 4

1.1 Motivation 4

1.2 Document Structure 4

2. Background 6

2.1 Universal Network Objects (UNO) 7

2.1.1 UNO Objects 8

2.1.2 UNO Services 9

2.1.3 UNO Interfaces 10

2.1.4 UNO Properties 12

2.2 OpenOffice.org Application Environment 12

2.2.1 Desktop Environment 13

2.2.2 Framework API 15

2.2.3 Frame-Controller-Model Paradigm in OpenOffice 15

3. Implementation 20

3.1 OpenOffice.org Add-On with NetBeans IDE 21

3.2 Text working and getting a paragraph from current processing text document 22

3.2.1 XTextDocument, the first step to work with a OpenOffice.org document 22

3.2.2 Model Cursor and View Cursor 24

3.2.3 Getting text content from paragraphs 25

3.3 Working with OpenOffice.org GUI 27

3.3.1 Dialog Model 27

3.3.2 Create and insert FixedText and ListBox to Dialog 30

3.3.3 KeyListener, KeyHandler and MouseListener in UNO 34

3.3.4 Open a document in a new window 35

3.4 OpenOffice as a web client in web service 36

4. Conclusion and Outlook 38

5. Bibliography 40

Chapter 1
Introduction

1.1 Motivation

Openoffice.org(OpenOffice) is an open-source office software which supports word processing with the *OpenOffice Writer*, spreadsheets with *OpenOffice Calc*, presentations with *OpenOffice Impress*, graphics with *OpenOffice Draw* and databases with *OpenOffice Base*, etc. The user can save all data in different formats like *.odg, .odp, .ods* or *.odt*, which can be read or changed by other office softwares.

The main purpose of this project is the semantic text analysis from OpenOffice Writer. By using OpenOffice *Universal Network Objects (UNO)*, the *OpenOffice Application Environment* is accessed to get the content of a document processed in OpenOffice Writer. From this step, the text content is analyzed semantically to get a desired instance such as a paragraph, a sentence or a word. In this project, the desired instance is defined as a paragraph that is used as an input to an external server. The main function of the server is a semantic search machine. It analyzes the input and the other documents semantically and produces a response as a list of semantic related documents together with their information via the names of the documents, the links to access them, etc.

OpenOffice Writer receives the response from the server and provides the user with a service, so that he can choose a desired document from the list and open it in a new window to read or process. The main focus of this project is on the side of OpenOffice Writer.

1.2 Document Structure

Chapter 2 introduces the theoretical background of OpenOffice. This chapter provides the reader with an overview of *UNO* and *OpenOffice.org Application Environment*.

Chapter 3 points out different aspects of this project through its implementation. Session 3.1 explains how to start working with Add-On by using NetBeans. While session 3.3 involves working with text contents of the OpenOffice Writer document, session 3.4 introduces the way to work with GUI of the OpenOffice Writer. Session 3.4 discusses its role as a web client in web service of OpenOffice.

Chapter 4 presents several frequently met problems when working with OpenOffice and suggests further improvements that can be made with OpenOffice.

Chapter 5 shows the bibliography of this project.

Chapter 2

Background

To start working with OpenOffice, it is important to understand the concept of *Universal Network Objects* and the *OpenOffice.org Application Environment*. They are the basic concepts providing the developer with an overview of Openoffice components as well as the way to access and to program with OpenOffice. UNO make it possible to work with OpenOffice using different programming languages such as Java, C++, OpenOffice Basic and .NET. An overview and the components which the developer works on will be introduced in session 2.1 Universal Network Objects. UNO define the amount of objects, interfaces and properties that programmers must understand before analyzing the structure and the way to work with a new programming environment. UNO services provide the instances of OpenOffice components such as paragraphs, sentences, cursors, etc. Being either visible or invisible, they allow the interfaces to access themselves or let the function set their properties. Therefore, UNO services, UNO interfaces and UNO properties are the UNO components that build the structure of UNO.

Together with UNO, the *OpenOffice.org Application Environment* builds the basic structure for OpenOffice. It provides the developer with a model hierarchy to explain how the instances are organized in OpenOffice. Figure 1.1 provides an overview of the *OpenOffice.org Application Environment's* structure. The *OpenOffice.org Application Environment* is divided into two parts: the *Desktop Environment* and the *Framework API*. The *Desktop Environment* is built by *Desktop Service* and *Auxiliary Objects*. The *Desktop Service* is the central management instance of the OpenOffice application framework, while the *Auxiliary Objects* are the sub instances of the hierarchy and interact directly with the UNO-based office. The *Framework API* is responsible for setting the visibility of the components and processing the requests from the user or from OpenOffice. To have a clearer overview of this Framework API, the *Frame-Controller-Model Paradigm* is analyzed.

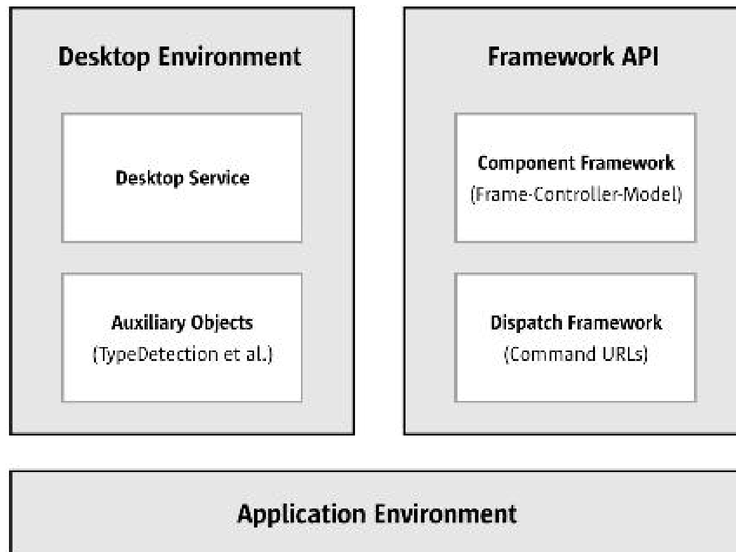

Figure 1.1: OpenOffice.org Application Environment [1]

2.1 Universal Network Objects (UNO)

By using UNO, the developer can program and work with OpenOffice from various programming languages such as Java, C++, .NET, or Openoffice.org Basic. Furthermore, UNO supports many operating system environments like Linux, Solaris, Windows, Mac OS X, FreeBSD, etc.

OpenOffice can be accessed from UNO, because *Application Programming Interface (API)* is supported by UNO. It is OpenOffice API that makes OpenOffice programmable from the user's view. This API contains a big amount of services and interfaces which allow the developer to work with every component of OpenOffice such as the OpenOffice Writer, the OpenOffice Cal, etc. by writing an Add-On, Add-In or a complete application as a client of OpenOffice.

One of the Java components used for this project is Java Beans which supports UNO and integrates with Java *IDEs (Integrated Development Environment)* to create a container used for data transmission to OpenOffice. It makes the access to OpenOffice easier. After installing the *OpenOffice.org API plug-in* (found in the plug-in package of NetBeans) and the *OpenOffice.org Software Development Kit (SDK)*[2], the developer still needs to add in some API references to NetBeans IDE, then he can choose which kind of application he wants to write to access OpenOffice. It can be an *OpenOffice.org Client Application Project Type*, an *OpenOffice.org Calc Add-In Project Type*, a *General UNO Component Project Type* or an *OpenOffice.org Add-On Project Type*. These types simplify the access and usage of the API in new projects and make it easier and faster to program or create complete OpenOffice extension packages. In this project, the *Openoffice.org Add-On Project Type* is used.

2.1.1 Objects in UNO

An object in UNO is a software artifact that has methods to call and attributes to set and get. The UNO object supports a set of interfaces that specify methods and attributes. UNO use multiple-inheritance interfaces and services to specify complete objects which can have many aspects. Each single-inheritance interface describes only one aspect of an object. Therefore, one object can implement multiple interfaces.

OpenOffice objects can inherit services, including interfaces from only one parent object. Figure 2.1 demonstrates the relationship between objects, services and interfaces. Within this figure, *OfficeDocument* object is the parent object of *TextDocument* object.

