# Projektarbeit

## Development of a Java based framework for the management of multiple desktops

*Moritz Plößl*

Juli 2007

Supervisors: Prof. Dr. Andreas Dengel
Dipl. Inf. Sven Schwarz

Knowledge-based Systems Group
Prof. Dr. Andreas Dengel

University of Kaiserslautern
Computer Science Department

Moritz Plößl
Kleestrasse 33
67659 Kaiserslautern

Kaiserslautern, den 30 Juli 2007

## Erklärung

Hiermit erkläre ich, dass ich die Arbeit selbständig verfasst und nur die angegebenen Hilfsmittel verwendet habe.

(Moritz Plößl)

# Contents

**Abstract**

This document describes the multidesktop framework called Mydesk which was developed for the Mymory research project. It is a multidesktop frame work which is implemented mostly in Java and runs under Microsoft Windows. It has the ability to provide a various number of desktops. Also, it provides an event interface and interfaces for application to support the storing and recovering of desktops. This framework is also intended to be easily portable to other operating systems.

# Chapter 1

# Introduction

For the Mymory research project a tool is needed which can detect context switches and assists the user to switch to other working contexts. There is a similarity between contexts and desktops. A user's context can be defined by the applications and the mostly documents used by the user. Thus a desktop can be associated with a context. By courtesy of this association between context and desktop it is possible to use a multidesktop system or desktop manager as a tool to manage user contexts and detect context switches.

Such a multidektop system has to comply with various conditions. It has to be easily integrated into the software infrastructure of the mymory research project which is mostly written in Java. The same multidesktop system should be run on different operating systems. It has to have the ability to deal with a large number of desktops. This also includes that it can provide access to desktops (contexts) which are very seldom used. It has to have an event like system to detect context (desktop) switches of the user. To Provide a user interface. And at last it has to be extensible to accommodate new requirements and needed changes. For the reason that there does not exist a multidesktop system which fits this requirements a new system has been developed.

The decision to use Java was made because this makes it easier to integrate the new framework in the Mymory infrastructure. As not all of the needed functionality is available to Java, some functionalities have to be implemented in another programming language to gain access to the operating systems windowing functionality. Because the main application logic is implemented in Java only the small parts which are operating system dependent have to be (re-)implemented to port the framework to another operating system. This document describes the implementation of a multidesktop framework which fits most of this requirements. It is called MyDesk.

# Chapter 2

# Overview

In this chapter we will give a short overview over the functionality of the various Java classes in the MyDesk framework, as well as their interplay.

Detailed information about each of these classes can be found after this chapter (in chapter 3 and 4).

The class MultiDesktop provides the main functionality of a desktop manager. It also is the entry point to the multidesktop / multicontext system. A MultiDesktop manages many Desktops and, hence, many contexts because of the association of Desktop and context. Each Desktop arranges various DesktopsWindows which are representations of the application windows covering the user actions and Data. Each DesktopWindow has one DesktopWindowsState which stores information about the current state of that window. Such a window state holds all information necessary to restore the window as it currently is. For example, the currently viewed document(s), important selections of objects the state of a tree (collapsed/expanded nodes), toggle modes of switches (e.g., insert on/off), and so on.

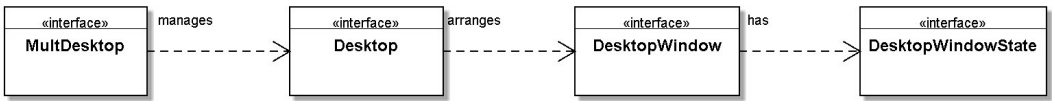



Figure 2.1: UML diagram of the core interfaces of MyDesk

No currently available operating system provides a multiple desktop architecture. Hence, a multi-desktop system managing up to hundreds or thousands of desktops can not really move windows to some (virtual) place. Instead, MyDesk controls their *visibility* according to internal information. To be able to manipulate the visibility of windows it is necessary to access the windowing functionality

of the operating system. The interfaces DesktopNativeInterface and WindowNativeInterface build an abstraction layer. This layer makes the application logic independent from the implementation of the bridge to the windowing system. This makes it easier to port the framework to another operating system, because only these interfaces have to be implemented again and the builder of them has to be adapted.

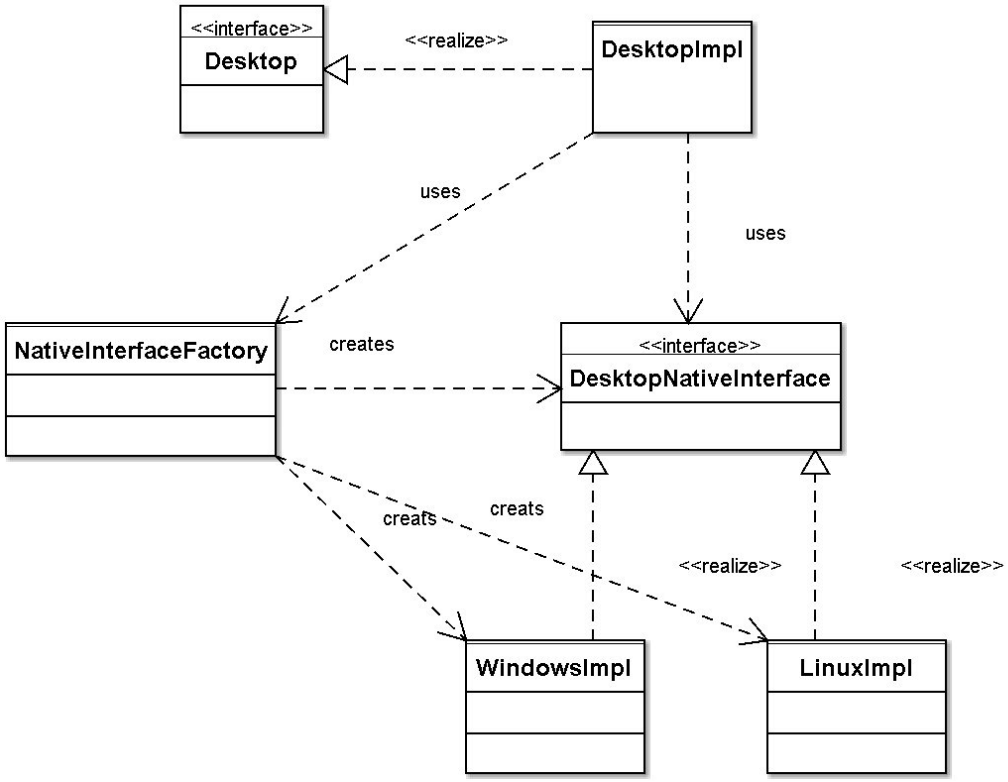



Figure 2.2:

The ApplicationModule, the ApplicationsDispatcher and the ApplicationRegistrationService build the store and recover module which is used to store a Desktop to a persistent medium and to recover from it later on. The ApplicationDispatcher is used by a Desktop to store or recover itself. It uses the ApplicationModule which implements the special knowledge to connect to the right application type and retrieve the information which can only be provided by the application itself.

