The Mjølner System
Frigg - Interface Builder
Users Guide

Mjølner Informatics Report
MIA 96-33(1.4)
November 1997

1. Introduction

Frigg is the interface builder of the Mjølner System, it aims at supporting rapid prototyping and is meant primarily for system designers and developers.

Frigg does not enforce one particular style of development on its users. In fact, Frigg supports at least the following approaches: (1) starting with design of user interface objects (UIOs) and building up a horizontal prototype having only minimal functionality; (2) starting from a model, and an implemented description of the functionality of a system, but with no UIOs implemented; (3) vertical prototyping, i.e. implementing the functionality behind a subset of a horizontal prototype or fully implementing a small subset of a system intended for incremental extension; (4) simulation of functionality, i.e. adding temporary short cut or dummy computations to a horizontal prototype to support sample data; and (5) full application development. In addition, these different ways of using Frigg can be combined.

Frigg is integrated with the other tools of the Mjølner System. It is embedded the Mjølner BETA Sourcebrowser, and supplies a graphical editor which interacts with the structure editor to create a prototyping environment. The graphical editor lets its users construct the user interface via direct manipulation, while the necessary code is generated automatically behind the scene. The generated code can be extended and tailored using the structure editor, and this tailoring can take place at any time during the development of the user interface, because the graphical editor can keep track of the code even though the code has been altered in the structure editor. The developer can therefore alternate between working on the user interface and coding the underlying functionality.

The integration of Frigg's editors is accomplished using the Mjølner System's uniform representation of programs: abstract syntax trees (ASTs). Manipulations of the ASTs are done through the Meta Programming System (MPS). Furthermore, all editors working on the same AST are informed when the AST changes, thereby allowing the editors to ensure consistency. The ASTs generated by Frigg are decorated with special comments that are used by Frigg to recognise user interface objects and to store links to layout information.

Frigg utilises the Mjølner BETA Fragment System to: (1) Get access to Lidskjalv, the Mjølner BETA Graphical User Interface environment (GUIenv), (2) Create user interface modules that are properly separated from the model modules of the program, (3) Separate the interface part and the implementation part of the user interface objects, to allow fine tuning of the users interface without recompiling the entire program.

The code generated by Frigg are specialisation's of the classes in GUIenv. Figure 1 shows a FindDialog created in Frigg, as it appears at runtime.

The BETA AST that representing the FindDialog is pretty-printed in its textual form below:

```
ORIGIN '~beta/guienv/v1.3.1/guienvall';
BODY 'finddialogbody'

-- guienvLib: Attributes --

findDialog: window
  (#
     onFind:< (# str: ^text;  enter str[] do INNER #);
     onCancel:< (#  do INNER #);

     open::<
       (#
       <<SLOT findDialogOpen:DoPart>>
       #);
     private: @<<SLOT findDialogPrivate:Descriptor>>;
  #)
```

The FindDialog fragment group

The first three lines are the fragment syntax, which is what makes the fragment group a GUIenv library that can be used in any GUIenv program.

The FindDialog pattern is a specialisation of the GUIenv window pattern because the FindDialog is a window and therefore inherits from the window class.

The interior of the FindDialog is hidden in the implementation file called a BODY file. This allows the developer to change the content of the FindDialog without recompiling the part of the program which uses the dialog. The user interface objects in the window are declared in the private part of the FindDialog pattern as individual singular objects.

The parts of the FindDialog that are automatically generated is the further binding of "`open`" and the "`private: @<<...>>`" declaration. The two virtual procedures "`OnCancel`" and "`OnFind`" are added by the developer as the interface between the FindDialog and the application. The "OnFind" virtual procedure is called when the user presses the "find" button in the dialog. The argument to "OnFind" is the contents of the text field in the dialog. Below is a pretty-print of the AST representing the find-button in the private part of the dialog:

```
findBtn: (*$ 7*) @pushButton
  (#
     open::< (# do (* initialize *) #);
     eventHandler::<
       (# onMouseUp::< (# do searchFld.contents -> onFind;  #) #)
  #)
```

The find button of the dialog.

The only part of the FindBtn singular object which is added by the developer, is the dopart of the OnMouseUp virtual. The OnMouseUp virtual procedure is automatically called by the GUIenv system, when the user presses the button. Here the developer has programmed the button to call the "OnFind" virtual procedure in the interface of the FindDialog.

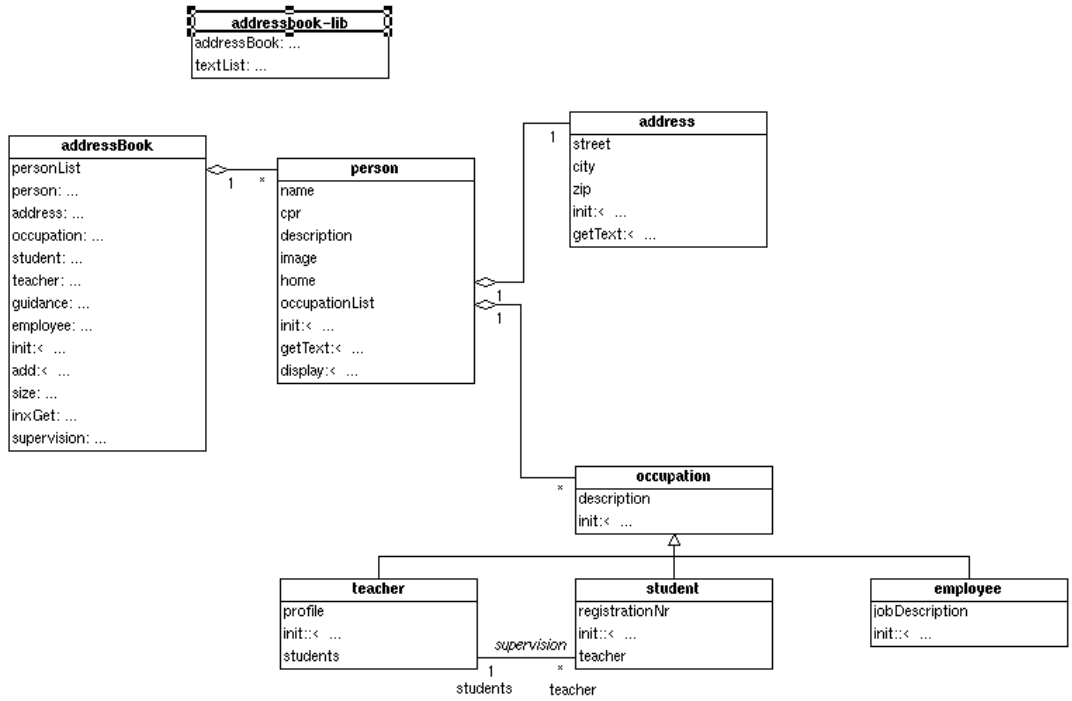
2. Tutorial

2.1 An Example Application

This tutorial will demonstrate how to use Frigg to make a small address book application. The application is structured into three parts:

`addressbook`	The classes that defines the data objects in the application.
`addressbookgui`	The classes that defines the graphical user interface.
`addressbookappl`	The controlling application that ties the data model to the graphical user interface.