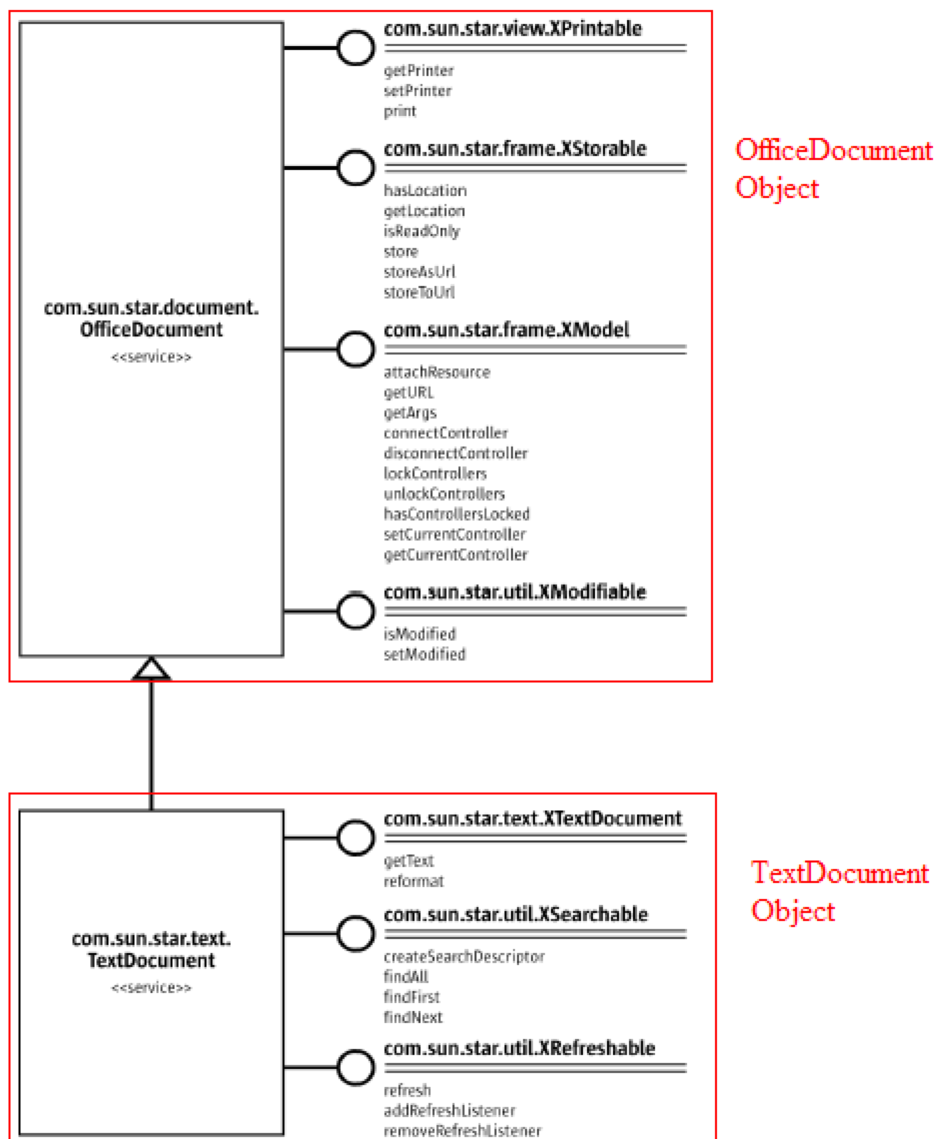

Figure 2.1: Relationship between Objects, Services and Interfaces

2.1.2 UNO Services

Objects in UNO can be seen as UNO services in OpenOffice.org. The *com.sun.star.lang.ServiceManager* is a main factory in every UNO application to create the services (see figure 2.2). It initiates the services with their service names, for example: *com.sun.star.frame.Desktop* for loading the documents or managing the access to current documents, *com.sun.star.text.GlobalSetting* for setting a view and a print format of text documents. We will discuss the importance of desktop service and the relationship between those services later in the session 2.2.

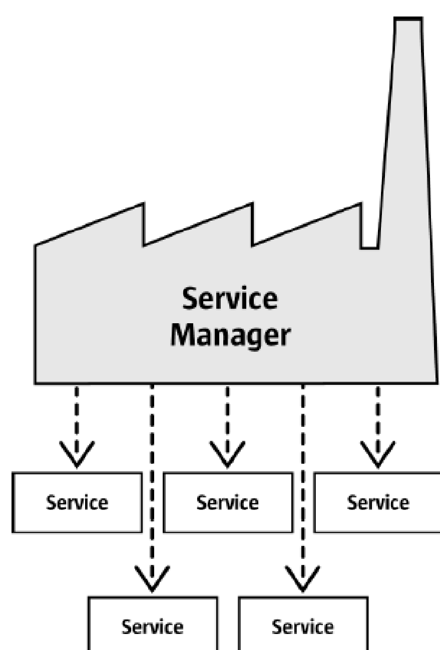

Figure 2.2: Service Manager [1]

There are various methods to create a service in OpenOffice. We go back to service manager. It provides a main *com.sun.star.lang.XMultiServiceFactory* interface offering three methods: *createInstance()* returning a default *com.sun.star.uno.XInterface* service instance which supports all interfaces specified for the requested service name; *createInstanceWithArguments()* returning a instantiation of the service with additional parameters; and *getAvailableServiceNames()* returning every service name that the service manager does support. For example: By using the service manger, a *FixedTextModel* service can be created.

```
Object oFixedTextModel = xMultiServiceFactory.createInstance
("com.sun.star.awt.UnoControlFixedTextModel");
```

However, the service manager described above is not suitable for the instantiation of every new component, because a new component needs more functions and the

information must be exchangeable during the implementation process. The service manager does not support in this case. Therefore, the concept of component context is needed. It can be seen as a container offering named values and every named value goes with a component, for example: service manager is one of the named values.

A component can be reached from the component context by giving a request together with a named value. The component context supports only the *com.sun.star.uno.XcomponentContext* interface that offers *getValueByName()* method to get an implementation property or a service property but it is used mostly to get a singleton. Singletons are used to specify named objects where exactly one instance can exist in the life of a UNO component context like *theServiceManager*, for example:

```
Object serviceManager = xComponentContext.getValueByName
("/singletons/com.sun.star.lang.theServiceManager");
```

Figure 2.3 shows the relationship between the component context and the servicemanager together with their interfaces.

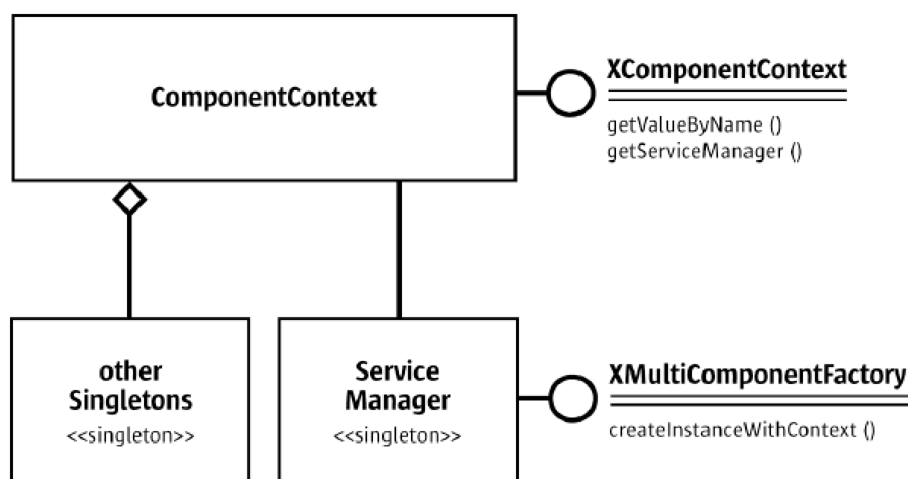

Figure 2.3: ComponentContext and the ServiceManager [1]

In terms of the relationship between the service manager and the component context, the service manager supports the *com.sun.star.lang.XMultiComponentFactory* interface which replaces the *XMultiServiceInterface*, because it has an additional *XComponentContext* parameter for the two object creation methods, for example: both objects *Desktop* and *UnoControlDialogModel* can be created from a context:

```
Object oDialogModel =
xMultiComponentFactory.createInstanceWithContext(
"com.sun.star.awt.UnoControlDialogModel",
xComponentContext);

Object desktop =
xMultiComponentFactory.createInstanceWithContext(
"com.sun.star.frame.Desktop",  xComponentContext);
```

2.1.3 UNO Interfaces

In OpenOffice.org API, names of all interfaces start with “X” to be distinguishable from the names of other entities. The developer cannot use a factory like a service manager to access methods of an interface which the object supports, because the developer needs to request the *UNO Runtime Environment* to get the appropriate reference for this interface while the compiler is not aware of it. Indeed, the compiler only knows about objects but does not know about interfaces of the object. However, it is sometimes necessary to access the interfaces, as there is a need to call a method of an interface. Thus the Java *UNO Runtime Environment* provides *queryInterface()* method to solve this problem. This method is about safe casting of UNO types across process boundaries. It makes sure the developer gets a reference that can be casted to the needed interface type, no matter if the target object is local or a remote object, for example: the *XtextDocument* interface is gotten from the *Desktop* object by using the function *queryInterface()*.

```
XComponentLoader xComponentLoader = (XComponentLoader)
UnoRuntime.queryInterface(XComponentLoader.class, desktop);
```

The developer usually does not need to query an interface from an object, because he can also query an interface from another interface, for example:

```
XSpreadsheetDocument xSpreadsheetDocument =
(XSpreadsheetDocument) UnoRuntime.queryInterface(
XSpreadsheetDocument.class, xComponent);
```

The call to *queryInterface()* is necessary in Java only when the developer has a reference to the object which supports the desired interface. Nonetheless, he does not have the proper reference type yet. Another method which can return the desired interface from the existent interface is that, the existent interface provides a function for this purpose, for example:

```
XController xController = xFrame.getController();
```

The *XController* interface is gotten from the *XFrame* interface by using the *getController()* method . Both methods to get an interface from *UnoRuntime* or from another interface are used many times in this project.

Sometimes the developer does not need an agent service to get the desired service containing the function he needs. He can also work with the interfaces of the services. This is the best way to work with OpenOffice.org containing a complicated structure.