During recovering a desktop it is necessary to start applications and, maybe, open some document in that application. How to start an application differs from

one operating system to another. For that reason there exists another abstraction layer. This layer is realised by the interface StartApplicationNativeInterface. After some application has started it registers itself to the ApplicationRegistrationService to enable MyDesk to set or request its internal state (viewed document(s), important selections of objects).

The MultiDesktop also offers the opportunity to add MyDeskEventListeners which listen to MyDesk Events. The MyDesk framework sends such an event if one of the following actions occurs:

- Switch to another desktop
- Window added, if a window has been moved from one desktop to another. Currently, newly emerging windows (e.g., by starting another application) are not yet registered.
- Window removed, if a window has been removed from a desktop. Currently, this event only will be send if the window was removed during moving it to another desktop
- Windows added or removed, send during synchronising a desktop

# Chapter 3

# The interfaces of MyDesk

## 3.1 The parts of the MyDesk system core

The core of the MyDesk API are the interfaces defined in the package de.dfki.km.mydesk and their implementations (placed in de.dfki.km.mydesk.impl) together with the nativeInterfaves and theirs specific operating system implementations. They build a multidesktop system which is extensible and provides an event interface and interfaces to store and recover Desktops.

### 3.1.1 DesktopWindow

For a better understanding we distinguish between two types of windows. First there are the windows which are created by an application and can be accessed by the user. These windows will be called native windows or simply windows. The other type of window is the DesktopWindow, which is the representation of a native window in the MyDesk system. According to this a DesktopWindow is a representation of a native top-level window in Java which is displayed on the screen by the windowing system of the operating system.

For every toplevel window on the screen the MyDesk system will create a DesktopWindow with a unique identifier. This identifier is used to determine the native windows. With the help of the nativeInterfaces and the unique identifier it is possible to control the visibility and size of the native top-level window. This means that every change of the visibility done by calling the corresponding method of the DesktopWindow will affect the native top-level window. However, if the user directly changes the visibility of the native window (minimising, maximising, etc.) this will not influence the DesktopWindow. Therefore it is necessary to update the DesktopWindow before using it in some way, to make sure that the DesktopWindow is consistent to the state of its native window. But this can only happen to

DesktopWindows of the active desktop, because the native windows of all other desktops are hidden and thus not accessible by the user. The DesktopWindow interface describes the functionality of a DesktopWindow.

### 3.1.2 DesktopWindowState

While the DesktopWindow is a representation of a native window, the DesktopWindowState is an information container for informations about the DesktopWindow. Thus each DesktopWindow has one DesktopWindowState that contains information about this window like the visibility parameter and the name of the executable which is associated with the native window. If the application, to which the window belongs, supports the MyDesk system, the DesktopWindowState contains further information. This information indicates how to connect to the application and how to start the application during recovering a desktop.

### 3.1.3 Desktop

Concerning the windows we distinguish between two different types of desktops. One is the desktop which consists of the windows that are displayed on the screen and will be manipulated by the user. We will call it user desktop. The other one is the representation of this user desktop in the MyDesk system, which is not all the time consistent with the desktop visible for the user (see below). This desktop will be called "Desktop" like the interface which defines its functionality in the MyDesk system. A Desktop is a unit which displays, arranges and controls several DesktopWindows, thus a Desktop consists of an bundle of DesktopWindows.

Many Desktops belong to a multidesktop system, but only one of these Desktops can be active at a time (this Desktop will be called "active Desktop"). This means that only the windows of one desktop are visible and can be accessed by the user at the same time. The windows (DesktopWindows and native Windows) of the other Desktops are hidden. It is important to know that the "active Desktop" in the MyDesk system and the user desktop (which means the configuration of all visible windows on the screen) are not always synchronised. This caused that the active Desktop can contain DesktopWindows whose associated native windows do not exist anymore (for example when the user has closed the window). Furthermore there can be some windows on the screen which have no DesktopWindow representative yet. The reason for this is that the desktop will be only updated if either the user or someone else switches to another desktop, or if the user or some service synchronises the user desktop and the active Desktop explicitly. Whenever another part needs proper information about the active Desktop or by one of its Desktop-

Windows it is necessary to update the desktop to enable the synchronisation of the Desktop with the real desktop.

Normally only the active Desktop can run out of synchronisation, because only its windows can be accessed by the user. Every Desktop which has at least one DesktopWindow has a specific window which is called foreground window. This window owns the user focus and is on top of all other windows. Thus it is the first window in the z-order.

### 3.1.4 MultiDesktop

The MultiDesktop interface describes a desktop manager. It controls the Desktops and is responsible for manage the movement of windows between different Desktops. The MultiDesktop is also the component where the EventListener and new Desktops can be added to the system. User interface components like tray icons and the global ShortCutNativeInterface implementation use the MultiDesktop to enable the user to control the MyDesk system. It is important that only one instance of implementation exists for the MultiDesktop interface, because it will cause trouble if there are multidesktop systems in a multidesktop system. This is not a problem for the system itself but for the user who will lose the overview of the windows.

MultiDesktop is also the entry point for gaining access to the individual Desktops, DesktopWindows and DesktopWindowStates objects. Most of the application logic will be implemented in the realisation of the MultiDesktop interface, like the organisation of the desktops etc.

According to this a Desktop could be seen as a container of DesktopWindows, which provides methods to access DesktopWindows, like the foreground window. In addition functionality is provided to change the state of all windows belonging to the Desktop like hiding and showing all windows.

### 3.1.5 WindowVisibility

The window visibility enumeration lists the values which the visibility state of a DesktopWindow can have. The potential values are:

0 hidden: the window is not seen on the screen
1 normal: the normal mode which means the windows is neither maximized nor minimised
2 minimised: the window is not visible but shown in the task bar
3 maximised: the window fills the whole screen and and can not be moved

To use the WindowVisibility inside of methods of the native Interfaces use the toString method of this enumeration to get the value of the WindowVisibility.

## 3.2 The native interfaces

The interfaces in the package de.dfki.km.mydesk.nativeInterfaces define the functionality which is needed by other MyDesk components. This functionality is provided by the operating system and its windowing system only. For this reason the implementation of these interfaces have to be realised with techniques which give access to the functions and operations of the windowing system and the operating system. To port the MyDesk system to another operating system, is is only necessary to write a new implementation of the interfaces DesktopNativeInterface, WindowNativeInterface and StartApplicationNativeInterface for this operating system and to suite the NativeServiceFactory to the operating system.

The interfaces of the nativeInterface package are divided in a way that every interface is only needed by one component of the MyDesk system.

