# jACOB Programming

Andreas Herz

# Table of Contents

# Gui Programmierung

# Gui Programmierung

# Table of Contents

# List of Figures

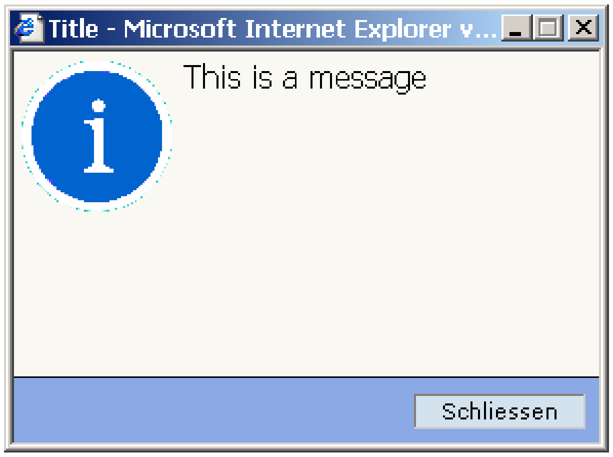
# Chapter 1. MessageDialog

## 1. Introduction

It is clear that these dialog boxes are always needed in programs that have a user interface. After all, we are humans, humans make mistakes, and dialog boxes notify us when we err.

Not much more can be said about "MessageDialog" -- it really is easy.

**Figure 1.1. Screenshot of an *IMessageDialog* in action**



This is a message

## 2. An Example

### 2.1. Preface

In this simple example we notify the user that they have pressed a button. We can call them the *Hello World* of dialog boxes.

### 2.2.1. The *MessageDialog* within a ButtonHandler

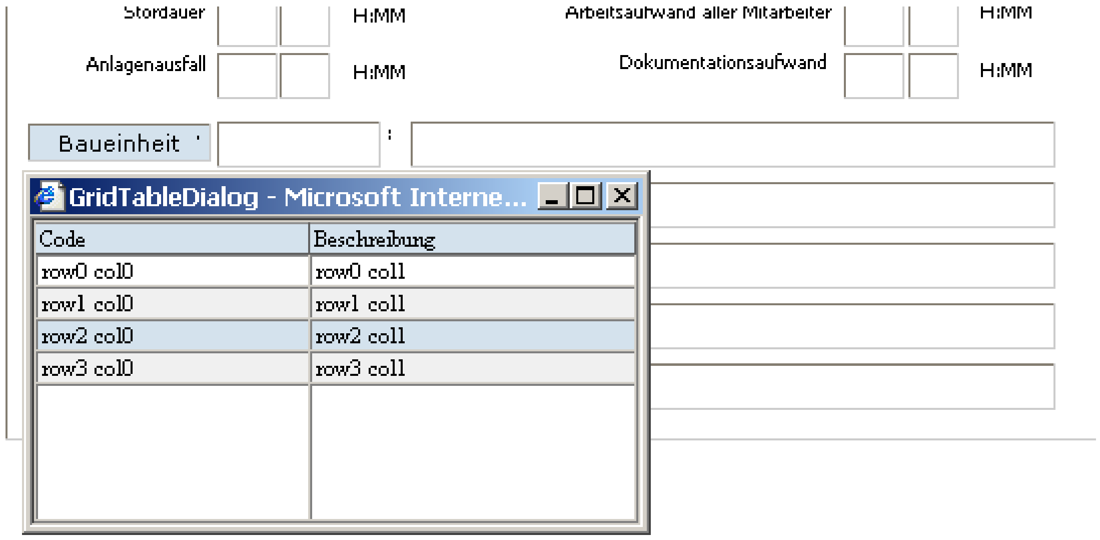
```
package jacob.event.screen.f_call_manage.taskEdvin;
import java.util.*;
import de.tif.etr.screen.*;
import de.tif.etr.screen.dialogs.*;
import de.tif.etr.screen.event.IButtonEventHandler;

public class GetBaueinheit extends IButtonEventHandler
{
  public void onAction(IClientContext context, IGuiElement button)throws Exception
  {
    IMessageDialog dialog =context.createMessageDialog("Title","This is a message");
    dialog.show();
  }

  public void onGroupStatusChanged(IClientContext context, int status, IGuiElement button) throws Excep
  {
  }
}
```

# Chapter 2. GridTableDialog

## 1. Introduction

GridDialogs are a subclass of IDialog components specifically designed to display and select data in a grid or table view. Grids in a crossware application provide a broad choice of interfaces through which data can be represented in tabular form. In the case of our *GridTableDialog* we enable the user to select *one* row from an *n-size* dataset. The fields of the data can be linked to fields in the UI or you can proceed the click event manualy.