The data model is created in Freja (Mjølner BETA CASE tool). The following diagram describes the data model for the addressbook application:



The Freja manual can be consulted to get an explanation of the graphical notation used in the diagram.

Here is a short description of the classes in the diagram:

`addressbook`	Has a list of persons.
`person`	Has a number of simple attributes: name, CPR etc., a reference to the current address, and a list of occupations.
`address`	Simple attributes: Street, city etc.

`occupation` Can be teacher, student and employee. This part of the model is left out of the application for simplicity.

The addressbook model does not have visibility to the addressbookgui (i.e. it does not INCLUDE the addressbookgui fragmentgroup). This ensures that the addressbook data objects can be made persistent objects.

The addressbookgui does not know about the addressbook data model. This allows the addressbook user interface to be reused in other applications.

The addressbookappl includes both the data model and the user interface, and ties the two together.

The following sections will explain how to create the user interface and the application.

2.2 Creating the User interface

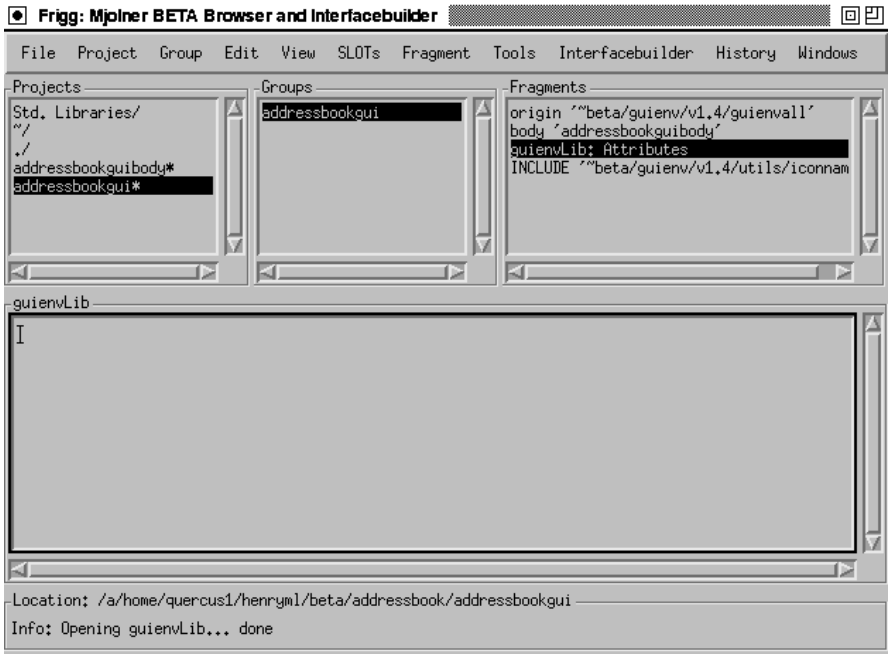
2.2.1 Creating the adressbookgui fragmentgroup

The sourcebrowser that are part of Frigg are activated by writing

```
frigg
```

in the command line (UNIX) or by double-clicking the Frigg icon (Macintosh).

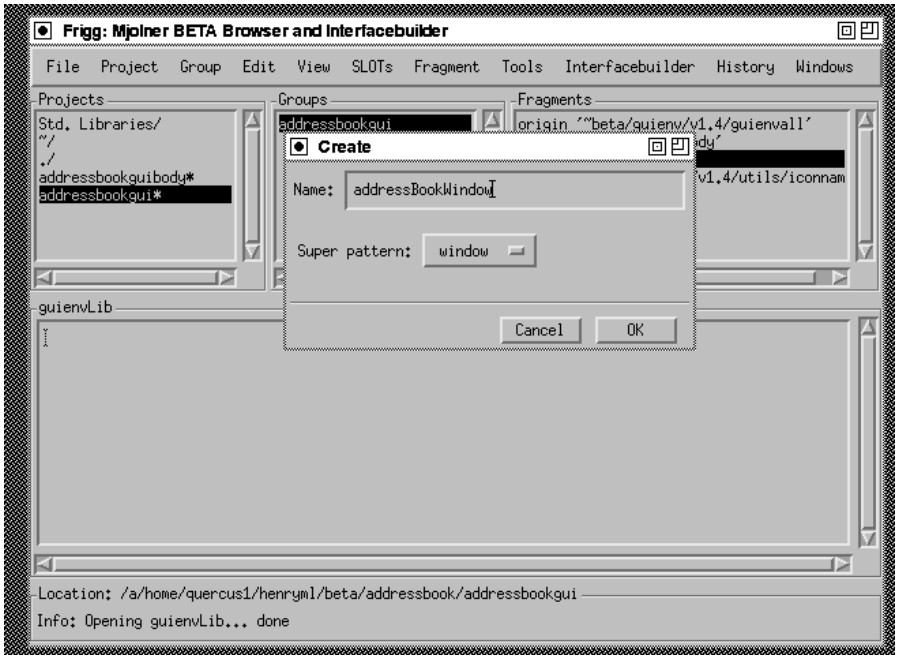
Activating Frigg, the sourcebrowser window will appear:



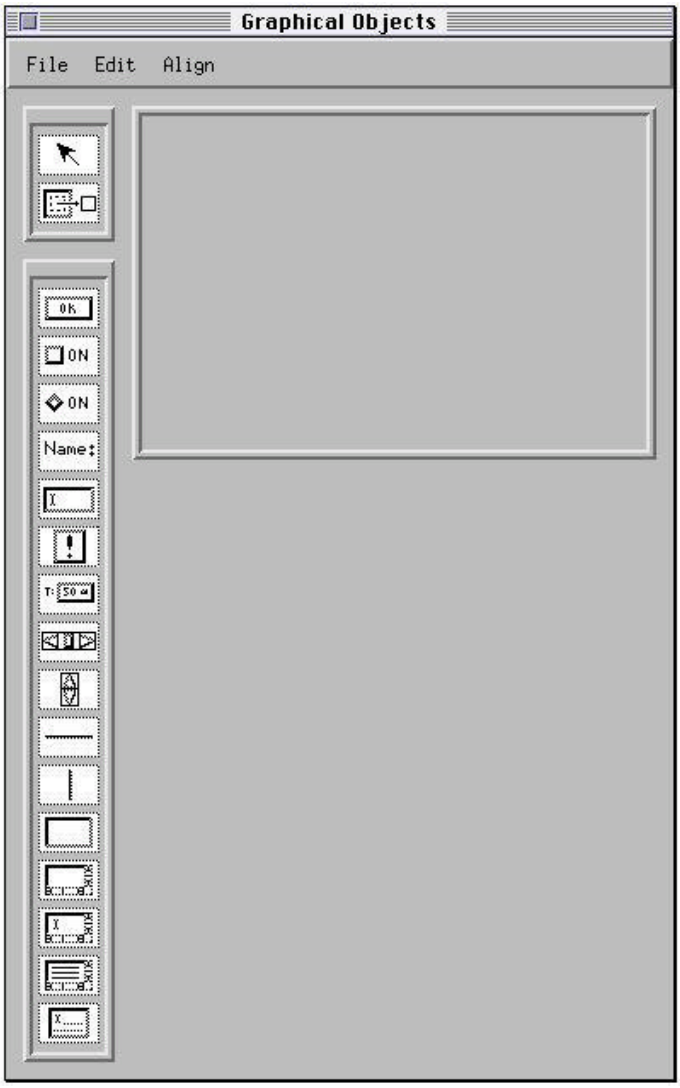
Here the addressbookgui fragmentgroup is already created. This was done by using the **New Window Library** command in the **Interfacebuilder** menu. This fragmentgroup has ORIGIN in guienvall and contains a GUIenvLib fragmentform. The GUIenvLib fragmentform can contain specializations of the window class in Lidskjalv (The Mjølner BETA user interface framework).