2.1.4 UNO Properties

Each OpenOffice.org object has a lot of properties and offers these properties through its interfaces so that the developer can work with them. There are two types of property interfaces. The most basic form is *com.sun.star.beans.XPropertySet*. The other type, *com.sun.star.beans.XmultiPropertySet,* allows the developer to get or set many values of properties with a single method call. The latter is more popular among the developers. Two methods to work with *XPropertySet* in Java:

```
void setPropertyValue(String propertyName, Object propertyValue);

Object getPropertyValue(String propertyName);
```

It can also be possible to query *XPropertySet* from *XMultiPropertySet*, for example:

```
//XPropertySet from (XMultiPropertySet)  xLBModelMPSet

XPropertySet xLBModelPSet = (XPropertySet) UnoRuntime.queryInterface(XPropertySet.class,
xLBModelMPSet);



//set value for (XPropertySet) xLBModelPSet

xLBModelPSet.setPropertyValue("MultiSelection",
Boolean.FALSE);
```

The way to set values with *XMultiPropertySet* is a bit different. The developer must remember to pass the property names in alphabetical order.

```
void setPropertyValues(String[] propertyName, Object[] propertyValue);
```

Usage sample of *XMuiltiPropertySet* will be explained further in the chapter 3.

2.2 OpenOffice.org Application Environment

The OpenOffice allocates defined interfaces which provides a way to access OpenOffice from external programs or to load another OpenOffice document from the processing document. To understand the structure of OpenOffice, it is necessary to have knowledge of the *Openoffice.org Application Environment* which consists of the *Desktop Environment* and the *Framework API*. The overview is shown in Figure 1.1.

Desktop Services and *Auxiliary Objects* are the principle parts of the *Desktop Environment* whose functions are executed by using the *Framework API*. On the other side, the *Framework API* includes *Component Framework* and *Dispatch Framework*. While the *Component Framework* is responsible for making components viewable in OpenOffice, the *Dispatch Framework* handles command requests from the graphic user interface.

2.2.1 Desktop Environment

The *com.sun.star.frame.Desktop* service is the centre of the *OpenOffice.org Application Environment*. All windows with the viewable components are based on this service and organized in a hierarchy of frames because the desktop is the root of this hierarchy. Through the *com.sun.star.frame.XDesktop* interface of the *Desktop* service, the viewable components can be loaded, the frames and components can be accessed, and the office can be terminated, etc. Figure 2.4 shows the relationship between desktop service, frames and components.

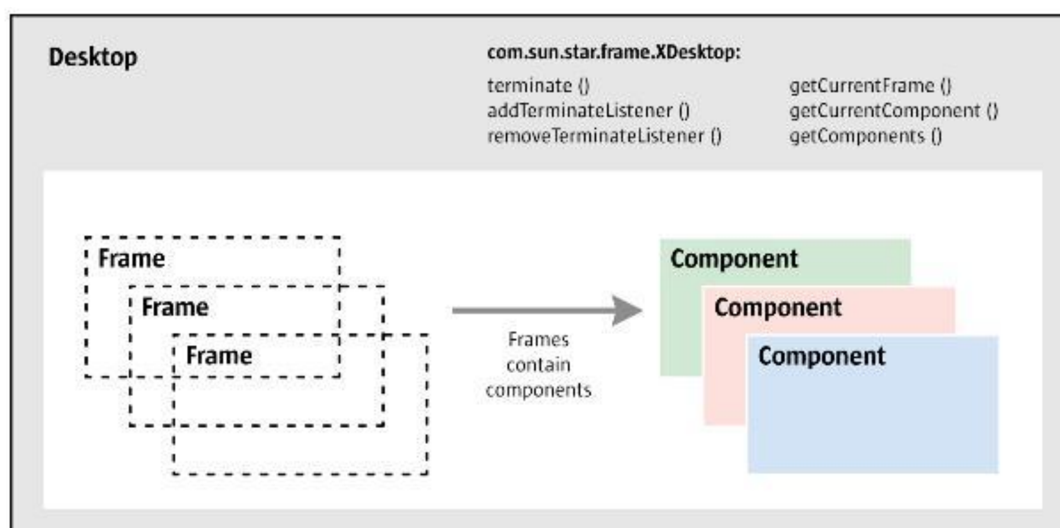

Figure 2.4: The Desktop terminates the office and manages components and frames [1]

The desktop object and frame objects use the *Auxiliary Services* which interact with the UNO-based office, but are not accessible by the OpenOffice.org API. Therefore, the desktop service and surrounding objects are grouped together in the *Desktop Environment*. This definition is used, whereas the problem concerns with the viewable components, the windows, as well as the frames.

The viewable components are classified into three different kinds:

- Office documents with a document model and controller, for example, a normal text document or calculation document.

- Components with a controller but without model, for example, a database browser.

- Windows without API-enabled controller, for example, message windows.

All of them contain the *com.sun.star.lang.XComponent* interface, which offers the facility to access those components.

The frames are used for establishing the connections between the components, the windows and the desktop environment. The frame plays therefore an important role in *OpenOffice.org Application Environment*. The frame is seen as a sub class of the desktop service. It is shown in figure 2.5.

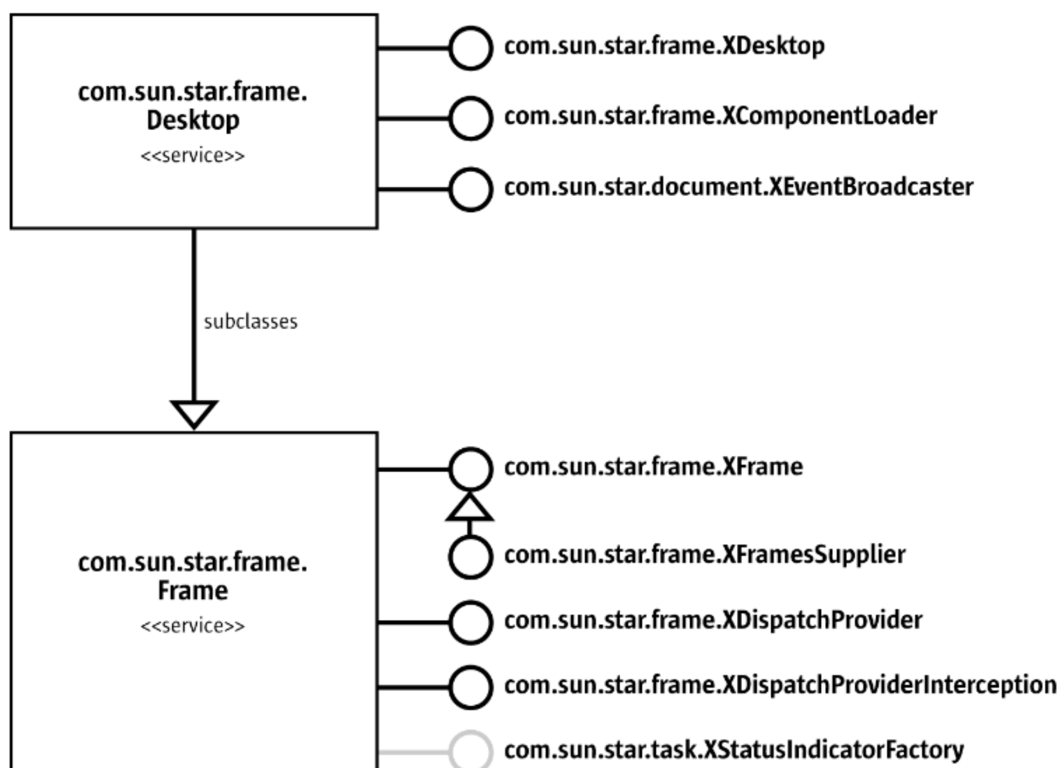

Figure 2.5: Description of the desktop service and the frame [1]

The above figure shows the relationship between *com.sun.star.frame.Desktop* service and *com.sun.star.frame.Frame* service together with their interfaces. The most important interface of the service *Desktop* is *com.sun.star.frame.XDesktop*. The *XDesktop* interface makes the access to the frames and the components available. It defines the following methods:

```
com.sun.star.frame.XFrame getCurrentFrame();

com.sun.star.lang.XComponent getCurrentComponent();

com.sun.star.container.XEnumerationAccess getComponents();
```

getCurrentFrame() and *getCurrentComponent()* functions return the active frame and the active document model, while *getComponent()* function returns the interface to all loaded

documents. This interface also provides the methods to control the termination of the office process:

```
boolean terminate ();

void addTerminateListener (
com.sun.star.frame.XTerminateListener xListener);

void removeTerminateListener (
com.sun.star.frame.XTerminateListener xListener);
```

2.2.2 Framework API

There are two types of framework: *Component Framework* and *Dispatch Framework*, depending on which interface of OpenOffice components it implements. The *Component Framework* is responsible for implementing the *Frame-Controller-Model Paradigm* to define the relationship among objects in OpenOffice. We will discuss this paradigm further in the session 2.2.3.

The *Dispatch Framework* helps the components in the *Component Framework* work together, because the frame provides the communication context with the components that it contains and the communication from a controller to the *Desktop Environment* with help of the interface *com.sun.star.frame.XDispatchProvider*. The *Dispatch Framework* also manages or implements command requests from or to application environment. The dispatch API is called from UI to access the component whenever a command is needed to be dispatched. Because only the component knows how to execute a command, the dispatch API calls are handled by the frame. The frame can assign exactly the component that can handle the command.

2.2.3 Frame-Controller-Model Paradigm in OpenOffice

Frame-Controller-Model (FCM) Paradigm is similar to the well-known *Model-View-Controller (MCV) Paradigm* in software engineering. MCV is an architectural pattern for structuring in software development. The aim of MCV is to show a flexible program design that facilitates changes or expansions and enables the reusability of the separate component. The MCV isolates three application areas: document data (model), presentation (view) and interaction (controller). But FCM is quite different from MCV. It is applied in three other areas: model for document object, controller for managing the screen presentation of the model, and frame for establishing the connection between controller and window. Figure 2.6 shows the organization of the FCM.