### 3.2.1 NativeServiceFactory

The NativeServiceFactory class is the central entry point to gain to access the interfaces of the nativeInterface package. It defines a factory which creates the appropriate instances of the operating system depending implementations of the interfaces DesktopNativeInterface, WindowNativeInterface and StartApplicationNativeInterface. This will have the effect that no changes have to be done in other parts of the system expect in the implementation of the NativeServiceFactory, if the MyDesk system should run on a new operating system.

### 3.2.2 DesktopNativeInterface

The DesktopNativeInterface interface provides the functionality which is needed for the implementation of the Desktop interface to realise its functionality. It defines a method which returns all visible application windows in z-order (the top most window first) and methods to get and set the foreground window.

### 3.2.3 WindowNativeInterface

This interface provides the functionality which is needed for the implementation of DesktopWindow interface. It contains methods to get and set the visibility informations and also the definition of methods to get information about the application to which this native window belongs.

### 3.2.4 StartApplicationNativeInterface

The StartApplicationNativeInterface interface is used for the implementation of the ApplicationModule interface to start application during recovering a desktop. It contains three different methods to run applications.

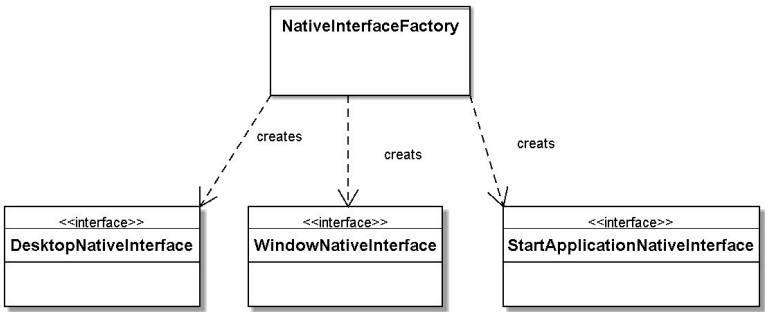



Figure 3.1: UML diagram of the nativInterfaces

### 3.2.5 Using the native interfaces

The Interface TrayIconNativeInterface and ShortCutNativeInterface are special cases because the do not provide functionality for some implementations of an interface of the MyDesk system. It is more likely that they use and control the implementation of the mydesk system.

How to get the identifier of the foreground window with the help of the native interfaces

```
Service service = new ServiceImpl();
DesktopNativeInterface ndesktop = service.getDesktopNativeInterface();
String windowID = ndesktop.getForegroundWindow();
```

Now we are able to get the visibility and the geometry of this native window

```
WindowNativeInterface nwindow = service.getWindowNativeInterface();
WindowVisibility win_vis = nwindow.getVisibility(windowID);
Rectangel geometry = nwindow.getGeometry(windowID);
```

But normally there is no need to use the native interfaces directly, expect for a new implementation of the Desktop or the DesktopWindow interface. The shown way to get the identifier of the foreground window only works for the active Desktop, but if the foreground window identifier of a window which is places on a not

active desktop is requested, the `getForegrundWindow` method of the Desktop object has to be used.

To port the MyDesk system to another operating system the following tasks have to be done.

1. Implement the there interfaces DesktopNativeInterface, WindowNativeInterface and StartApplicationNativeInterface for this operating system
2. Adjust the implementation of the Service interfaces to create the correct instances for this operating system.
3. Implement new user interface components to realise global shortcuts and tray icons to enable the user to do easy desktop switches.

## 3.3 The MyDesk event system

MyDesk provides one EventListener interface called MyDeskEventListener. It could be used by calling the addEventListerner method of the MultiDesktop interface. The MyDesk system differentiates between two types of events. Both types are subclasses of the abstract class MyDeskEvent. The first type are events raised by a MultiDesktop event source, the other type is raised by the Desktop event sources. If the event source belongs to the type MultiDesktop then an event object of the type MyDeskMDEvent is triggered. Otherwise the event source is a realisation of the Desktop interface and the event object type is MyDeskDesktopEvent. If the action ‘desktop switching’ and "window moving" occur, a MyDeskMDEvent is triggered. MydeskDesktopEvents are triggered if windows are added or removed from a Desktop. An object of the type Desktop has no method to add an EventListener. It would use its reference to its hosting system to get the added EventListerners. This has the advantage, that there is no need to care how a new desktop gets the registered Eventlisterners. To choose the type of event in the implementation of the MyDeskEventListerner interface you should look at the class type of the event.

```
public void myDeskActionPerformed(MyDeskEvent event)
{
  if(event.getClass() == MyDeskMDEvent.class)
  {
    //handle the event from the Multidesktop
  }
  if(event.getClass() == MyDeskDesktopEvent.class)
```

```
  {
    //handle event from a Desktop
  }
}
```

The attributes of the MyDeskMDEvent are:

- Source Desktop: The source desktop of the action. The active desktop by switching or the owner of a DesktopWindow before a moving the windows to another desktop
- Destination desktop: The destination of the action
- Action: The name of the action "move" or "switch"
- Position: The position of the destination desktop if the action this "switch" otherwise -1
- Window: Reference to the window which is moved, if the action is "move" otherwise null

The MyDeskDesktopEvent object has the following attributes:

- Source: The desktop which rise the action
- Action: addwindow, removewindow, addedorremoved
- Window: The reference to the DesktopWindow if action is "add" or "remove" null if windows "addedorremoved"

## 3.4 Storing and recovering of Desktops

A very essential component of the MyDesk system is the store/recover part. Its task is to provide the possibility to store a desktop. This means that this desktop can be loaded (restored) later on, maybe after a restart of the computer. After restoring a desktop, it should look the same way as it looked at the moment it was stored. This can only be done in a good and reliable way, if the applications support the MyDesk system. In that case the MyDesk system can get information about the application's internal state, such as the internal window configuration, opened documents and so on. This information will be requested during the storing of a desktop.

To recover the desktop MyDesk has to connect to the application to set the internal state which was saved by MyDesk before. For these both procedures requesting (getting) and setting the internal state of an application, it has to provide

an interface which has to be accessible from outside the application. To enable MyDesk to get the necessary information about an application the application has to register itself to the MyDesk system. For the procedure of getting, setting the internal state and the registration we will use XMLRPC. The decision to use XMLRPC was made, because there are many implementations in different languages. If there does not exist an implementation, it is not difficult to implement a XMLRPC service for this special case. But the architecture of MyDesk also allows other implementations of the registration and the communication with an application.

### 3.4.1 The ApplicationDispatcher

The ApplicationDispatcher decides which ApplicationModule should be used during storing or recovering to get or set the special application information. The implementation also decides in which way the Desktop should be stored (file, database, RDFStore etc.). Thus it is necessary to modify the implementation of the ApplicationDispacher to add the support for every new application type which should support MyDesk store recover mechanism.

### 3.4.2 The ApplicationModule

The ApplicationModule has the knowledge of the application type and how to get and set information about the internal state of the application. For every application type which supports MyDesk a new implementation of this interface has to be done.