**Figure 2.1. Screenshot of an *IGridTableDialog* in action**


Stordauer
H:MM
Arbeitsaufwand aller Mitarbeiter
H:MM
Anlagenausfall
H:MM
Dokumentationsaufwand
H:MM
Baueinheit
GridTableDialog - Microsoft Interne...

| Code | Beschreibung |
|---|---|
| row0 col0 | row0 col1 |
| row1 col0 | row1 col1 |
| row2 col0 | row2 col1 |
| row3 col0 | row3 col1 |

The *GridTableDialog* encapsulates common functionality so that you can easily reuse it. For example, you often need to display and select information from a foreign sql table or view. To generalize this code, you might first create a function to collect the data and then pass in the column names and the data to display the GridDialog. The user selects one row in the popup dialog and the GridDialog fill back the selected row in the connected input fields - thats all.

## 2. An Example

### 2.1. Preface

We create test data and display them in an GridTableDialog. The user has now the capability to select one row. The GridTableDialog fill back the selected row in the GUI input fields. A simple example which will covers all the relevant aspect of using this type of dialog.

### 2.2. The data binding

The Grid is data-bound, which means it's linked to an java array, which provides data for the grid to display. An array for the header naming and one for the data itself.

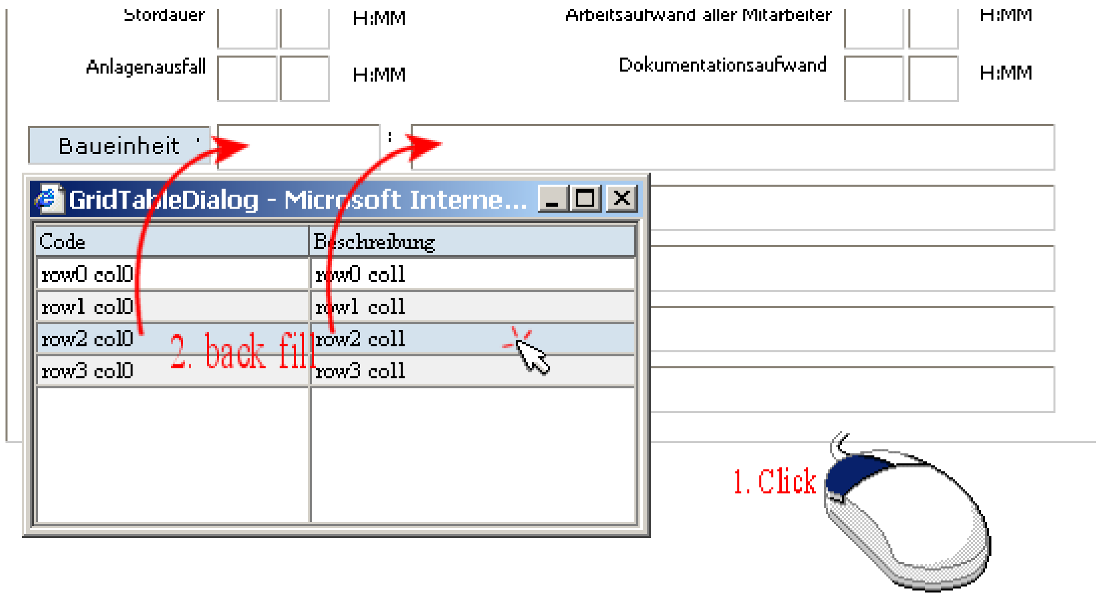
**Figure 2.2. bind a grid dialog with a data table**

```
String[] header = new String[]{"Code" , "Beschreibung"};

String[][] testdaten=new String[4][2];
testdaten[0]= new String[]{"row0 col0","row0 col1"};
testdaten[1]= new String[]{"row1 col0","row1 col1"};
testdaten[2]= new String[]{"row2 col0","row2 col1"};
testdaten[3]= new String[]{"row3 col0","row3 col1"};

dialog.setHeader(header);
dialog.setData(testdaten);
```

## 2.3. The GUI binding

One option to process the user selection is the binding with *ISingleDataGuiElement* - the data representation of all input fields. The user clicks on one row and all linked *ISingleDataGuiElement* fields will be filled with the cell data of the connected columns.