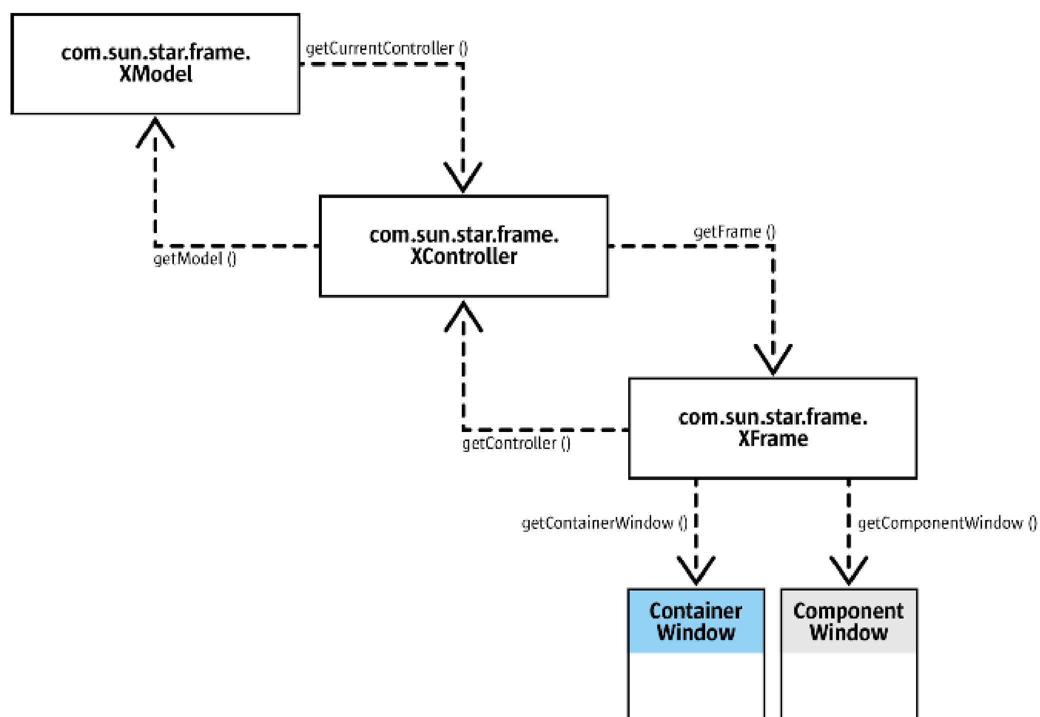

Figure 2.6: Frame-Controller-Model Organization [1]

- *The Model:* Through the model, we can access all data of a document directly, for example: a text, a table or other components. It is important to understand that the developer has to work with the model directly when he want to change it through the OpenOffice API. The model has a controller object which enables the developer to manipulate the visual presentation of a document in the user interface.

- *The Controller:* The controller is used as a bridge to present the document on the screen without any knowledge of the data in this document. It interacts with the user interface for movements, such as moving the visible text cursor, flipping through screen pages or changing the zoom factor. The controller establishes the connection between a frame and a document model with help from the *com.sun.star.frame.XController* interface. This interface has two methods: *getModel()* for getting the document frame and *getFrame()* for providing the frame that the controller attaches to. A detailed explanation can be found in *The Frame* part.

  Different controllers have different ways to present a document on the screen. It is also possible that two controllers are used for the same model. Figure 2.7 shows how the controller connects the frame and the model.

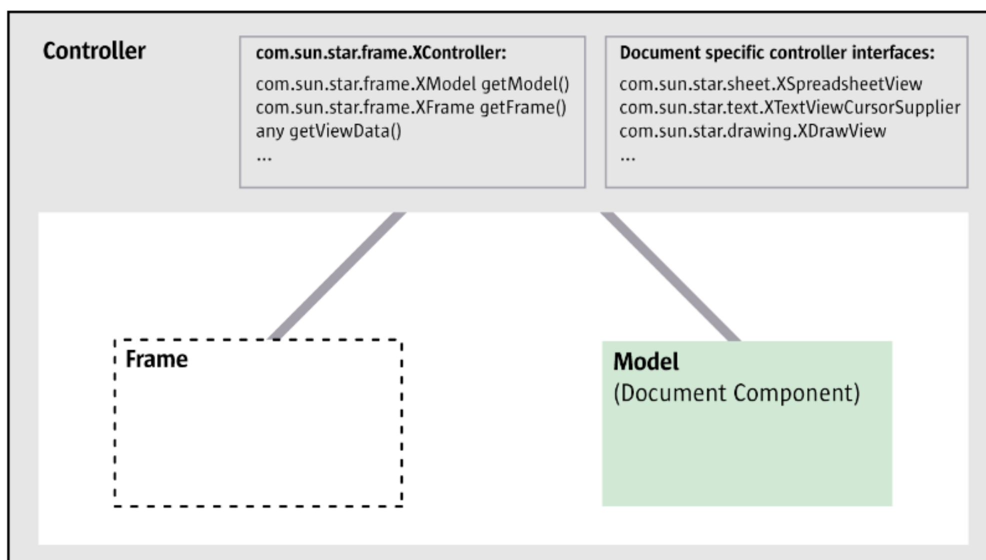

Figure 2.7: Controller with Model and Frame [1]

Another purpose of the controller is to gather information about the current view status, such as current selection, current page, total page count or line count. Usually the controller holds the model and only the model can have access to the data.

- *The Frame:* The main task of the frame is linking the components and the windows. A frame can contain one component or one component with subframe. A frame usually has two windows: a *container window* and a *component window*. A frame is constructed in a *com.sun.star.awt.XTopWindow* and this window becomes a container window when its frame is initialized in calling the *initialize()* method at the *com.sun.star.frame.XFrame* interface. The container window can appear in front of other windows or be sent to the background.

  The component window has a rectangular area to display the component and it receives GUI events while it is active. A *com.sun.star.awt.XWindow* is an instant which becomes a component window whenever the components are set in the frame with the *setComponent()* method at the *XFrame* interface. At the same time when the components are loaded into the frame, the interface *com.sun.star.frame.XController* is also called. The controller holds the model, which can access the components. That is why the components can get its window and the controller is called a bridge to present the components on the screen. Figure 2.8 shows the relationship between frame, subframe, container window and component window.

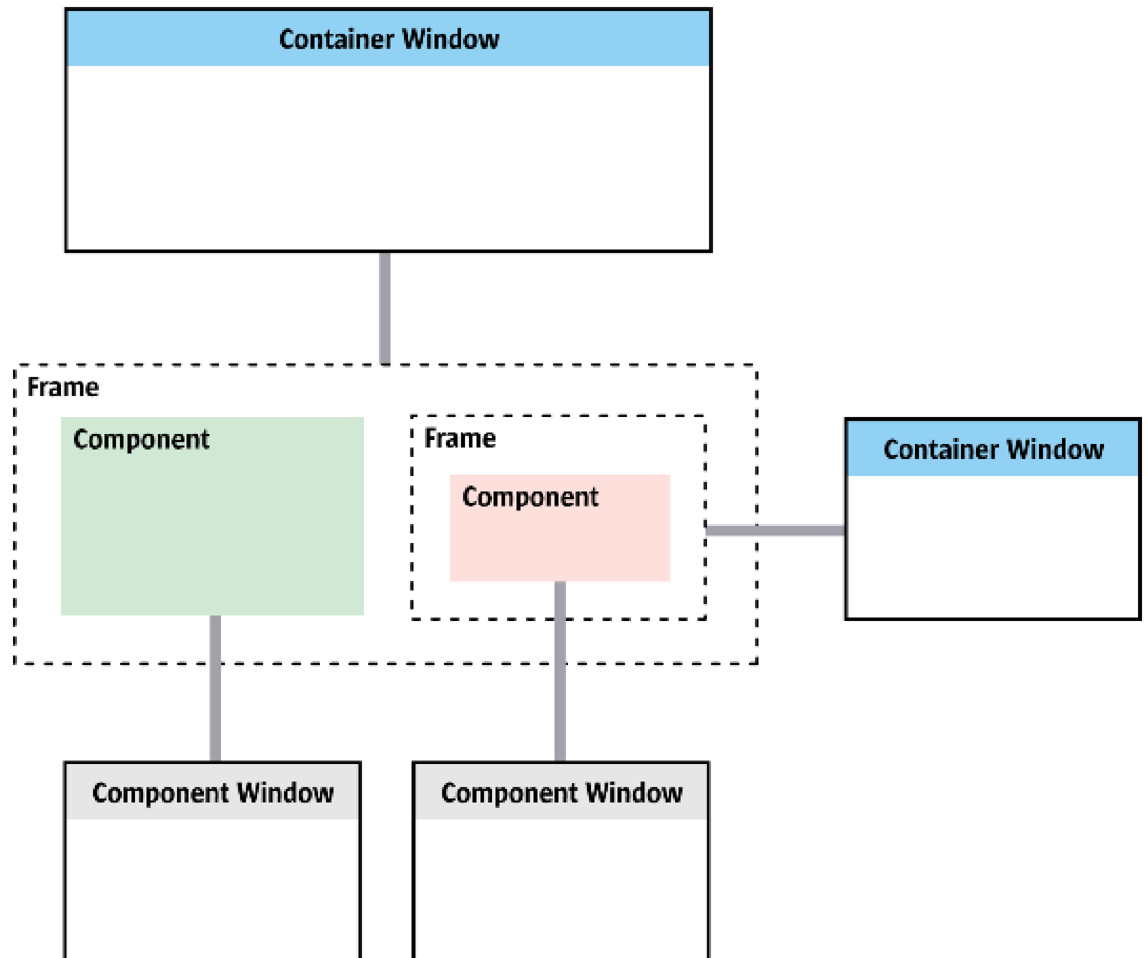

Figure 2.8: Frame containing a component and a sub-frame [1]