### 3.4.3 The ApplicationRegistrationService

The ApplicationRegistrationService is the service in which applications that support MyDesk can register themselves to the MyDesk system. The ApplicationRegistrationService then searches a window which matches the transmitted information. If no such window can be found the registration process will fail. To find the matching window which belongs to the requesting application the following informations are needed.

1. The visibility of application window, on this it depends on which desktop the search for the window has to be started.
2. The Title of the application window is the first criterion for the search.
3. The geometry of the application window is the second criterion which has to match. But in the case of the geometry we have to consider if the window is maximised, minimised or normal. If the transmitted geometry has the size of a maximized window we have to look if the DesktopWindow's matching title

is also maximized. Otherwise if the transmitted geometry's size is not the size of a maximized window we have to compare the transmitted geometry with the one which is retrieved by the DesktopWindow object.

4. The service name (handler) is needed to connect to the application and to identify the appropriate ApplicationModul implementation.

5. The XML-RPC port to connect to (for inquire the windows internal state later on)

6. The command how to start the application during recovering.

Below follows an example how to save the active desktop using the MyApplicationDispacher which implements the ApplicationDispatcher interface:

```
ApplicationDispatcher ap = new MyApplicationDispatcher();
MultiDesktop multiDesk = MyDeskMultiDesktop.getMultiDesktop();
Desktop desk = multidesk.getactiveDesktop();
desk.store(ap);
```

How and where the desktop is be stored will be decided by MyApplicationDispatcher. It is very easy to change the way of storing a Desktop because it only depends on the implementation of the ApplicationDispatcher and is independent form the implementation of the multidesktop System.

## 3.5 MyDesk Package overview

**de.dfki.km.mydesk** The interfaces in this package describe the functionality of the core part of the MyDesk system. They are MultiDesktop, Desktop, DesktopWindow and DesktopWindowState.

**de.dfki.km.mydesk.impl** Containing the implementation of the core interfaces placed in the de.dfki.km.mydesk package. This implementation realises a desktop manager written in Java.

**de.dfki.km.mydesk.event** Defines the EventListener and the event objects which are in use of the implementation of the MyDesk system.

**de.dfki.km.mydesk.nativeInterfaces** The definition of the functionality which is provided by the windowing system of the operating system and by the operation system itself. These interfaces have to be implemented in a way that allows access

to the windowing system.

**de.dfki.km.mydesk.nativeIntervaces.winimpl / com / com.events** Winimpl containing wrapper classes which implement the nativeInterfaces and use com4j to access the windowing system. The subpackages com and com.events containing the interfaces which are realised by COM components. The components themselves are accessed via COM4J.

**de.dfki.km.mydesk.store / impl** The store package holds the interfaces describing the functionality to store and recover Desktops. The subpackage impl holds some sample implementation of the interfaces defined in the store package.

**de.dfki.km.mydesk.registerapplication / impl** Containing the definition of a service which allows applications that supports the MyDesk system to register themselves to the multidesktop system.

**de.dfki.km.mydesk.visualisation** Contains some classes which visualise the running MyDesk multidesktop system. In a JWindow the desktops and their DesktopWindows are displayed in a tree structure. It is also possible to move windows from one desktop to another by drag and drop, and to get further information about a DesktopWindow.

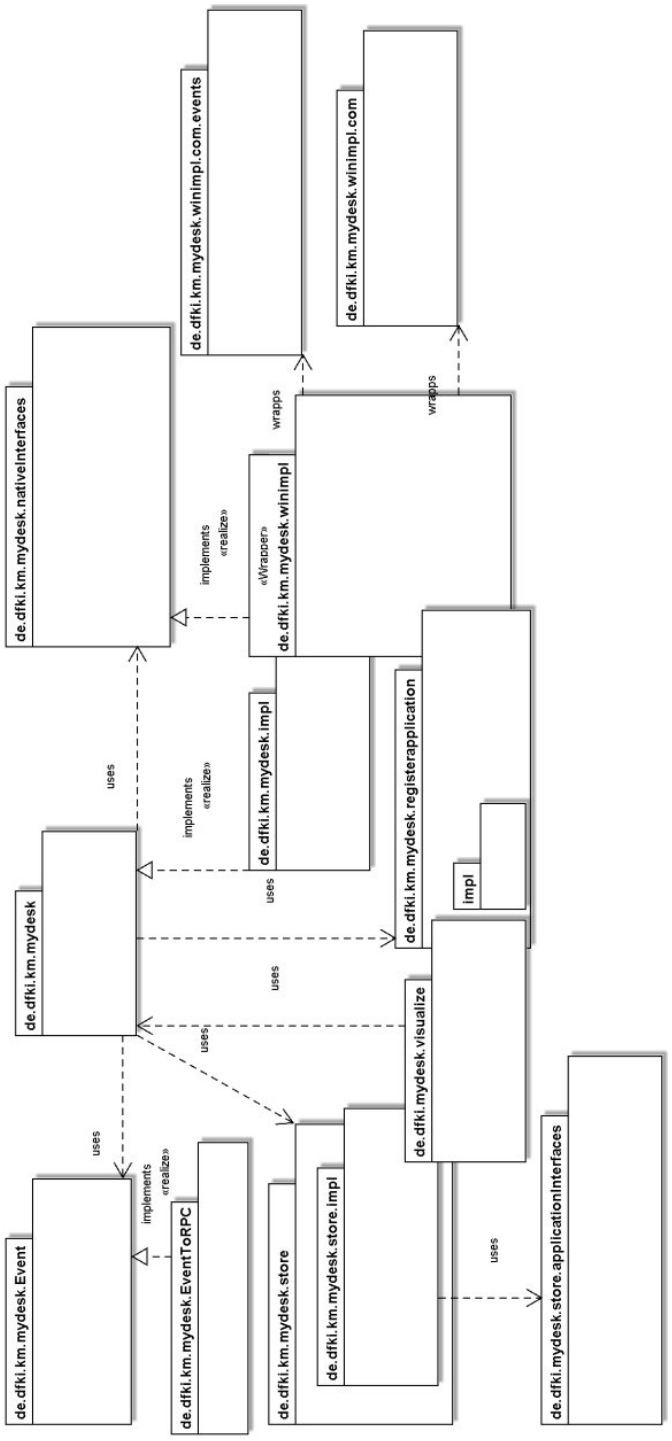



Figure 3.2: UML package diagram of the MyDesk framework

# Chapter 4

# The implementation of the interfaces

This chapter elucidates interesting facts about the implementation of the interfaces.