**Figure 2.3. Backfill of an *IGridTableDialog* in action**



| Code | Beschreibung |
|---|---|
| row0 col0 | row0 col1 |
| row1 col0 | row1 col1 |
| row2 col0 | row2 col1 |
| row3 col0 | row3 col1 |

**Figure 2.4. bind a grid dialog selection with an input field**

```
ISingleDataGuiElement code =(ISingleDataGuiElement)context.getGroup()
                                         .findByName("inputfield_A");
ISingleDataGuiElement desc =(ISingleDataGuiElement)context.getGroup()
                                         .findByName("inputfield_B");

IGridTableDialog dialog = context.createGridTableDialog();  ❶

dialog.connect(0,code);                                      ❷
dialog.connect(1,desc);                                     ❸
```

❶ Create the grid dialog.
❷ Connect the column 0 with the *ISingleDataGuiElement* with the name *inputfield_A*.
❸ Connect the column 1 with the *ISingleDataGuiElement* with the name *inputfield_B*.

## 2.4. But I Want More

The *GridTableDialog* can do much more. The generic feature to proceed the user selection is the binding via callback object. You define a callback object which will be called (1) if the user selects an row in your grid.

### Figure 2.5. bind a grid dialog selection with an callback object

```
class myCallback implements IGridTableDialogCallback        ❶
{
  public void onSelect(IClientContext context, int rowIndex, Properties cellData)
  {
    System.out.println("the data ==================================");
    cellData.list(System.out);
  }
}
IGridTableDialog dialog = context.createGridTableDialog(new❷ myCallBack());
```

❶ Create a callback class which is an subclass of *IGridTableDialogCallback*.
❷ Create the grid dialog with an instance of myCallback for manually operated user selection.

## 2.5. The *GridTableDialog* within a ButtonHandler

```
package jacob.event.screen.f_call_manage.taskEdvin;
import java.util.*;
import de.tif.etr.screen.*;
import de.tif.etr.screen.dialogs.*;
import de.tif.etr.screen.event.IButtonEventHandler;

public class GetBaueinheit extends IButtonEventHandler
{
class GridCallback implements IGridTableDialogCallback
{
  public void onSelect(IClientContext context, int rowIndex, Properties columns)
  {
    System.out.println("- Cell data -");
    columns.list(System.out);
  }
}

public void onAction(IClientContext context, IGuiElement button)throws Exception
{
    IGridTableDialog dialog = context.createGridTableDialog(new GridCallback());

    String[] header = new String[]{"Code" , "Beschreibung"};

    String[][] testdaten=new String[4][2];
    testdaten[0]= new String[]{"row0 col0","row0 col1"};
    testdaten[1]= new String[]{"row1 col0","row1 col1"};
    testdaten[2]= new String[]{"row2 col0","row2 col1"};
    testdaten[3]= new String[]{"row3 col0","row3 col1"};
```

1There is no warranty that the callback object will be called. Nothing happens if the user closed the window with **Alt-F4**

```
    dialog.setHeader(header);
    dialog.setData(testdaten);

    dialog.show(300,200);
}

public void onGroupStatusChanged(IClientContext context, int status, IGuiElement button) throws Excepti
{
}
}
```

# Chapter 3. FormDialog

## 1. Introduction

The jACOB FormDialog framework helps you lay out and implement elegant dialogs consistently and quickly. It aims to make simple things easy and the hard stuff possible, the good design easy and the bad difficult.

This document introduces the FormsDialog framework for jACOB, presents design goals, explains how these have been addressed, aquaints you with the Forms layout model and API.

Forms focuses on form-oriented panels much like the 'Segment' panel. Nevertheless, it is a general purpose layout system that can be used for the vast majority of rectangular layouts.

## 2. How Forms addresses the design goal?

A Forms framework follow five principles

1. Use a grid for a single layout.
2. Separate concers.
3. Provide a powerful layout specification language.
4. Allow string representations to shorten layout code.
5. Provide developer guidance on top of the layout manager.

### 2.1. Grid systems

Grid systems are a powerful, flexible ans simple means to lay out elements. Professional designers use a grid to find, balance, construct and reuse good design in artwork and everyday media. And we found that many user interface developers use grids implicitly to lay out or align components. This works well with paper and pencil and to a lesser extendt with today's visual builders. The latter often stifle creativity more than they assist in finding and constructing good design.

### 2.2. Layout manager

Layout manager often combine many features in a single class: specify a layout, fill a panel with components and set component bounds. On the other hand most layout systems lack support for frequently used layouts and provide no means to reuse common design.

### 2.3. Form layout

With the FormLayout you describe a layout before you fill apanel and before the layout manager sets the component bounds. You can specify the form's grid in an expression language so that readers can infer a panel's layout from your code quickly and modify it easily. A single specification can be applied to many components; for example, you can specify that all labels in a column are right aligned.