2.2.2 Creating the addressbook window

The addressbook window are created as a specialization of window via the **Create Window** command in the **Interfacebuilder** menu:



In the *create* dialog the name of the window are entered and the *inherits from* popup menu are set to *window*. Pressing OK in the dialog the graphical editor window appears:

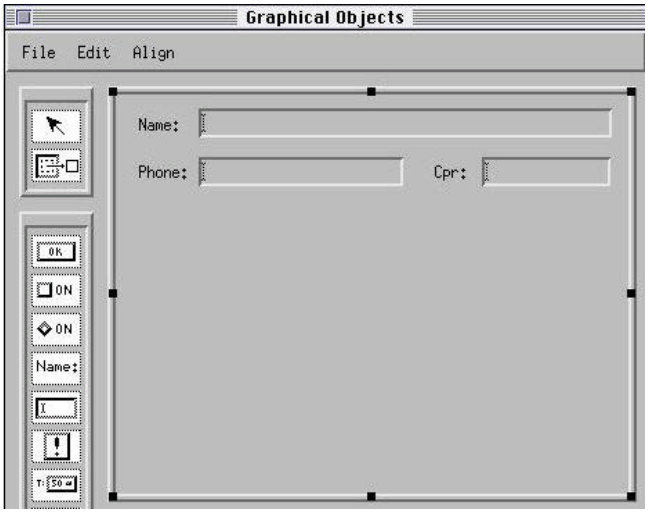


The empty area with a border is the content area of the window. This area has been resized to the desired size by dragging the border.

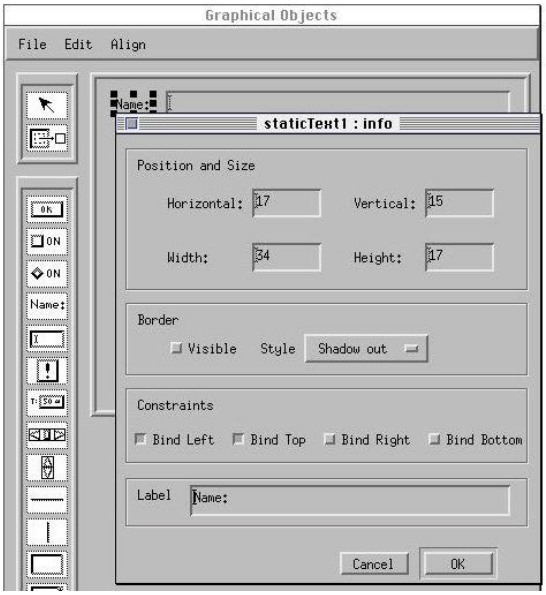
The palette to the left of the content area contains the standard Lidskjalv window items, such as push-button and text field.

2.2.3 Adding items

The first items that will be added to the window, is simple text label and text fields. This is the fourth and the fifth item on the simple item palette. The items are added by dragging the items from the palette to the content area of the window.



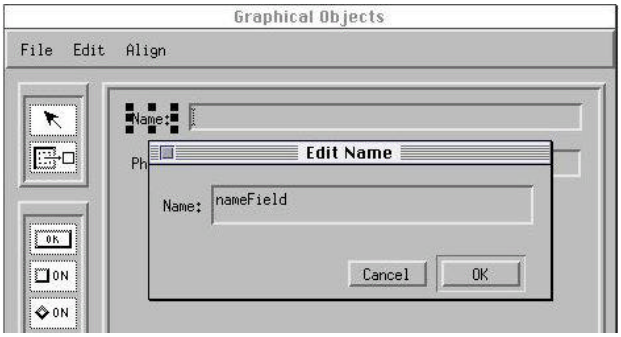
Here the items are added, and the text in the labels are changed to “Name:”, “Phone:” and “CPR:”. These changes are made via the “Object Info Dialog” that are invoked by selecting an item and then choosing the “Object Info” in the “Edit” menu:



The items are arranged in the window by using the alignment commands in the “Align” menu.

2.2.4 Changing the Name

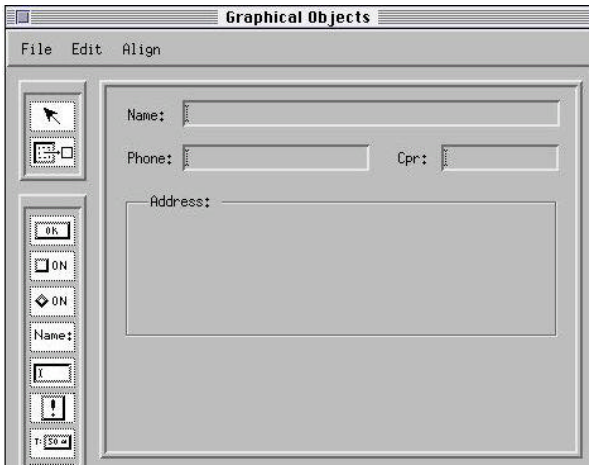
The items are given default names in the source code that looks like “editText1” and “staticText2”, but since these items needs to be referred to in the application (in order to tie the user interface to the data model) it is a good idea to change the names. This is done by choosing the “Edit Name” command in the “Edit” menu:



Here the name of one of the text fields are changed to “nameField”, which will be easier to remember later.

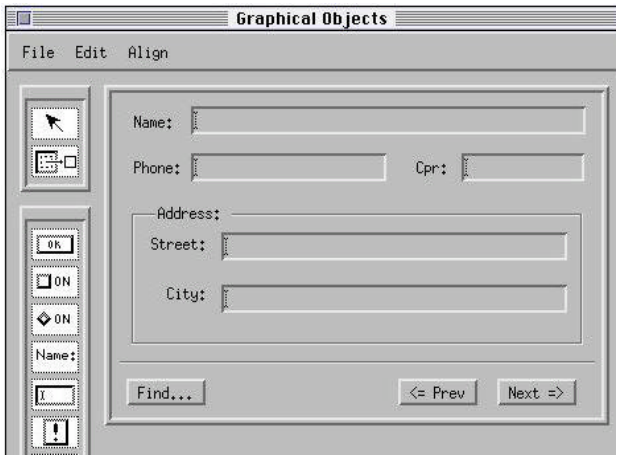
2.2.5 Compound items

The address part of the addressbook window are going to be a *canvas*, which is a compound item. This is the fifth item from the bottom on the palette. A canvas contains other items in its local coordinate system. Initially the canvas is empty, and then items can be added to the canvas. These objects move when the canvas are moved.