## 4.1 Implementation of the native Interfaces

At the moment there exists only one implementation of these interfaces for Windows NT, XP and Vista, to access the Win32API windowing functions C# and the .Net framework are used. The reason to use C# to gain to access the Win32API is that it is very easy and fast to implement and to export this implementation as COM component. This COM components will be wrapped by COM4J to give Java applications access to this components. The following code snippet shows how to import the Win32Api function to set the foreground windows.

```
[DllImport("user32.dll", CharSet = CharSet.Auto)]
private static extern bool SetForegroundWindow(IntPtr hWnd);
```

The solution using COM and COM4j allows to implement a bridge between Java and .Net

This snippet shows the method which wraps the imported Win23Api function and is exported as COM component.

```
public bool setForegroundWindow(string id)
{
if (String.IsNullOrEmpty(id))
      {
```

```
        return SetForegroundWindow(IntPtr.Zero);
      }else
      {
            return SetForegroundWindow(new IntPtr(Int32.Parse( id)));
      }
}
```

For every C# (.Net) class which should be accessible via COM there has to be a COM compatible interface which the class has to implement. This interface has to define a GUID (a Globally Unique Identifier) which is needed to identify the COM Type. Below an example of an interface definition that allows the export of the class which implements this interface

```
[Guid("64B7CE2E-654E-4007-BF30-36FDBF1DE459")]
[InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
[ComVisible(true)]
public interface Iexport
{
[DispId(3)]
      bool setForegroundWindow(string id);
}
```

To export the resulting .Net dll to COM and generate the needed tlb (typelibrary) the command line tool "regasm" is used. If only the tlb shall be generated from the dll the tlbexp tool can be used. But regasm does both registers the dll as COM component and generates the tlb both in one :

```
regasm Multidesk.dll /tlb:Multidesk.tlb

using tlbexp and regasm
tlbexp Multidesk.dll /out:Multidesk.tlb
regasm Multidesk.dll
```

The tlb file will be used by COM4j to generate the Java interface to gain access to the COM components. This is done by the tlbimp.jar from COM4J distribution.

The COM4J proxy implements the Java interfaces and connects to the COM component. The Java interface was generated from the type library definition of the COM component. In our special case a COM component callable wrapper will be created from the .Net runtime environment if the COM component is requested

from the COM4J proxy. This wrapper gives the COM client in our case the COM4J proxy access to the functionality implemented in C#.

## 4.2 Implementation of the MyDesk Interfaces

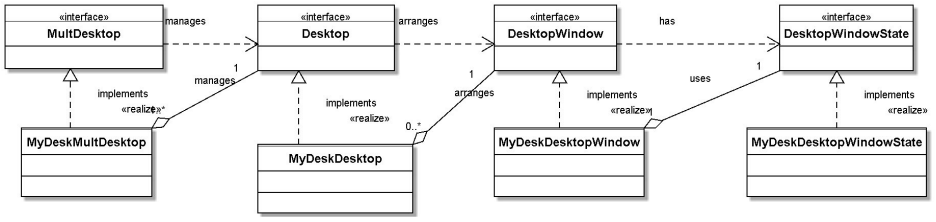



Figure 4.1: UML class diagram of the MyDesk core interfaces and their implementation

### 4.2.1 MyDeskMultiDesktop

The implementation of the MultiDesktop interface is the MyDeskMultiDesktop class. It organises the desktops by using a vector. The methods switchToNextDesktop and switchToPrevDesktop simulate a ring buffer on this vector but excludes its first element. As a result this implementation of the MultiDesktop has one hidden Desktop at the first position of the vector. It is not intended that this Desktop can be accessed by the user like the other Desktops. The profit of this Desktop is that the user can clean up his desktop by moving windows to this hidden Desktop. Windows which the user wants to hide might be windows which are not directly needed for the work the user is doing. The only way to gain access to this Desktop is to use the visualisation dialogue. This means that the hidden desktop cannot be shown like a normal Desktop, it is a place to put windows, which are used very seldom.

New Desktops can be added to the system at any time. If a new desktop should be added during runtime the programmer should take care that his user interfaces can manage it. The current MyDesk User interfaces does not provide the possibility for the user to add a new desktop.

The implemented user interface of the MyDesk system currently manages exactly four desktops only. Thus the system has a total count of five desktops.

Opportunities to switch to another desktop provided by the MulitDesktop are the following:

- Next / previous: switch to the next or previous desktop in the vector,

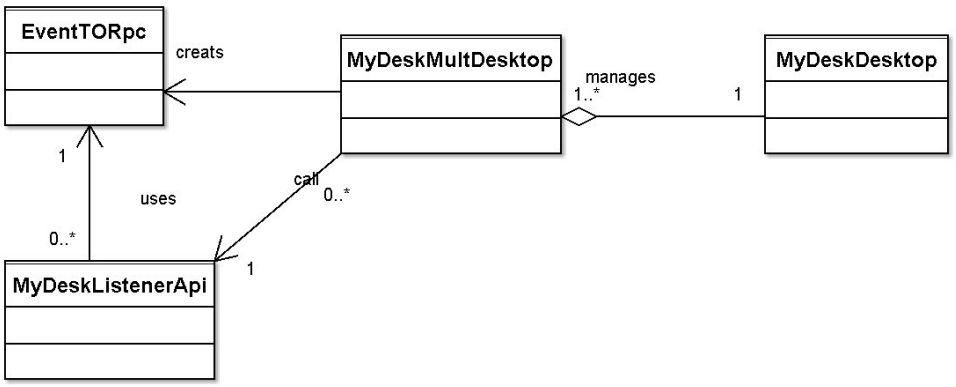



Figure 4.2: UML diagram of the MyDeskMultiDesktop class including related classes

- Switch to desktop object : if the reference to the destination desktop is known, it could be used to switch to that desktop
- Switch to the x-th desktop, x is the position of the destination desktop in the vector

### 4.2.2 MyDeskDesktop

The MyDeskDesktop class implements the Desktop interface (from the package de.dfki.km.mydesk) and uses the DesktopNativeInterface interfaces to gain access to the needed functionality of the operating system. MyDeskDesktop uses a Vector to organise the DesktopWindows. The windows in this Vector are organised in the way, that the top most window in z-order is the first element and the last window in z-order is the last element of this vector. There is only one location in the code of the MyDeskDesktop class where new DesktopWindows are created. This is the update method. It is a very important method because it will synchronise the Desktop object with the users desktop.

This method calls the getWindowsFromAktualDesktop method to get a list which contains the identifiers of all visible user windows sorted in z-order with the top most window fist. Now it passes through all elements of this list. For each identifier will be looked if there already exists a DesktopWindow which owns this identifier. If such a DesktopWindow exists it will be updated and added to the new vector. If there does not exist such a DesktopWindow a new DesktopWindow will be created and added at the end of the vector. If all identifiers are attended the new vector replaces the old DesktopWindow vector. This procedure assures that

the new vector is in z-order, because of the z-order of the identifier list and it also assures that no information about existing DesktopWindows will be lost.

## 4.3 The implemented user interface of the MyDesk system

There are three different ways for the user to control the MyDesk System. The first are global shortcuts, the second are the tray icons and the third is the visualisation dialogue. The visualisation dialogue will not be described in this section.