### 2.4. String representation

To further imporve the code readybility we allow to specify the grid in a human readable and concise language using string representations. This way even complex layouts can be expressed in a few lines of code.

## 2.5. Form

Forms provides abstraction on the top of the layout manager that lead to consistent UIs and assist you in style guide compliance. They help you to traverse the grid and seed it in components.

# 3. An simple Example

## 3.1. Preface

Let's jump into a example that demonstrate basic FormLayout features. We're going to learn the details step by step in the following sections and will learn to write more complex layouts with less code.

FormLayout is a powerful, flexible and precise layout manager that aligns components vertically and horizontally in a dynamic rectangular grid of cells, with each component occupying in one or *more* cell. To define a form layout you specify the form's *columns*, *rows*. Everyhing that applies to columns applies to rows too - just with a different orrientation. FormLayout uses the same API, algorithms and implementation for column and rows.

## 3.2. Requirements

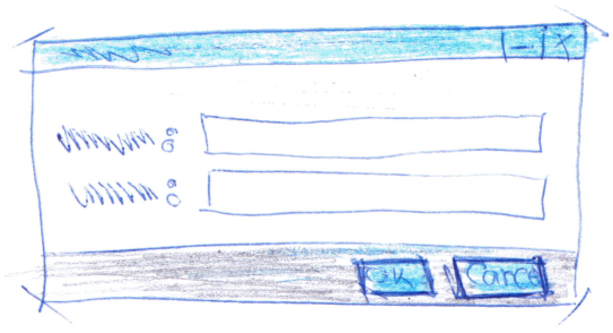
"We need a panel to edit person data with fields for firstname and surename".

—The Boss

## 3.3. Finding the Layout

A visual designer produces a design draft and hands it over to the developer

**Figure 3.1. The draft of the panel**



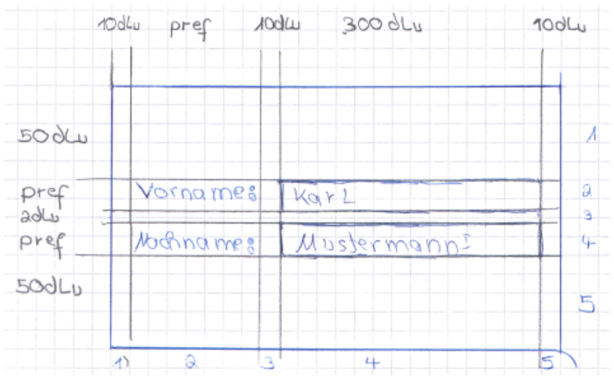
"..in mind: Follow the customer Style Guide!"

—The Designer

## 3.4. Finding the Grid

After the accurate definition of the requirements you can start to develop the dialog. Identifies the columns and rows in the form with the border and the spacing between the objects. Ignore the button bar. The buttonbar will be insert from the FormDialog framework.

**Figure 3.2. The grid of the panel**



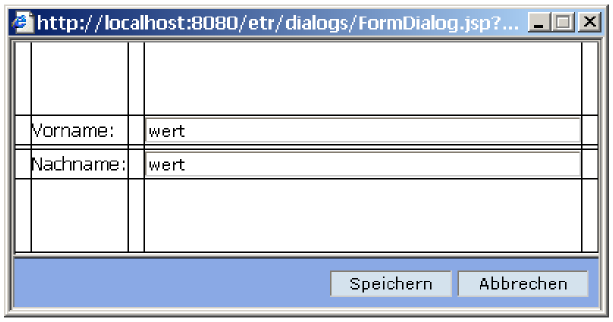
## 3.5. The GUI Implementation

**Figure 3.3. The java code for the panel**

```
FormLayout layout = new FormLayout(                                   ❶
               "10dlu,p, 10dlu,300dlu,10dlu",          // 5 columns   ❷
               "50dlu, p,2dlu,p,50dlu");               // 5 rows      ❸

CellConstraints c=new CellConstraints();

IFormDialog dialog=context.createFormDialog("Titel",layout, ❹new Callback());

dialog.addLabel("Vorname:",c.xy(1,1));                                ❺
dialog.addTextField("firstname","wert",c.xy(3,1));                    ❻

dialog.addLabel("Nachname:",c.xy(1,3));
dialog.addTextField("lastname","wert",c.xy(3,3));
dialog.show();
```

❶ We construct a FormLayout and specify the columns and rows using a string representation
❷ Defining the columns of the FormLayout where *p* or *pref* for 'prefered size', and *dlu* for 'dialog units' that scales with the font.
❸ Defining the rows of the FormLayout.
❹ Create the form layout dialog with the defined layout and a *callback* object for the user interaction.
❺ Insert a label in the form dialog with a given CellConstraint. The constrains defines the behavior and the position of the object. In this case we place the object in column=1 and row=1. Be in mind that the rows and columns begins with the index 0.
❻ Insert an Input field with the property name *firstname* and the default value *wert* in the form. We place the object at column=3 and row=1. At the right hand side of the label.
❼ Tells the framework that the dialog is ready and can be display to the user.