Here the address canvas has been added. The object info dialog has been used to give the canvas an “etched in” border. The “Address:” label are not part of the canvas, but are added to the window.

The address fields “street” and “city” can now be added to the canvas by dragging from the palette to the canvas.



Now the address fields are added. At the bottom of the window three push-buttons are added. They perform the main functions in the addressbook window. The other functions can be put in the menubar.

NOTE: The menubar can not be specified in the graphical editor. It will have to be coded in Sif, see the Lidskjalv manual for information about doing this.

Furthermore a separator is placed between the buttons and the “address” canvas.

Now the user interface is complete. In the next section one way to tie the user interface and the data model together in an application, will be explained.

2.3 Building the Application

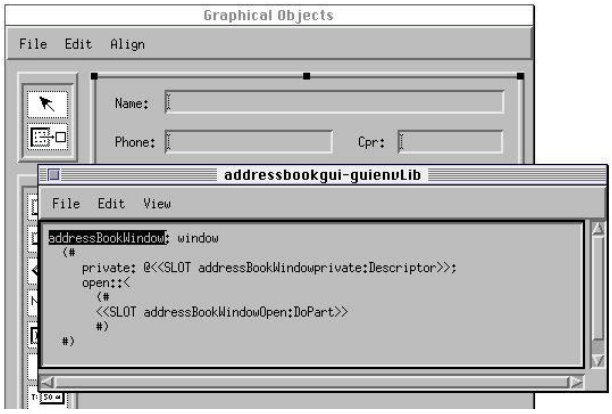
2.3.1 Structure of Generated Code

The code generated by Frigg is divided into two different files:

`addressbookgui` The public accessible portion of the window i.e. the declaration of the addressbook window.

`addressbookguibody` The private attributes of the window.

The addressbookgui fragmentgroup are named by the user when choosing the “New Window Library” command. The body file is created by Frigg. A Sif editor is opened on the declaration of the addressbook window by selecting the content area of the window and choosing “Open Subeditor” in the “Edit” menu:



All the items of the window are hidden in the private attributes of the window. These attributes can be edited in Sif by double-clicking

<<SLOT addressBookWindowPrivate: descriptor>>

The addressBookWindowPrivate fragment is located in “addressbookguibody". Here is a portion of this fragmentgroup:

```
ORIGIN 'addressbookgui'
-- addressBookWindowPrivate: Descriptor --
(# ...
   nameField: @editText
     (# open::<
          (#
          do (346,30)->size;
             (54,14)->position;
          #)
     #);
   addressGroup: @canvas
     (# streetField: @editText
          (# ... #);
        ...
     #);
   ...
#)
```

2.3.2 Extending the Public Interface

The interface of the addressBookWindow needs to be extended to allow the application to access certain parts of the implementation of the window.

In this example two principles will be used:

Text Fields: For each text field in the addressBookWindow a simple attribute is added to the interface of addressBookWindow that changes the content of the text field.

Push buttons: For each push-button, a virtual procedure is added that is called when the user presses the button.

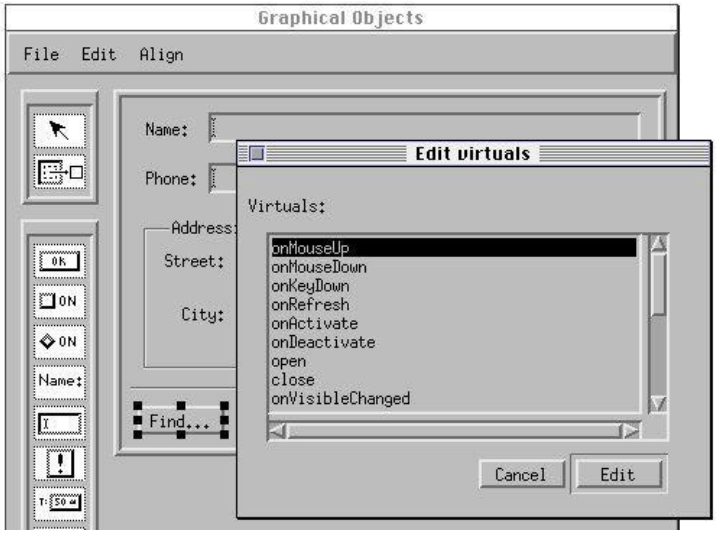
In the following code this is done for “nameField” and “findBtn”:

```
addressBookWindow: window
  (#
     nameField:
       (* Changes the content of the textfield “NameField” *)
       (# theName: ^text
       enter theName[]
       <<SLOT enterNameField:DoPart>>
       #);

     onFind:<
       (* Is called when the user presses the “Find” button *)
       (#  do INNER #);
     ...
  #)
```

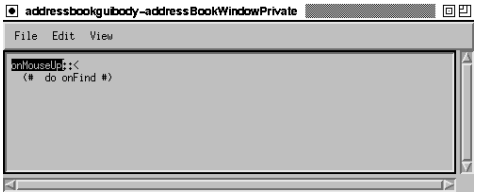
The implementation of <<SLOT enterNameField:DoPart>>, must be placed in the “addressbookguibody”. The implementation looks like this:

```
-- enterNameField:DoPart --
do theName[] -> private.nameField.contents
```

The “find” button needs to call the “onFind” virtual. This is done by selecting the “find” button in the graphical editor and choosing “Edit Virtuals” in the “Edit” menu.



The invoked dialog presents the available event types of the push-button. “onFind” should be called in the “onMouseUp” event. Selecting “onMouseUp” and pressing “edit” will open a Sif editor on a furtherbinding of “onMouseUp”.



Here “onMouseUp” is furtherbound to call “onFind”.

2.3.3 The Application

The application module, “addressbookAppl” includes both the user interface and the data model.