The global shortcuts are implemented in C#. The hook of the implementation to the Java system is done by COM4J. The shortcut implementation uses a hidden window with an overridden WndProc method to catch key strokes.

Tray icons are also implemented in C# with the help of the .Net Framework. This Framework provides a class to create tray icons. It is possible to set the path of the directory where the icons are placed. There exists an option to invert the order of the tray icons. This is needed by Windows NT bases systems. The tray icons also provide the menu of the MyDesk system. This menu allows to exit the system, move the top most windows, and configure and visualise the system.

In the implementation of the MyDesk system there exist three different kinds of user interfaces. The first one are the global shortcuts and the other both are graphical user interfaces. The first of the graphical user interfaces are the MyDesk tray icons, the second one is the visualisation dialogue. This visualisation dialogue displays all Desktops and their DesktopWindows in a tree and allows the user to move DesktopWindows from one Desktop to another by drag and drop. It also displays further informations about the DesktopWindow.

## 4.4 Using the MyDesk framework

In this section two implementations are described which use the framework. They can be used as an example to see what can be done with this framework.

### 4.4.1 A running desktop manager

The MyDesk class is used to realise a runnable system which uses the MyDesk framework. It creates a fix number of desktops (five) and connects the implementation of the user interfaces with the framework. Thus this class realises a desktop manager which uses the MyDesk framework.

### 4.4.2 Visualization

The class Visualize implements a visualisation tool which visualises a MultiDesktop instance. That means it uses a class of the type MultiDesktop and displays its desktops and their DesktopWindows in a tree. It allows to move DesktopWindows between the desktops by using drag and drop and shows the information about a DesktopWindow stored in the DesktopWindowState object in a table. This tool is implemented in a way that it can work with any implementation of the MyDesk interfaces.

# Chapter 5

# User documentation

## 5.1 How to install MyDesk

### 5.1.1 Requirements

The minimum software requirements for MyDesk are :

- J2SE Runtime Environment 5.0 or better,
- Microsoft .NET Framework 2.0 or better, it can be found here: http://www.microsoft.com/downloads/details.aspx?displaylang=de&FamilyID=0856eacb-4362-4b0d-8edd-aab15c5e04f5
- currently MyDesk runs only on Microsoft Windows NT or better

### 5.1.2 How to get

To get MyDesk, check out the project "MyDesk" from the following SVN repository: http://www3.opendfki.de/repos/mymory/trunk/Software/MyDesk

### 5.1.3 Installation

On Microsoft Windows systems run the register.bat located in the bat directory of the MyDesk project. You need administrator rights to do so. This registers the Multidesk.dll to the Global Assembly Cache and to the registry, so that the DLL can be used by any COM client. We do this from Java via com4j.

### 5.1.4 How to run MyDesk

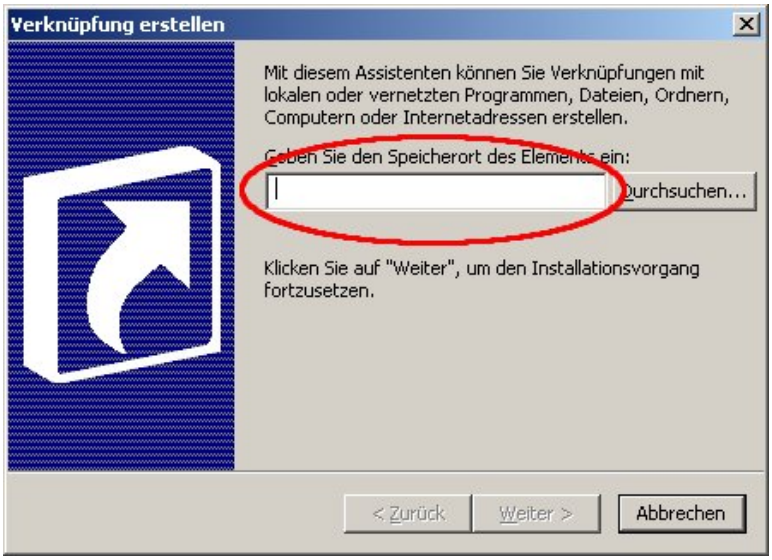
There are four possible opportunities to start MyDesk on a Windows system:

1. MyDesk can be started from eclipse, but this is only advisable for testing and debugging.
2. Use runMyDesk.bat to run MyDesk from the jar file, located in the lib subdirectory.
3. runMyDesk_classes.bat to run MyDesk from the classes/java subdirectory.

Both ways open a shell window where you can see the messages from the Java application (and on windows the messages from the DLL) which are written to the console.

### 4. Run MyDesk with the help of javaw

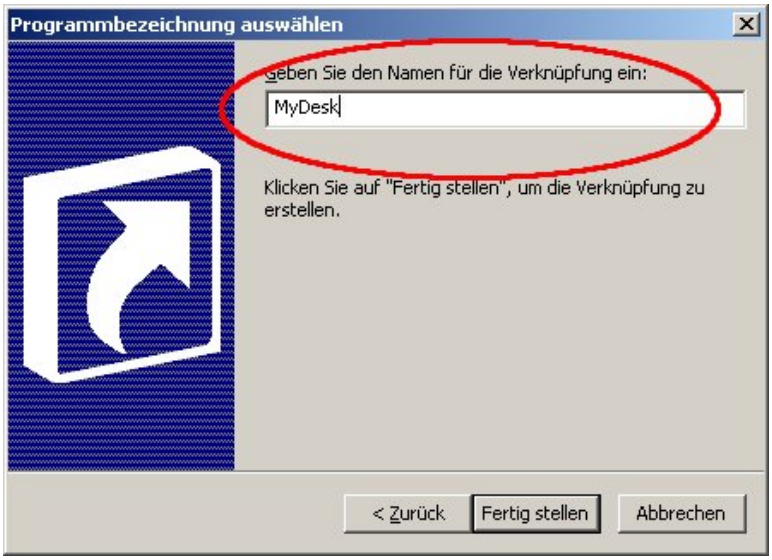
First create a new link.