### Figure 3.4. The panel in debug modus

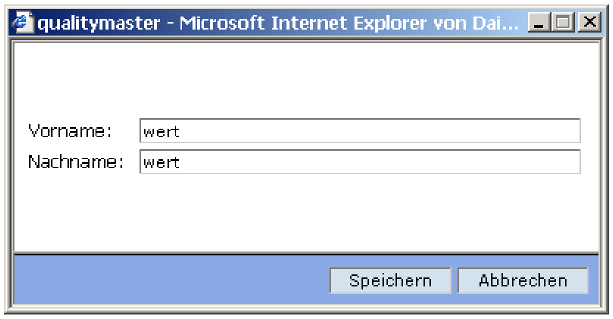


You can enable the debug modus during the development of the panel with:

```
dialog.setDebug(true);
```

It helps you to check your layout contrainst.

### Figure 3.5. The ready to run panel



## 3.6. The callback Implementation

The callback object is a tribute to the client/server architecture of jACOB or other C/S applications. It is not possible to wait on the server for next user interaction. In this case we must implement a callback mechanism to handle the user interaction. The callback object will be called if the user clicks on the [ok] or [cancel] button on the next request cycle(2).

```
class Callback implements IFormDialogCallback                ❶
{
  public void onCancel(IClientContext context)
  {
  }
  public void onOk(IClientContext context, Properties props)
  {
    String firstname = props.getProperty("firstname");      ❷
    String lastname  = props.getProperty("lastname");       ❸

    System.out.println("Te user name is :"+firstname+" "+lastname);
  }
```

2There is no warranty that the callback object will be called. Nothing happens if the user closed the dialog with **Alt-F4**

```
}
```

❶ The class must implement the interface *de.tif.etr.screen.dialogs.form.IFormDialogCallback*
❷ Retrieve the *firstname* from the property set. Be in mind that you have insert a TextField in the dialog with this name!
❸ Retrieve the *lastname* from the property set.

## 3.7. The FormDialog within a ButtonHandler

This illustrates the handling of a user interaction. The user press a button and the custom dialog will shown and proceed.

```
package jacob.event.screen.f_qualitymaster.request;

import java.util.Properties;
import de.tif.etr.core.data.DataTable;
import de.tif.etr.screen.*;
import de.tif.etr.screen.dialogs.form.CellConstraints;
import de.tif.etr.screen.dialogs.form.*;
import de.tif.etr.screen.event.*;

/**
 * My first FormDialog which will be triggerd from a generic button
 */
public class RequestNew extends IButtonEventHandler
{
  // The callback class for the user dialog(!!) interaction
  class Callback implements IFormDialogCallback
  {
    public void onCancel(IClientContext context) {}

    public void onOk(IClientContext context, Properties props)
    {
      String firstname = props.getProperty("firstname");
      String lastname  = props.getProperty("lastname");

      System.out.println("The user name is :"+firstname+" "+lastname);
    }

}
  /* Will be called if the user press the corresponding button
   * in the jACOB Form( not the dialog!!)
   */
  public void onAction(IClientContext context, IActionEmitter emitter)
  {
        FormLayout layout = new FormLayout(
            "10dlu,p, 10dlu,300dlu,10dlu",          // 5 columns
            "50dlu, p,2dlu,p,50dlu");               // 5 rows

        CellConstraints c=new CellConstraints();

        IFormDialog dialog=context.createFormDialog("Titel",layout,new Callback());

        dialog.addLabel("Vorname:",c.xy(1,1));
        dialog.addTextField("firstname","wert",c.xy(3,1));

        dialog.addLabel("Nachname:",c.xy(1,3));
        dialog.addTextField("lastname","wert",c.xy(3,3));
        dialog.setDebug(true);
        dialog.show();
  }

  /* Will be called if the state of the button group has been changed.
   */
  public void onGroupStatusChanged(IClientContext context, int status, IGuiElement emitter) throws Exce
  {
  }
}
```