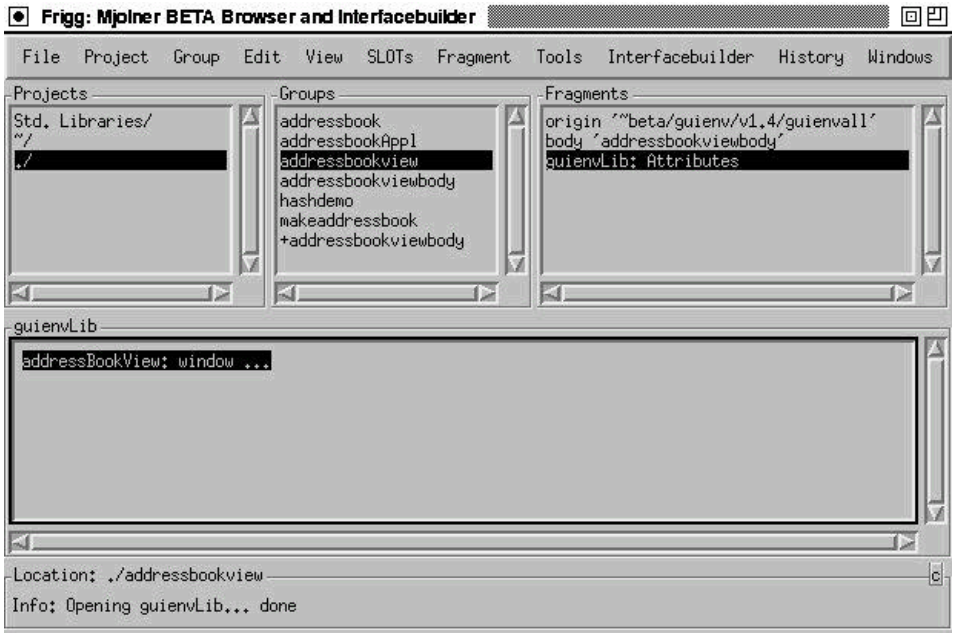
```
ORIGIN 'addressbookgui';
INCLUDE '~beta/persistentstore/v1.5/persistentstore';
INCLUDE 'addressbook';
-- program: Descriptor --
GUIenv
(# PS: @persistentStore;
     (* Use persistentstore to store the data objects *)
   theAddressBook: ^addressBook;
   theAddressBookWindow: @addressBookWindow
     (# currentInx: @integer;

        showPerson:
          (# thePerson: ^addressBook.person;
          enter thePerson[]
          do (*
              * Change the content of the name field etc.
              *)

             thePerson.name[] -> nameField;
             ...;
          #);

        onFind::
          (# do (* Invoke a find dialog *) #);

        open::
          (#
          do (*
              * Fetch the first person
              *)

             1 -> currentInx
               -> theAddressBook.inxGet -> showPerson;
          #);
     #)
do (*
    * Get the addressbook data in “uni”
    *)

    'uni' -> PS.openWrite;
   ('AddressBook', addressBook##)
       -> PS.get -> theAddressBook[];

   (*
    * Open the addressbook window
    *)

   theAddressBookWindow.open;
#)
```

Here the extended interface of the addressBookWindow are used to make the data model and the graphical user interface work together. The interface to the text fields in the window are used to show a person in the address book window. The virtual procedure “onFind” are furtherbound to invoke a “find” dialog. Persistentstore is used to retrieve the addressbook data objects.

3. Reference Manual

3.1 How to Get Started

Frigg is fully integrated with the sourcebrowser and structure editor. When Frigg is started, the sourcebrowser window is presented to the user. The “Interfacebuilder” menu contains commands to create windows that can be edited in a graphical editor.



3.1.1 Interfacebuilder Menu

New Window Library

This command creates a new fragmentgroup that are setup to contain window definitions. The ORIGIN of the fragmentgroup are the “guienvall” fragmentgroup, which includes all of GUIenv (Lidskjalv), the Mjølner BETA user interface framework. The fragmentgroup has a “GUIenvLib: attributes” fragment, which therefore can contain specializations of the window pattern from Lidskjalv. Furthermore a BODY fragmentgroup is automatically created. This fragmentgroup will contain the private attributes of the window definitions.

A file dialog prompts for a filename which will be the name of the new fragment group.

Assuming the name “foo” is used, the BODY group will have the name “foobody”.

New Canvas Library

This command does almost the same as the “New Window Library” command. The created fragmentgroup will have a “windowLib: attributes” fragment, which can contain canvas specialisations of the window pattern from Lidskjalv.

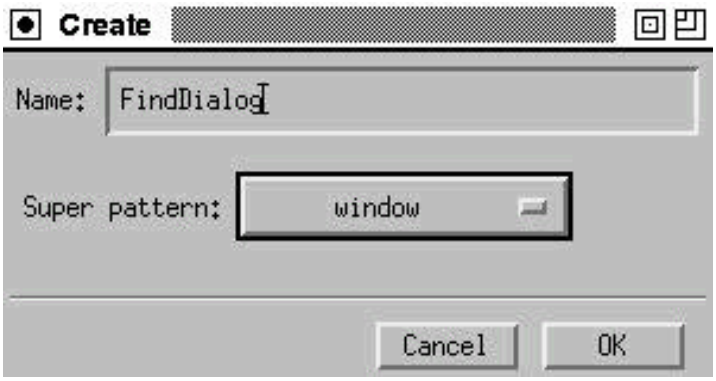
Edit

Selecting "edit" will open a graphical editor on the window pattern that are currently selected in the structure-editor. The whole pattern definition must be selected. It is not sufficient to just select the name.

NOTE: If the fragmentgroup is not checked, it is not always possible for Frigg to recognize the selected pattern as a user interface object. If the “Edit” command is disabled, the fragmentgroup should be checked via the “Check” command in the “Tools” menu.

Create

A specialization of window or canvas is generated in the currently selected lib fragment, dependent of the name of the fragment. If the current fragment is a GUIenvLib fragment, a window will be generated. If it is a windowLib fragment, a canvas will be generated. The following dialog are popped up:



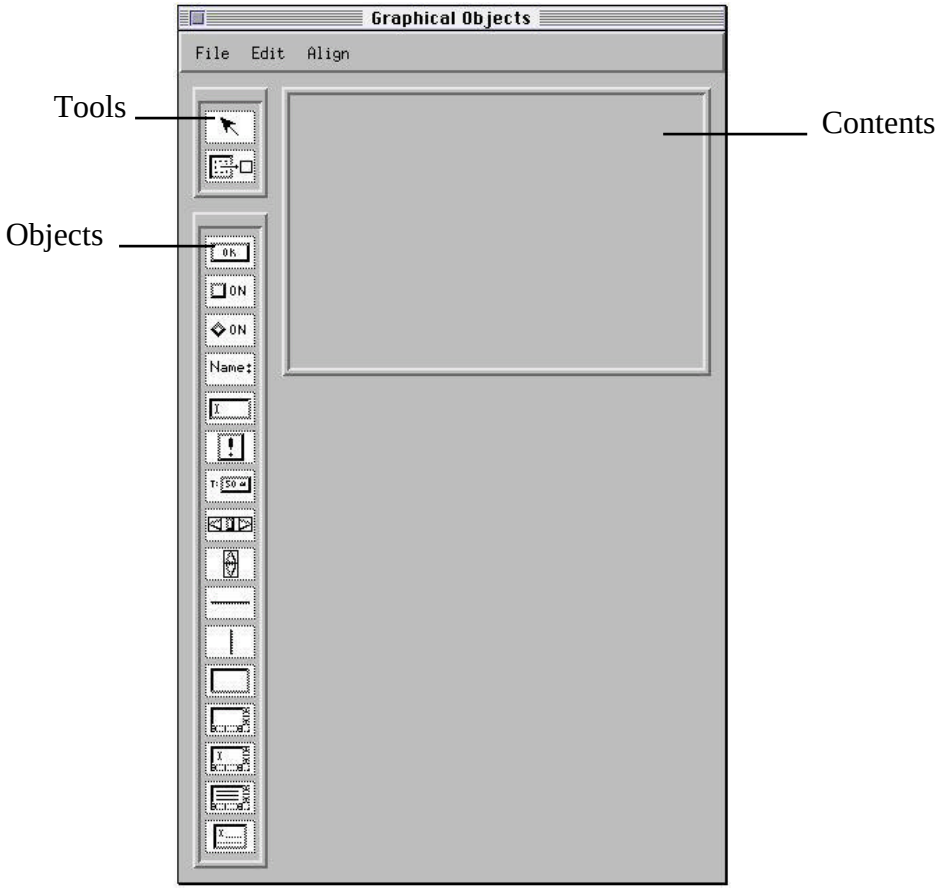
In the “name” field the name of the new window (or canvas) should be typed in. The name are expected to be a proper BETA identifier, since this is the name the pattern will given in the source code.