The *Desktop* service, which is available at the global service manager, includes the *Frame* service. In the desktop frame hierarchy, the *Desktop* becomes parent frame of other frames. The *Desktop* service provides the *com.sun.star.frame.XFramesSupplier* interface for this purpose. It is passed to the *setCreator()* method at the interface *XFrame*. Figure 2.9 shows a better overview about this relationship:

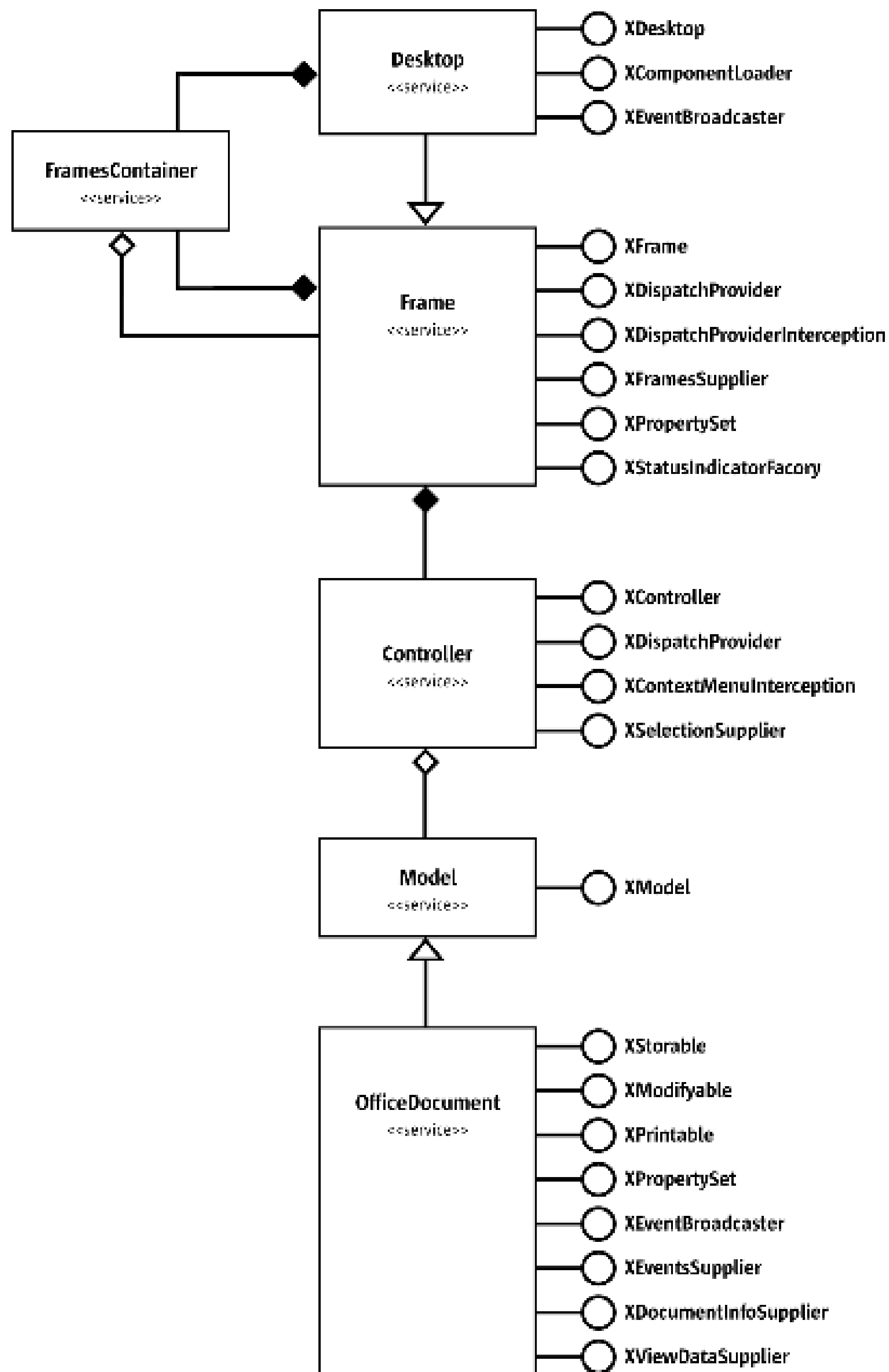

Figure 2.9: Desktop Service and Component Framework [1]

Chapter 3
Implementation

OpenOffice.org Add-On has two functions in this project:

- *The first function* allows the user to get a desired paragraph that is similar to the processing paragraph from the document.

- *The second function* sets OpenOffice as a client. The client sends the processing paragraph to the server and gets back the response as a list of documents.

But they both have the same first task, which is to handle the OpenOffice programming environment and to get the events from the users whenever the user selects the function he wants to start. Then the Add-On analyzes the semantic text content to extract a working paragraph for the next task. The following steps must be followed to complete this task:

- Create the Add-On from IDE and allocate the interfaces such as *XFrame*, *XComponentContext*, etc. (Session 3.1)

- Handle the selection of the user by creating and adding the *XKeyHandler* to OpenOffice from the *XFrame*. (Session 3.3.3)

- Analyze the text document and get the processing paragraph. (Session 3.2)

    - Get the processing document by accessing *XTextDocument*, *XController* from *XFrame.* (Session 3.2.1)

    - Get the *XTextCursor* model cursor and *XTextViewCursor* view cursor from *XController*. (Session 3.2.2). Then provide the *XParagraphCursor* and *XText* for the next step.

    - Save the text contents of all existent paragraphs in a *xParagraphArray* array from the *XText* and getting the text content of the processing paragraph from *XParagraphCursor*. This step is explained in session 3.2.3.

In terms of the first function, it is necessary to save all text paragraphs in order to compare with the processing paragraph in a later step. The program will display all similar paragraphs in a window on the screen so that the user can choose which one should be inserted into the working document without having to write the same text again.

One of the tasks of the second function is to pack the working paragraph in a message to send to the server. In this demonstration the *OpenOffice.org Writer* is used as a web client. Therefore the *OpenOffice Writer* can connect to the server with the help from a URL. Another important task is to process the response from the server and change it to a defined form. Like the first function of this program, the formatted data must be shown on a graphic user interface. It is a window containing the document index, from which the user can choose the desired document by a mouse click and using keyboard. The chosen document will be displayed on the graphic user interface on an external window for the user.

Thus, the following steps show how to display a window with the table form content and how to handle the selections from the user. This will be explained further in session 3.3.

- Create the dialog model and set the properties (Session 3.3.1). Then provide the *XMultiServiceFactory* of the dialog model from the *XComponentContext*. This factory is named as *xMSFDialogModel* which is used to create other components shown on the dialog. These components are inserted into the dialog by adding their names to the (*XNameContainer) xDlgModelNameContainer* provided in this step.

- Create and add the fixed text and the list box to the dialog by using *xMSFDialogModel* and *xDlgModelNameContainer* from last step (Session 3.3.2). Then add *XKeyListener* and *XMouseListener* to the dialog. (Session 3.3.2 and 3.3.3)

The second function still requires OpenOffice to send the processing paragraph to the server and to open the desired document from the selection of the user:

- The process is packed in a message and sent to the server. (Session 3.4)

- The desired document is opened by using the response from the server, the *XMultiComponentFactory* and *XComponentContext*.(Session 3.3.4)

3.1 OpenOffice.org Add-On with NetBeans IDE

OpenOffice.org Add-On is one of the applications supported by NetBeans IDE after installing the *OpenOffice.org API Plug-in*. From the main class of the Add-On project, the *XFrame* and *XComponentContext* interfaces are available. These interfaces play an important part in this project, because we can get more services or other interfaces from both of them.

There are still other available interfaces in the Add-On main class, for example: *com.sun.star.lang.XInitialization* to initialize our code or program in OpenOffice.org as soon as OpenOffice.org program starts up; *com.sun.star.frame.XDispatch* to implement the functions to execute whenever an event exists. The developer can create the buttons on the OpenOffice taskbar through the Add-On program and the interface *XDispatch* decides which functions will be executed once the user clicks on dedicated button. In this Project this function is not used, because the user will start up the Add-On program by clicking a defined key, for example F8 or F9 to start the program.

3.2 Text working and getting a paragraph from current processing text document

This session involves working with text documents and some objects or interfaces that are used to build this project. The *XTextDocument* interface is the first step to access the processing document. The important object of this project is the paragraph of a text document. However, to get the access to this object, we need agents as a bridge to reach the content of paragraphs from *XTextDocument*. The *view cursor* and the *model cursor* are offered for this purpose. The *view cursor* is visible from the user's view. There is only one *view cursor* which provides all information of the document from the current view, for example, the current page, the current paragraph or the current character. On the other side, the *model cursor* supports the user to travel over the document with different objects and interfaces to take what he wants, independently from view cursor. Two kinds of cursors offer two ways to get paragraphs from a text document: one for the current processing paragraph, the other for collecting the paragraphs for the purpose of the first function.