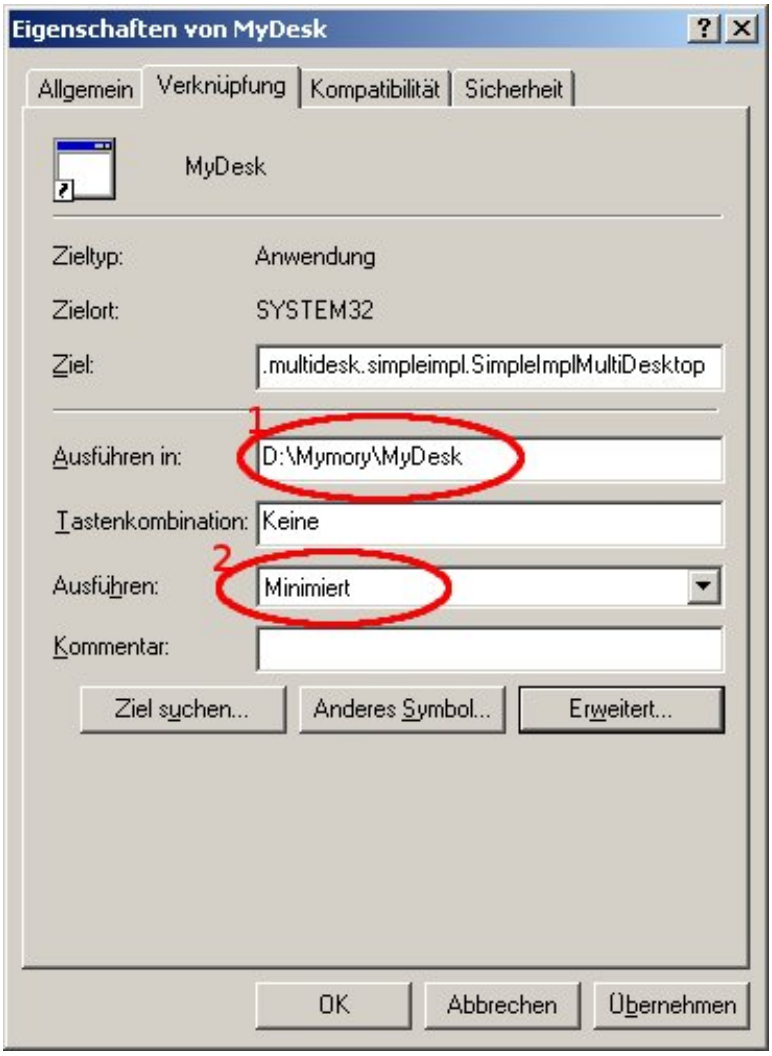

Paste :

```
%windir%\SYSTEM32\javaw.exe -classpath lib/MyDesk.jar;import\com4j.jar;impo
de.dfki.km.mydesk.impl.MyDesk
```

and click on "next".




Change here the name of the link e.g. "MyDesk", because javaw.exe is not a striking name. Afterwards click on "finish". Then you have to edit the properties of the newly created link. for this purpose open the Properties Dialogue of the link.




1. For "Ausführen in" enter the path of your installation of MyDesk.

2. For "Ausfüren" select minimized.

If you want to you can also change the icon. You can take all icons from the icon directory that can be found in your MyDesk installation directory. All icons with the prefix link are designed for the use with links.

## 5.2 First Steps

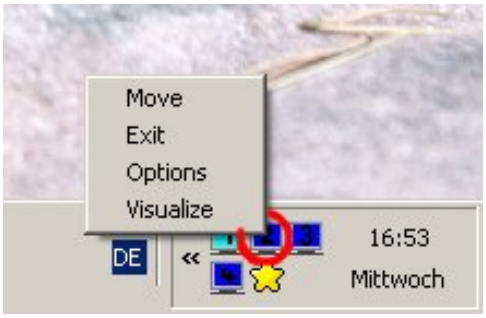
### 5.2.1 The tray icons

If you have started MyDesk in the way you prefer, four tray icons appear in the taskbar.

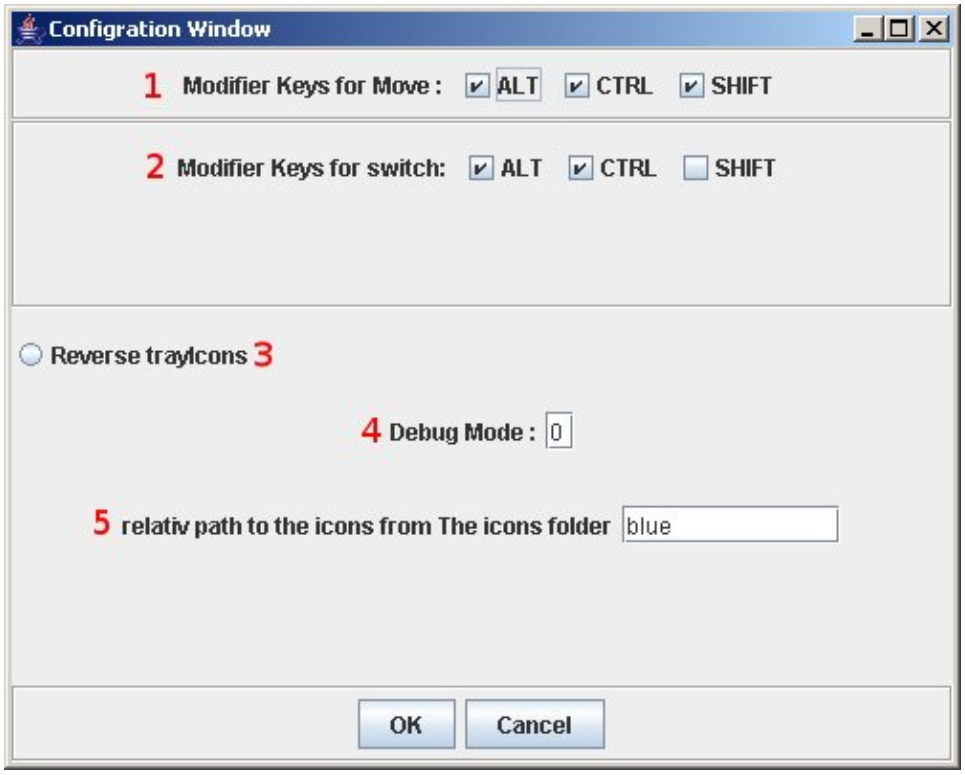

MyDesk desktop manager provides four desktops. Each desktop is represented by one tray icon and can be identified by the number which is displayed in the icon. The desktop that is currently active (i.e. whose windows are visible), has a tray icon showing a brighter colour than the other desktops.

### 5.2.2 Configure MyDesk

To configure MyDesk you have to start it first. After MyDesk has been started, click with the right mouse button on one of the MyDesk desktop icons in the taskbar. The context menu will appear.




Select the menu entry "Options" and the MyDesk Configuration Window appears.




1. The modifiers for the move shortcut can by changed. Changing the keys 1-4 is not possible.

2. The shortcut modifiers of the switching action can by modified. At this point the keys ArrowLeft and ArrowRigth cannot by changed either.

3. The reverse tray icons option is only needed if the placement of the MyDesk tray icons has to be reversed. On the operating systems Microsoft Windows NT and Windows2000 this option should be activated to get the correct order.

4. Normal users should let this option unchanged (debug mode = 0). Developers who want to have access to eclipse and visual studio windows on every desktop use value 3. The value debug mode = 2 will induce that only the first desktop has icons on the desktop (the documents and links lying on the desktop), all other desktops will have no such icons.