The “super pattern” popup menu allows the user to choose what pattern to inherit from.

A graphical editor is opened on the newly created object.

3.2 Graphical Editor

The graphical editor is the part of interface builder which allows the user interface to be constructed interactively by direct manipulation of Lidskjalv objects. The figure below shows the graphical editor on a newly created object.



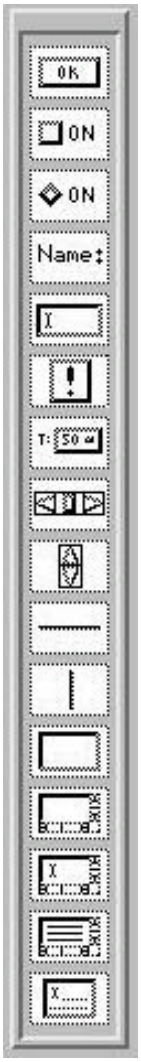
The menubar consists of three menus: File, Edit and Align. There are four areas inside the graphical editor: The tools palette, the object palette, and the contents area.

3.2.1 Creating Items

Items are added to the window by utilizing the two item palettes. An instance of some item on a palette is created by dragging the item and placing it in the contents area. The items are in an aggregation hierarchy. The canvas pattern has a border and contains other items in its local coordinate system.

When an item is dragged from the palette, the border of the receiving canvas is highlighted to indicate in which canvas the item will belong.

The figure below explains which pattern in Lidskjalv each item on the simple item palette corresponds to.



PushButton

CheckBox

RadioButton

StaticText

EditText

IconButton

OptionButton

Horizontal Scrollbar

Vertical Scrollbar

Horizontal Separator

Vertical Separator

Canvas

Scroller

TextEditor

ScrollList

TextField

These different patterns are described in the Lidskjalv reference manual.

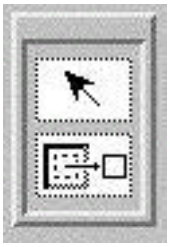
3.2.2 Moving and Resizing

The items in the graphical editor can be moved and resized simply by dragging with the mouse. If the item is grabbed in the interior the item is moved, if it is grabbed near the border it is resized.

3.2.3 Changing Hierarchy

The items in the window are arranged in a hierarchy. The canvas item is a compound item, that contains other items (including other canvas items). The items are positioned in the local coordinate system of the canvas it belongs to. The window itself is a kind of canvas in this respect.

The "Change Hierarchy" tool on the tools palette are used to change the hierarchy of a group of items.



Selection

Change Hierarchy

After the "Change Hierarchy" tool is chosen, a selection of items can be dragged to another canvas. When a selection of items is dragged, the border of the receiving canvas is highlighted to indicate in which canvas the items will end up.

3.2.4 File Menu

Close

Closes the graphical editor.

3.2.5 Edit Menu

Undo

All the operations in the graphical editor can be undone by choosing the "Undo" command in the edit menu. The undo is multilevel, which means that all changes can be undone all the way back to when the window was opened for editing in the session.

Redo

A sequence of undo-commands can be redone by invoking the redo command in the edit menu - as long as no other operation has been performed after the undoing.

Cut

A copy of the selected objects is placed on the clipboard along with the underlying BETA code, and then deleted from the window. The objects can then be pasted into the window again. It is possible to copy and paste between graphical editors.

Copy

A copy of the selected objects is placed on the clipboard along with the underlying BETA code. The objects can the be pasted into the window again. It is possible to copy and paste between graphical editors.

Paste

A copy of the selected objects is placed on the clipboard along with the underlying BETA code. The objects can the be pasted into the window again. It is possible to copy and paste between graphical editors.

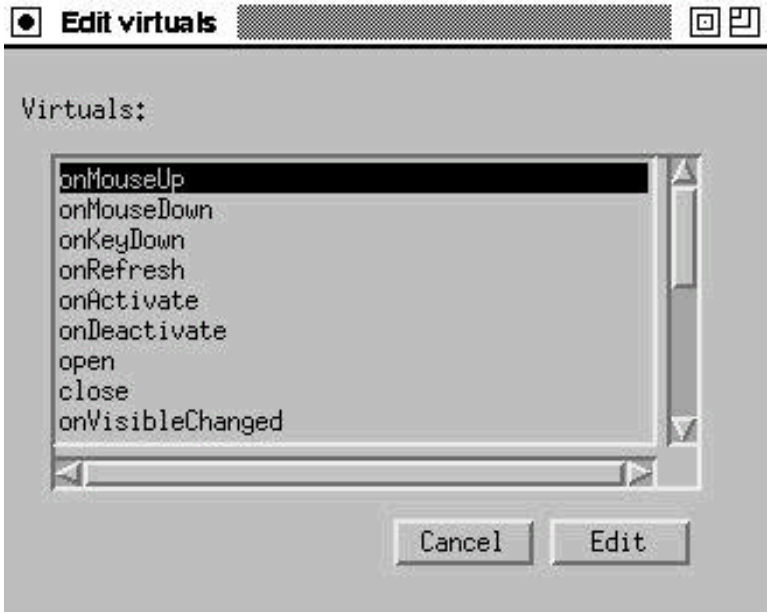
Delete

The selected objects along with the underlying BETA code are deleted without affecting the clipboard.

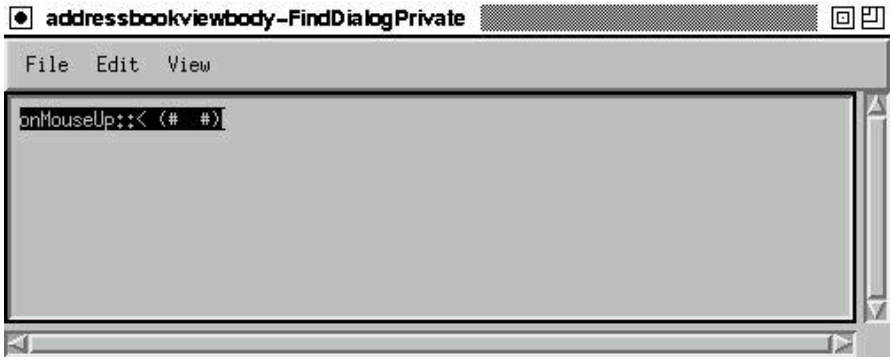
Edit Name

This command will invoke a dialog in which the name (in the source code) of the selected object can be entered. The name is required to be a legal name in BETA.