3.2.1 XTextDocument, the first step to work with an OpenOffice document

The text document model in OpenOffice API has five major architectural areas shown in Figure 3.1. The five areas are: the text, the service manager (document internal), the draw page, the text content suppliers and the objects for styling and numbering.

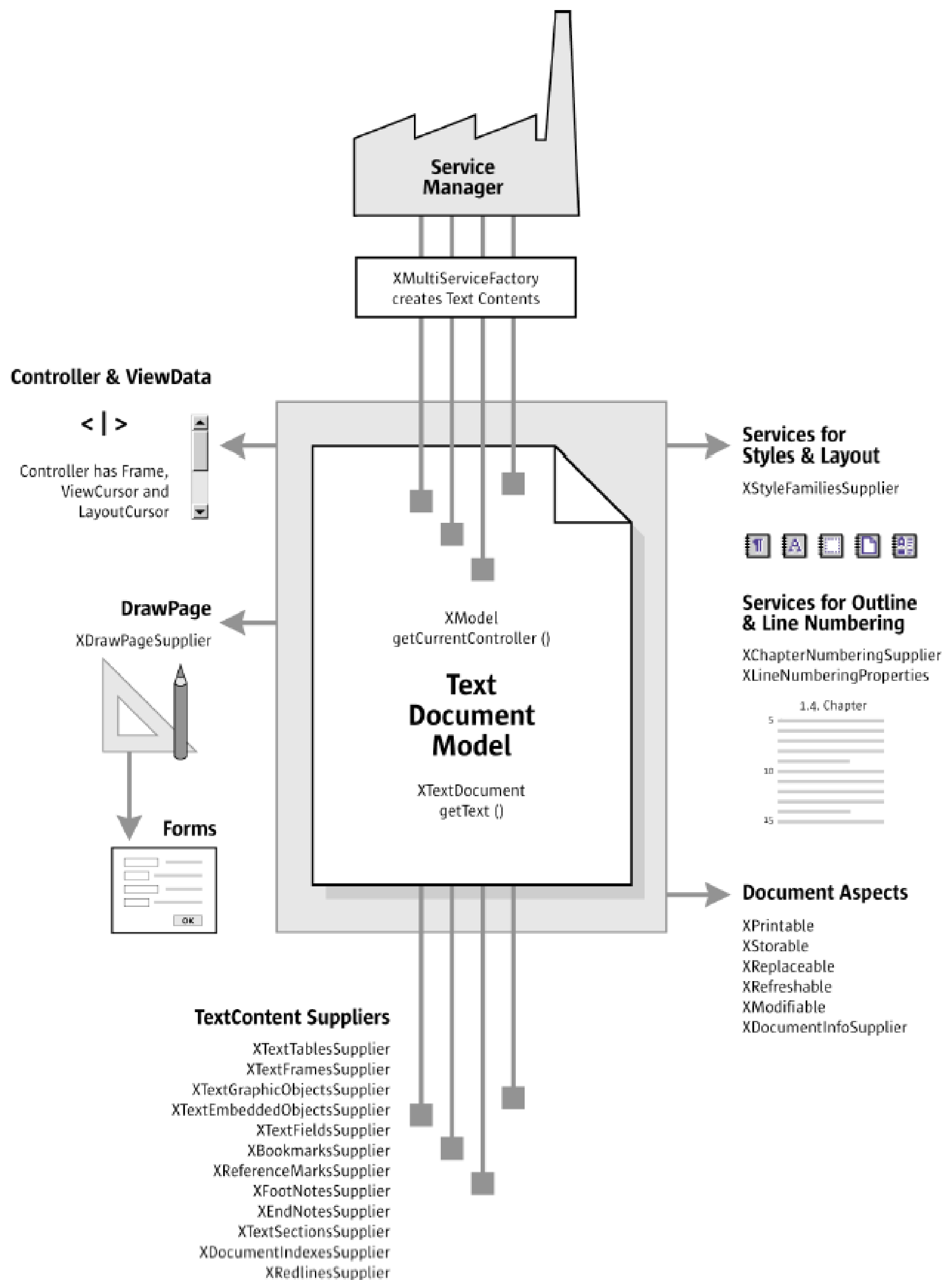

Figure 3.1: Text Document Model

The core of a text document model is the text. It consists of character strings organized in paragraphs and other text contents. In order to get the processing paragraph, the *XTextDocument* interface is needed. It provides the developer with a useful way to access the current text document with all of its components. To get *XTextDocument* from the interface *XFrame*, we need the *XModel*'s help:

```
//get interfacce xModel from xFrame, xController is the
//bridge between them

xController  = xFrame.getController();

xModel  =  xController.getModel();



//get xTextdocument from xModel with help of the function
//queryInterface

xTextDocument = (XTextDocument) UnoRuntime.queryInterface
(XTextDocument.class, xModel);
```

Both methods to get the interface are discussed in session 2.1.3. The *XModel* interface is accessed from the existent *XFrame* interface, but to get the *XTextDocument,* the *queryInterface()* provided method by *UnoRuntime* must be used.

The above method enables to get the current document processed by OpenOffice and this is also the document that the Add-On program is working on. There is another way to get other documents with its known URL from the current work. UNO support the developer to work with different documents at the same time. This method will be discussed later in session 3.3.4.

The *XTextDocument* interface is the key to work with an OpenOffice document. The *XTextDocument* plays a bridging role for the developer to get other components contained in the document. It can be a paragraph, a sentence, a word or just a picture or a chart. As we have mentioned about the two functions of this project earlier, the component we need to get from the *TextDocument* is the paragraph. But it is impossible for the developer to access it directly from the *XTextDocument*. He still needs helps from the other agent interfaces like *com.sun.star.text.XText* to access the text content of the document or the group of cursor interfaces. More details will be discussed in next session.

3.2.2 Model Cursor and View Cursor

There are two kinds of cursor in the OpenOffice structure: *model cursors* and *visible cursors*. *Visible cursors* are also referred to as *view cursors*.

The *View cursor* is visible on the screen. It enables the user to travel over the document to get information about the current layout, such as character, line number, view page, document page presented on the UI. The *com.sun.star.text.TextViewCursor* view cursor is the only one and it is based on *com.sun.star.text.XTextRange.* We can get it from the *XController* interface that is available from *XFrame* discussed in the previous chapter:

```
// the controller gives us the TextViewCursor query the view
// cursor supplier interface

XTextViewCursorSupplier xViewCursorSupplier = (XTextViewCursorSupplier) UnoRuntime.queryInterface( XTextViewCursorSupplier.class, xController);



// get the cursor

XTextViewCursor xViewCursor = xViewCursorSupplier.getViewCursor();
```

The *model cursor* allows the developer to move freely over the model by paragraphs, sentences or words. Every element here is suitable for a special cursor such as *ParagraphCursor*, *SentenceCursor*, *WordCursor* which are accessible from the *TextCursor*. The following codes show how to get them. At first, the *Text* service must be gotten from the *TextViewCursor*. The cursor is only a *XTextRange* and has therefore the *getText()* method. This *XText* interface is used to access the text content displaying on the screen from the view cursor.

```
XText xText = xViewCursor.getText();

// the text creates a model cursor from the viewcursor

XTextCursor xModelCursor = xText.createTextCursorByRange(xViewCursor.getStart());
```

Now the *XWordCursor*, the *XSentenceCursor* and the *XParagraphCursor* can be queried from the *XTextCursor*. Here is an example for *XParagraphCursor*:

```
XParagraphCursor xParagraphCursor = (XParagraphCursor) UnoRuntime.queryInterface(XParagraphCursor.class, xModelCursor);
```

3.2.3 Getting text content from paragraphs

To the first and the second function of this project, it is necessary to get the content of the current processing paragraph. From the *XParagraphCursor* interface, it is possible to take the content by scanning from the beginning to the end of the paragraph:

```
xParagraphCursor.gotoStartOfParagraph(false);

xParagraphCursor.gotoEndOfParagraph( true );

String mainString = xParagraphCursor.getString();
```

To execute the first function, all text contents of the paragraphs need to be gained and saved in an array so that it is easier to find the desired paragraph that resembles the processing paragraph. OpenOffice helps the developer to complete this task by using the *com.sun.star.container.XEnumerationAccess* interface. It enumerates all paragraphs in a text and returns the objects which support *com.sun.star.text.Paragraph*. The enumeration access can be gotten from *XText* by querying the UNO runtime environment. But the *XText* interface of the text content of the whole document must be accessed from *XTextDocument*.

```
XText xDocText= m_xTextDocument.getText();

XEnumerationAccess xEnumerationAccess = (XEnumerationAccess) UnoRuntime.queryInterface (XEnumerationAccess.class, xDocText);
```

The next step is to create an enumeration access of all paragraphs of the document.

```
// the enumeration contains all paragraph form the document

com.sun.star.container.XEnumeration xParagraphEnumeration = xEnumerationAccess.createEnumeration();
```

Now it is possible to iterate over the paragraphs of the text and create the text portions for each paragraph.

```
XTextContent xParagraph = null;

ArrayList xParagraphArray = new ArrayList();

//check if a paragraph is available

while ( xParagraphEnumeration.hasMoreElements() ) {

   //get the next paragraph

   xParagraph = (XTextContent)UnoRuntime.queryInterface

   (XTextContent.class, xParagraphEnumeration.nextElement());


   //we need the method getAnchor to get a TextRange -> to

   //manipulate the paragraph. This sText here is the content

   //of every Paragraph

   String sText = xParagraph.getAnchor().getString();

   if (xParagraphEnumeration.hasMoreElements()) {

       xParagraphArray.add(sText);
```

```
    }
}
```

All paragraphs except for the processing paragraph are saved in an *ArrayList*. The last *if*-statement makes sure that the processing paragraph is not saved in this list.