5. You can change the look (actually only the colours) of the tray icons. The default value is blue. Other available colours are yellow, blue_ yellow. If

you want to create your own coloured icons, create a copy of one of the subfolders of icon folder with another name and change the colour of icons. Now you can use the name of the new folder to load your icons. To reload the icons you have to restart the MyDesk desktop manager.

### 5.2.3 Switching between desktops

There are three ways to switch between the four desktops.

- switching by using the mouse
  - you switch to another desktop by clicking the tray icon of the desktop to which you want to switch with the left mouse button.
- switching by keyboard
  - you can switch to the next or previous desktop by using the shortcuts ctrl+alt+ArrowRigth or crtl+alt+ArrowLeft
  - or you can jump either with the mouse or by using ctrl+alt+1-4 directly to another desktop.

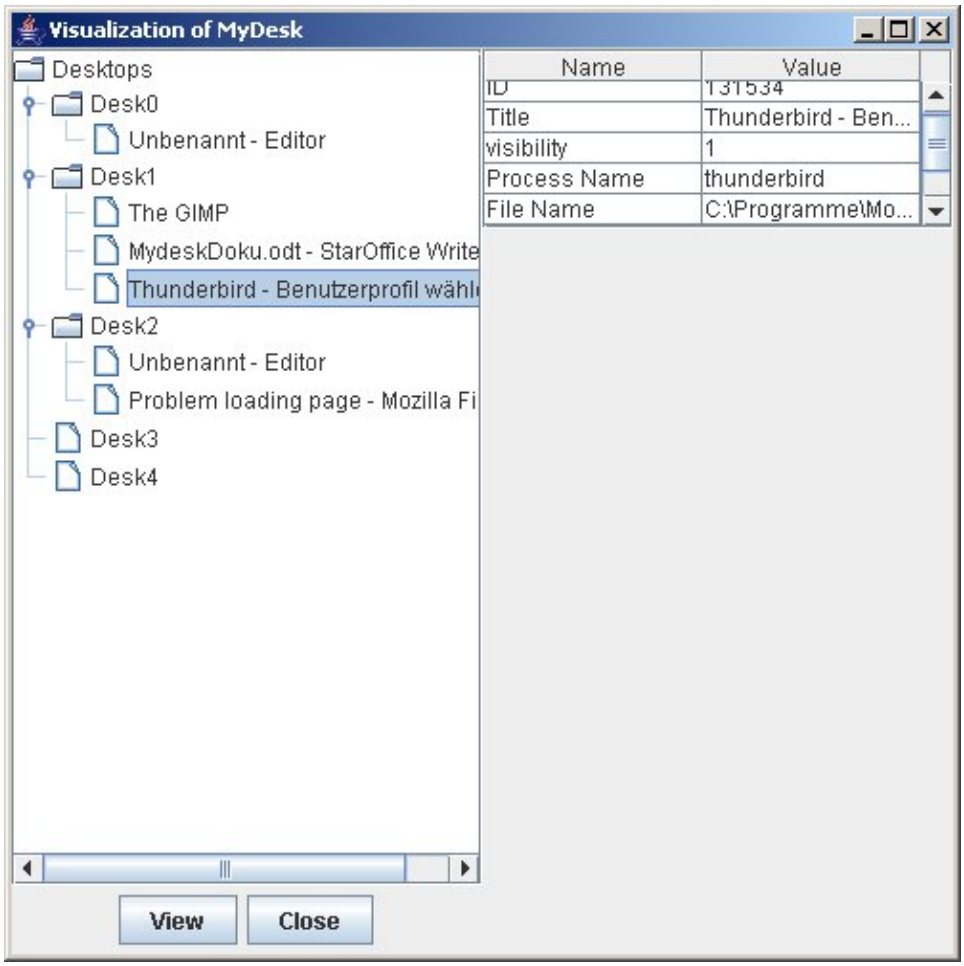
### 5.2.4 Moving windows between desktops

You can move the active window (i.e. the window which is in the foreground and is focused) by clicking with the right mouse button on the tray icon of the target desktop and select the "Move" menu entry in the appearing context menu. Or you can use the shortcut ctrl+alt+shift+1-4 . 1-4 is the number of the target desktop. Windows that are not in the foreground can only be moved to another Desktop by using the visualisation dialogue.

## 5.3 The Visualization Dialogue

The visualise menu entry of the context menu will open a dialogue which displays all desktops and their windows in an tree. The visualisation dialogue can be used to manage the Desktops and their DesktopWindows. All desktops are displayed in a tree structure. The windows are displayed in the tree as children of the desktops. By selecting a window all information about this window which is stored by the MyDesk desktop manager is displayed at a table on the right side of the dialogue. Furthermore it is possible to move windows from one desktop to another by drag and drop. This makes it more easier to arrange the windows on the desktops and to get an overview on which desktop a window is placed. There also exists a fifth

desktop named "‘desk0"’. This desktop can be accessed only by the visualisation dialogue. It can be used to hide some windows which have to be open, but are not needed by the user.



# Chapter 6

# Conclusion

During this work a framework has been developed which provides the functionality of a desktop manager. This desktop manager implements an event system to enable services responding to desktop changes for example. In further work this event system will be used to connect this framework to a user observation and context elicitation framework. For this purpose a prototype already exists. Furthermore the architecture has been chosen to be highly extensible. It was realised by the implementations of the interfaces which only use the other interfaces instead of the realisations of these interfaces. Thus it is easy to replace the implementation of one interface or to port the system to another operating system.

To store and recover a desktop to 100 percent it is necessary that all applications support the MyDesk framework to 100 percent. Because of the large number of applications this is not realistic. It will be a good result if all applications which are used in the Mymory scenarios would support this framework. To reach this degree of support there have to be written extensions for all these applications to get the desktop-specific state information of these applications. The store and recover part of the MyDesk framework has already the state prof of concept. It shows that it is possible to store and recover a desktop completely if the applications support the framework. Currently a Mozilla Firefox extension provides this support. Firefox is the main application needed in the Mymory project. In further work support for more applications has to be implemented.

Furthermore a desktop manager with 5 desktops has been created during this work. This application runs very stable and can be used during normal work. The store and restore functionality works for Mozilla Firefox only. However, if you intend to use the multi-desk framwork only temporarily and/or without the need to you store and restore desktops, you can surely work with any windows application.

# Bibliography

[Ara01] C. Aravind. Understanding classic com interoperability with .net applications. http://www.codeproject.com/dotnet/cominterop.asp, 2001.

[COM] com4j. https://com4j.dev.java.net/.

[Ell06] Frank Eller, editor. *Visual C# 2005*. Addison-Wesly, Muenchen, 2006.

[Mic] Microsoft Corporation. *MSDN Library for Visual Studio 2005*.

[Par04] Nick Parker. Exposing .net components to com. http://www.codeproject.com/dotnet/nettocom.asp, 2004.