Edit Virtuals



Invokes a dialog in which the different event types of the selected object are listed. When the user clicks on an object, the virtual procedure "onMouseUp" are executed in that object. There is a virtual procedure for each event that the object may receive. The "edit" command in the dialog opens a structure editor (sif editor) on the virtual procedure corresponding to the selected event type. If the specialization of the virtual procedure does not yet exist, it is first be created.

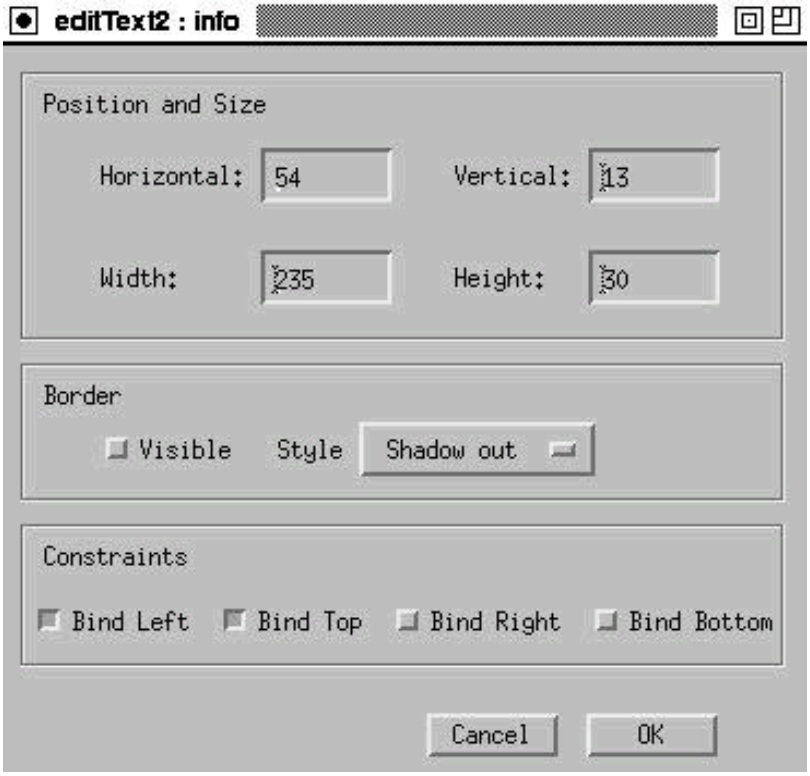


Open Subeditor

Opens a structure editor (sif editor) on the code that corresponds to the selected object.

Object Info

The object info dialog allows the layout properties of the objects in the graphical editor to be edited in a dialog form in contrast to direct manipulation. Most properties can only be changed via the dialog. The content of the dialog depends on the selected object. The basic info dialog has the properties that are common to all objects.



Position and Size: The position and the size are most easily changed via direct manipulation, but sometimes it may be easier to enter the precise values via the dialog

Border: All objects have a border. The visibility of the border can be controlled via the "visible" check box. There is different kinds of shaded border types, that can be selected in the "style" popup menu.

Constraints: The constraints control how the object reacts when the surrounding object are resized. If the bindleft is false and bindRight is true, the object will follow the right edge of the surrounding object without stretching. If bindLeft is true and bindRight is true, the object will stretch etc.

See the "Object Info Dialog" section for more details.

Fit to Contents

Frequently, there is a natural size of an object, dependent of the content of the object. For example a StaticText object would have the extent of the text as the natural size. The "Fit to Contents" command in the edit menu will adjust the size of the selected object to its natural size. Not all objects have natural sizes.

3.2.6 Align Menu

The alignment commands in the alignment menu supply facilities to align objects in a row or centered underneath each other etc. The alignment commands work on the current selection. The first object selected will stay where it is.

These alignment commands are available:

Align left side
Aligns the left sides of the selected objects to the first selected object.

Align right side
Aligns the right sides of the selected objects to the first selected object.

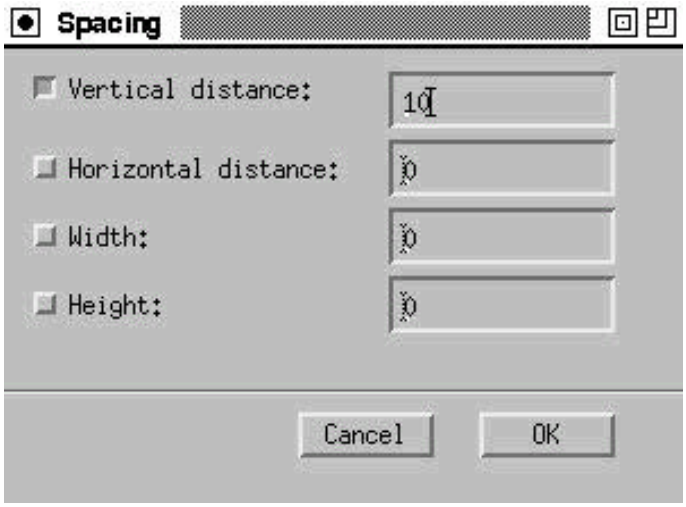
Align top edge
Aligns the top edges of the selected objects to the first selected object.

Align bottom edge
Aligns the bottom edges of the selected objects to the first selected object.

Align vertical center
Aligns the vertical centers of the selected objects to the first selected object.

Align horizontal center
Aligns the horizontal centers of the selected objects to the first selected object.

Spacing...
This command will present a dialog that allows a group of objects to be given the same size or the objects to be arranged, so the distance between any two adjacent objects are the same.



To give a group of objects the same vertical distance do the following:

1) Select the objects that should be given the same vertical distance
2) Choose the Spacing... command
3) Check the "Vertical distance" check box and make sure the other check boxes are unchecked
4) Type the desired distance into the field next to the check box
5) Press OK