We only need the collection of the paragraphs in the document to compare with the processing paragraph later. But the *XEnumerationAccess* interface can do more than that. Every paragraph also has a *XEnumerationAccess* of its own. It can enumerate every single text portion that it contains. A text portion is a text range containing a uniform piece of information that appears within the text flow. A paragraph can have different formatted words or other contents. The text portion enumeration returns one *com.sun.star.text.TextPortion* service for each differently formatted string and for every other text content. They will be gained with the help from the *com.sun.star.text.TextRange* service. The function *getAnchor()* can return the interface *XTextRange* from *XTextPortion*.

3.3 Working with the OpenOffice.org GUI

This chapter introduces the way to work with the OpenOffice.org GUI. Concerning the functions of this project, it is necessary to create a window with a table form to display the information. In the first function, this information can be acquired after comparing the processing paragraph with the saved paragraphs. In the second function, the OpenOffice receives the information from the server after sending the request with processing paragraph. To set table form for the window, at first a dialog model must be created to be inserted in the window. Every component that we want to present on the window must be firstly added to this dialog, because the dialog control decides how to display the other components. The controls of other components must also be created and added to the dialog model by names. Therefore the dialog model is considered as a container of controls. After the dialog is executed, the inserted components will be presented on the screen.

A header showing the introduction and a *ListBox* displaying the information in the table form are the components supported by the OpenOffice UNO. After showing the information on the window, the *KeyListener* and the *MouseListener* are used to get events from the user to finish the defined task.

3.3.1 Dialog model

A dialog is known as a control container for other controls of the components which will be displayed on the window. All controls belonging to a dialog are grouped together logically. The dialog has the *com.sun.star.awt.UnoControlDialog* service that supports the *com.sun.star.awt.XControlContainer* interface. This interface is a container whose controls can be accessed by name. The hierarchy between a dialog and its controls can be seen in the *com.sun.star.awt.UnoControlDialogModel* dialog model which is a container of control models and therefore supports the *com.sun.star.container.XNameContainer* interface. This interface is used to insert the other created controls. The way to use this model will be introduced in this chapter.

The *XMultiComponentFactory* and *XComponentContext* interfaces are used to create a dialog model *UnoControlDialogModel*. While *XComponentContext* is available from Add-On instantiation, the *XMultiComponentFactory* must be gained from *XComponentContext*.

```
xMultiComponentFactory =
xComponentContext.getServiceManager();



//Now it is possible to create a dialog model

Object xUnoControlDialogModel =
xMultiComponentFactory.createInstanceWithContext(
"com.sun.star.awt.UnoControlDialogModel",
xComponentContext);



//The named container is used to insert the created
//controls into

XNameContainer xDlgModelNameContainer = (XNameContainer)
UnoRuntime.queryInterface(XNameContainer.class,
xUnoControlDialogModel);
```

The *XMultiServiceFactory* interface of the dialog model is a useful key when the developer begins to work with OpenOffice GUI. Through this interface, it is possible to create the controls of the other objects that are inserted later into the dialog, for example: in this project, the *ListBox* and the header need to be created and inserted in the dialog, so their controls must be added to the dialog container.

The component's control models can be created by using *XMultiServiceFactory* of the dialog model but they are not available until they are inserted into the dialog container by adding their control names to *XNameContainer*.

```
// The XMultiServiceFactory of the dialog model is needed to
// instantiate the controls

XMultiServiceFactory xMSFDialogModel =
(XMultiServiceFactory)UnoRuntime.queryInterface
(XMultiServiceFactory.class, xUnoControlDialogModel);
```

The *XUnoControlDialog* object dialog is also created by *XMultiServiceFactory* of the dialog model.

```
// create the dialog

Object xUnoDialog =
xMultiComponentFactory.createInstanceWithContext(
"com.sun.star.awt.UnoControlDialog", xComponentContext);



//get control of the dialog

XControl xDialogControl = (XControl)
UnoRuntime.queryInterface(XControl.class, oUnoDialog);



//The scope of the control container is public

XControlContainer xDlgContainer = (XControlContainer)
UnoRuntime.queryInterface(XControlContainer.class,
xUnoDialog);



// link the dialog and its model

 XControlModel xControlModel = (XControlModel)
UnoRuntime.queryInterface(XControlModel.class,
xUnoControlDialogModel);

xDialogControl.setModel(xControlModel);
```

After creating the dialog model and its control or its window, it is essential to set the properties of this dialog like its size and attributes. It should be done before the other objects are inserted into it. There are two ways to set the properties of the components in UNO, which were discussed earlier in session 2.1.4. The *XMultiPropertySet* is used for this dialog model.

```
XMultiPropertySet xMultiPropertySet = (XMultiPropertySet)
UnoRuntime.queryInterface(XMultiPropertySet.class,
xDlgModelNameContainer);


xMultiPropertySet.setPropertyValues(String[] PropertyNames,
String[] PropertyValues);
```

The properties of the dialog model can be found at [3], for example, there are some properties used in this project: *Height, Moveable, Name, PositionX, PositionY, Step, TabIndex, Title* and *Width.* Now the other component can be created and inserted into the dialog. After inserting of all components, the dialog will automatically create its window to present on the screen.

3.3.2 Create and insert FixedText and ListBox to Dialog

Displaying the information on the screen is almost the same in both functions of the project. Only in the second function, the information must be displayed in table form which is not supported by UNO, meaning the information must be formatted before it is added to a *ListBox* provided by UNO. Because OpenOffice.org Writer receives the response from the server in the format of *String[]* we can also create new strings with the same length from elements of *String[]*.

For this purpose the new font will be set as a property of the header or the *ListBox*. After setting the format of the information and changing the font, it is ready to create and insert the header or the *ListBox* to the dialog model.

```
//format the header to be looked like the title of the
//table. The function setFormat is written to create a text
//to insert it in the listbox

String label = setFormat("Nr", "Name Of Documents",
"Comment", "Percent");


//create the header model by using  UnoControlFixedTextModel

Object oFTHeaderModel = xMSFDialogModel.createInstance(
"com.sun.star.awt.UnoControlFixedTextModel");
```

```
//get property interface of the header (FixedTextModel)

XMultiPropertySet xFTHeaderModelMultiPropertySet =
(XMultiPropertySet)
UnoRuntime.queryInterface(XMultiPropertySet.class,
oFTHeaderModel);



//set properties for the header

xFTHeaderModelMultiPropertySet.setPropertyValues(

                 new String[] {"FontDescriptor", "Height",
"Label", "Name", "PositionX", "PositionY", "Width"},

                 new Object[] { font, new Integer(8), label,
"HeaderLabel", new Integer(0), new Integer(3), new
Integer(250)});



//add the model to the NameContainer of the dialog model

xDlgModelNameContainer.insertByName("Headerlabel",
oFTHeaderModel);
```

The developer must be careful whenever he tries to insert a component into the dialog by name. The name must be unique in the *(XNameContainer) xDlgModelNameContainer*. In case there are many components to be inserted into the dialog, it stands a high chance that several components have the same name. The best solution to this problem is to write a function only to alter the duplicated name. For example, the function can add more space character to it.

After receiving the information from the server, the response is formatted and saved in a *String[] itemList* which will be set in the properties of the *ListBox* later by its instantiation. The code below shows how to create and insert the *ListBox* into the dialog:

```
// create a list box model by using the MultiServiceFactory
//of the dialog model

Object oListBoxModel = xMSFDialogModel.createInstance
("com.sun.star.awt.UnoControlListBoxModel");



//get property interface of the ListBoxModel
```

```
XMultiPropertySet         xLBModelMultiPropertySet          =
(XMultiPropertySet)
UnoRuntime.queryInterface(XMultiPropertySet.class,
oListBoxModel);



// Set the properties at the model, we must keep in mind to
// pass the property names in alphabetical order!

xLBModelMultiPropertySet.setPropertyValues(

                   new      String[]            {"Dropdown",
"FontDescriptor",    "Height",    "MultiSelection",    "Name",
"PositionX",    "PositionY",    "Printable,    "StringItemList",
"Width" } ,

                   new  Object[]  {Boolean.FALSE,  font,  new
Integer(height),      Boolean.FALSE,      "ListBox",      new
Integer(posX),  new  Integer(posY),  Boolean.TRUE,  itemList,
new Integer(width)});



// add the model to the NameContainer of the dialog model

xDlgModelNameContainer.insertByName("ListBox",
xLBModelMultiPropertySet);
```

More properties of *FixedTextModel* and *ListBoxModel* will be found in [4] and [5]. It is necessary to add *KeyListener* and *MouseListener* to the *ListBox* so that the *ListBox* can manage events from the user. In this project, the user has two possibilities to choose a paragraph: to insert it in the document in the first function and to choose the document to open in the second function.

```
//ask the XControlContainer of the dialog to get control of
//the ListBox by name

XControl xControl = xDlgContainer.getControl("ListBox");



//get the window containing the ListBox

XWindow          xListBoxWindow          =          (XWindow)
UnoRuntime.queryInterface(XWindow.class, xControl);
```

```
//add KeyListener and MouseListener to the window to handle
//the events from the user

xListBoxWindow.addKeyListener(new XKexListener());

xListBoxWindow.addMouseListener(new XMouseListener());
```