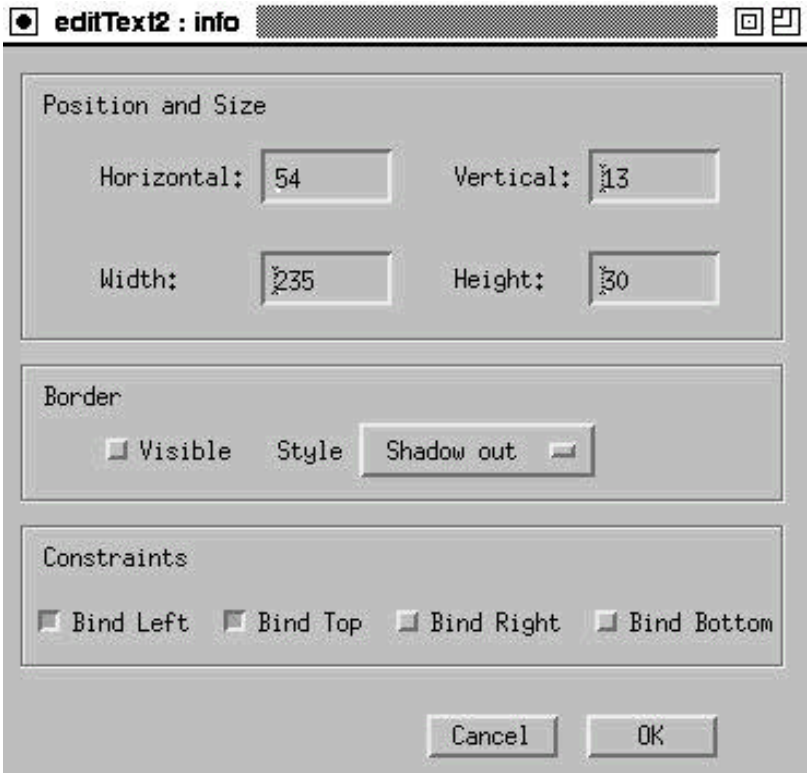
NOTE: The objects will keep their vertical and horizontal order in the window when adjusting the distance. Furthermore, the top left object will always stay where it is.

3.2.7 Object Info Dialog

The object info dialog allows the layout properties of the objects in the graphical editor to be edited in a dialog form in contrast to direct manipulation. Most properties can only be changed via the dialog. The content of the dialog depends on the selected object.

3.2.7.1 Basic Dialog

The basic info dialog has the properties that are common to all objects.



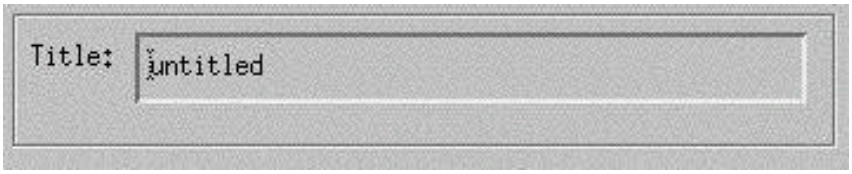
Position and Size: The position and the size are most easily changed via direct manipulation, but sometimes it may be easier to enter the precise values via the dialog

Border: All objects have a border. The visibility of the border can be controlled via the "visible" check box. There is different kinds of shaded border types, that can be selected in the "style" popup menu.

Constraints: The constraints control how the object reacts when the surrounding object are resized. If the bindleft is false and bindRight is true, the object will follow the right edge of the surrounding object without stretching. If bindLeft is true and bindRight is true, the object will stretch etc.

3.2.7.2 Window Dialog

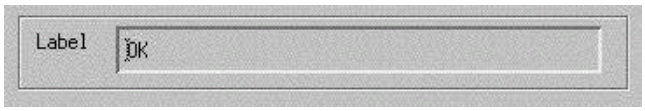
The object info dialog for "window" has a "title" property:



The title of the window is shown in the titlebar.

3.2.7.3 Button Dialog

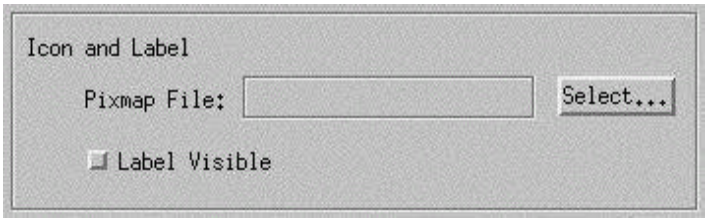
The dialog for Button and specializations of Button: PushButton, CheckBox, RadioButton, OptionButton, StaticText and IconButton, has a "label" property:



The label is a text that are visible on the screen. e.g. a PushButton is a text surrounded by a shaded border (Shadow out).

3.2.7.4 IconButton Dialog

The IconButton dialog has, in addition to the “label” field, the following portion:



The pixmap file contains the pixmap that are shown in the iconbutton. The format of the pixmap file are:

Unix: Xpm. Must have the extension “.xpm”.

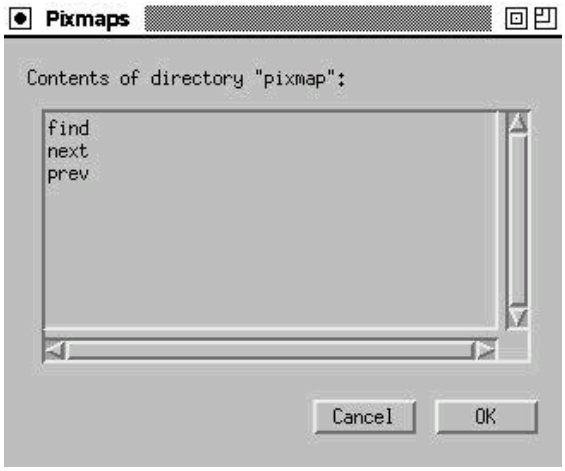
Windows: BMP. Must have the extension “.bmp”.

Macintosh: PICT. Must have the extension “.pict”.

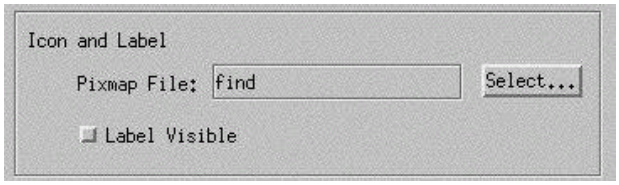
The application, xv version 3, can convert between various pixmap formats on UNIX. And “GraphicConverter” can be used on the Macintosh.

NOTE: The pixmap files for the application are required to be in a directory “pixmap”, which must be in the same directory as the main program.

Pressing the “select” button will invoke a pixmap selection dialog.



This dialog lists the content of the “pixmap” directory.



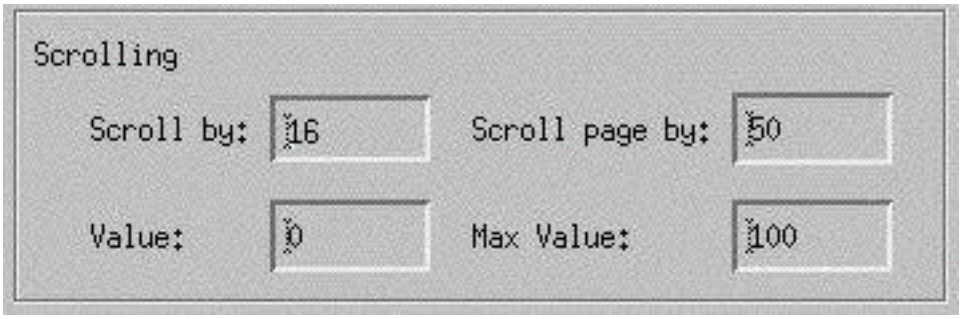
Here the “find” pixmap is selected.

3.2.7.5 Separator Dialog

The separator dialog allows the style of the separator to be chosen.

3.2.7.6 Scrollbar Dialog

The scrollbar dialog contains various scrolling parameters. See the Lidskjalv manual for more details.



4. Index

R

S

T

U

V

W