The implementation of both listeners will be discussed in the next session. Now the window of the dialog must be created and executed to present the dialog on the screen. So the object *ToolKit* must be created to access the *XtoolKit* interface. This interface specifies a factory interface for the window toolkit. This is similar to the abstract window toolkit (AWT) in Java.

```
Object                         toolKit                          =
xMultiComponentFactory.createInstanceWithContext
("com.sun.star.awt.Toolkit", xComponentContext);



//get XToolKid from its object

XToolkit           xToolkit           =           (XToolkit)
UnoRuntime.queryInterface(XToolkit.class, toolKit);



//creates a "child" window on the screen. If the parent is
//NULL, then the desktop window of the toolkit is the
//parent.

xWindowParentPeer = null;

xDialogControl.createPeer(xToolkit, xWindowParentPeer);
```

It is possible to get the *Xdialog* interface now. By executing this interface, the dialog together with the inserted components like *FixedTextModel* and the *ListBoxModel* will be present on the screen.

```
//get XDialog

XDialog            xDialog            =            (XDialog)
UnoRuntime.queryInterface(XDialog.class, xDialogControl);



//execute the XDialog

xDialog.execute();
```

3.3.3 KeyListener, KeyHandler and MouseListener in UNO

In this project, *com.sun.star.awt.KeyListener* and *com.sun.star.awt.MouseListener* are used to get events for the dialog. It enables the user to choose a paragraph or a document to open by mouse click or using the keyboard. They must be added to the window containing the processing components. In this project the *(XWindow) xListBoxWindow* is the *container window* of *ListBox* object. The provided functions of *KeyListener* and *MouseListener* are similar to *KeyListener* and *MouseListener* (AWT) in java.

Nevertheless, UNO still provides another interface *com.sun.star.awt.XKeyHandler* to handle with key events. The difference is that the handler consumes the event. To add a key handler, the developer must use the *addKeyHandler()* method from the *XUserInputInterception* or *XExtendedToolkid* interface. Both of these interfaces are implemented by the *com.sun.star.frame.Controller* controller.

The KeyHandler is used in this project to get events from the user to decide which function should be started. The user must press F8 for the first function or F9 for the second function. This *KeyHandler* must be added to the *XExtendedToolkit* of the main frame. The *XExtendedToolkit* is an extension of the *XToolkit* interface. It basically provides access to three event broadcasters used for instance in the context of accessibility. The first event broadcaster is used to get the set of currently open top-level window; the second event broadcaster informs its listeners about key events and the last event broadcaster sends events on focus changes of all elements that can have the input focus. The following code introduces the way to get *XExtendedToolkit* from the existent *XFrame*:

```
//get the component window

XWindow xWindow = xFrame.getComponentWindow();



XWindowPeer MyWindowPeer = (XWindowPeer) UnoRuntime.queryInterface (XWindowPeer.class, xWindow);



XToolkit MyToolkit = MyWindowPeer.getToolkit();



XExtendedToolkit MyExtToolkit = (XExtendedToolkit) UnoRuntime.queryInterface (XExtendedToolkit.class, MyToolkit);
```

```
MyExtToolkit.addKeyHandler (new KeyListenre());
```

Nonetheless, the *KeyHandler* is not always a good choice because the controller may get disposed. Still, the document may be opened, so the handler won't be called anymore. Hence, the *KeyHandler* is substituted for the *KeyListener* to be added to the *ListBox*. The functions of *KeyHandler* are similar to the functions of *KeyListener*.

3.3.4 Open a document in a new window

To open a document from a given URL, it is necessary to get the desktop service that was explained in the session 2.2.2. We can access this service by using the *XMultiComponentFactory* and *XcomponentContext* interfaces.

```
Object desktop = xMultiComponentFactory.createInstanceWithContext(
"com.sun.star.frame.Desktop", xComponentContext);


// Query the XComponentLoader interface from the desktop Object.

XComponentLoader xComponentLoader = (XComponentLoader) UnoRuntime.queryInterface(XComponentLoader.class, desktop);

//set properties for ComponentLoader

PropertyValue[] myProperties = new PropertyValue[1];

myProperties[0] = new PropertyValue();

myProperties[0].Name = "Hidden";

myProperties[0].Value = new Boolean(false);


XComponent xComponent = xComponentLoader.loadComponentFromURL(

        source_File,                // Url

        "_blank",                   // TargetFramName

        0,                          // Is ignored
```

```
        myProperties);              // Special properties

}

catch(Exception e) {

}
```

The *TargetFrameName* "_blank" is set to load the new document in a new window. More properties or *TargetFrameName* of the *ComponentLoader* will be found in [6].

3.4 OpenOffice as a web client in web service

Java Development Kit 6 (JDK 6) imports the Java Web Services Developer Pack (JWSDP) to facilitate the implementation of web service [8]. The web service has two sides: web client and web server. But OpenOffice only plays the role as web client in this project. The following code shows how to define a web service:

```
// name of this web service is "Search Machine" with the URL

@WebService(name = "SearchMachine", targetNamespace = "http://OpenOfficeClient/service/")

@SOAPBinding(style = SOAPBinding.Style.RPC)

public interface SearchMachine {

  @WebMethod

  @WebResult(partName = "return")

  // arg0 is the input and also the processing paragraph

  // the return value is the list of document with their

  //information


  public String[] askForDocumentList(

       @WebParam(name = "arg0", partName = "arg0")

       String arg0);

  }

}
```

Then the Client of the web service can be implemented like this following code:

```
// The main string here is the list of documents with their
// information

public String[] sendAndGetMessage(String paragraph) {

SearchMachineService service = new SearchMachineService();

// this method to get the service Search Machine

SearchMachine                    searchMachine                    =
service.getSearchMachinePort();

String[]                         mainString                       =
searchMachine.askForDocumentList(paragraph);

return mainString;

}
```

After getting the response from the server as a *String[]*, a dialog must be created to present this information on the screen. The URL of every document is also saved in this String[].

Chapter 4

Conclusion and Outlook

To develop an OpenOffice.org application, it is essential to learn about UNO and the OpenOffice.org Application Environment. It provides not only an overview about the structure of OpenOffice.org but also the introduction of how the components in UNO are organized.

The aim of this project can only be attained when it is possible to open the pdf-document with OpenOffice, either by itself or by using another application such as Acrobat Reader.

Programming with OpenOffice can be confusing at the beginning due to a huge amount of objects, interfaces and properties. Moreover, the API is not stable among different versions of OpenOffice and many problems still exists whilst programming with OpenOffice, for example: the *XMultiServiceFactory* in the old version is substituted for the *XMultiComponentFactory* in the new version. It is also confusing that the two interfaces *XKeyHandler* and *XKeyListener* have the same task which is to manage the key events, but they are spilt into two different interfaces. Furthermore, related research is still in developing phases, so it is more time-consuming to find the solutions to the problems. Another difficulty is that OpenOffice does not support pdf-document. We are unable to import pdf-documents in *OpenOffice* Writer and it is also not possible to program to open pdf-document with an external application such as Acrobat Reader. These functions should be developed in next OpenOffice.org version. It is useful if the user can start another application from OpenOffice.org. Up to now, only the OpenOffice Draw and the OpenOffice Impress can import pdf-document. There is an on-going project trying to make it possible in OpenOffice Writer. More information can be found in [7].

However, OpenOffice UNO provides many useful services and interfaces to work with the content of OpenOffice Writer. The interfaces of the *view cursor* service and the *model cursor* service enable to get and set any text content in the document. It can not only be a paragraph but also a sentence, a word or text contents with different characters. Therefore future work with OpenOffice can aim to analyze semantic text by using useful interfaces such as the *XWordCursor*, the *XSentenceCursor* or the *XParagraphCursor* which can be queried from the *XTextCursor.*

Chapter 5

Bibliography

[1] Sun Microsystems, Inc, OpenOffice.org 2.3 Developer's Guide 2007. http://api.openoffice.org/docs/DevelopersGuide/DevelopersGuide.pdf. Date Retrieved: January 1st, 2009.

[2] The *OpenOffice.org Software Development Kit.* http://download.openoffice.org/3.0.0/sdk.html. Date Retrieved: January 1st, 2009.

[3] The properties of the service *UnoControlDialogModel.* http://api.openoffice.org/docs/common/ref/com/sun/star/awt/UnoControlDialogModel.html. Date Retrieved: January 1st, 2009.

[4] The properties of the service *FixedTextModel.* http://api.openoffice.org/docs/common/ref/com/sun/star/awt/UnoControlFixedTextModel.html. Date Retrieved: January 1st, 2009.

[5] The properties of the service *ListBoxModel* http://api.openoffice.org/docs/common/ref/com/sun/star/awt/UnoControlListBoxModel.html. Date Retrieved: January 1st, 2009.

[6] The properties of the interface *XComponentLoader* http://api.openoffice.org/docs/common/ref/com/sun/star/frame/XComponentLoader.html. Date Retrieved: January 1st, 2009.

[7] OpenOffice.org Ninja. The project for importint pdf-document in OpenOffice. http://www.oooninja.com/2008/06/pdf-import-hybrid-odf-pdfs-extension-30.html. Date Retrieved: January 1st, 2009.

[8] WebService in Java. http://www.theserverside.de/webservice-in-java/. Date Retrieved: January 1st, 2009.