CS 535 Object-Oriented Programming & Design
Fall Semester, 2008
Doc 13 VW GUI
Oct 28 2010

References

Pattern-Oriented Software Architecture, Buschmann et al., 1996

VisualWorks GUI Developer's Guide, GUIDevGuide.pdf in the docs directory of the VW distribution

Dialog Windows

Windows used to

- Display information to the user
- Request information from user

VW Dialogs are modal

- User has to respond before application continues

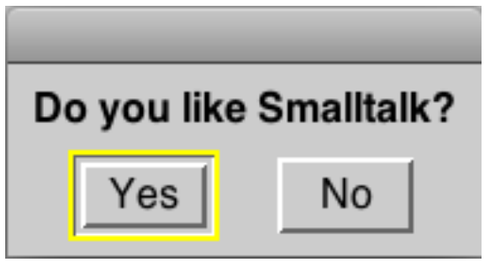
Warn

Dialog warn: 'This is a simple dialog window'.

Returns: nil

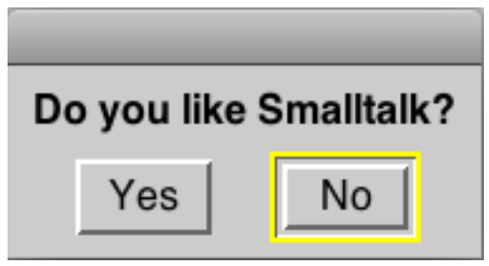
Confirm

answer := Dialog confirm: 'Do you like Smalltalk?'.



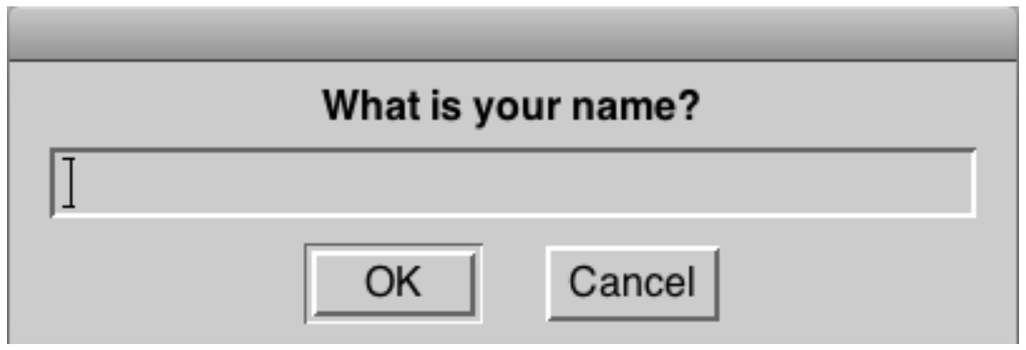
Returns: true or false

```
answer := Dialog
                confirm: 'Do you like
Smalltalk?'
                initialAnswer: false
```



Request

answer := Dialog request: 'What is your name?'



Returns:

Text entered on "OK"

Empty string on "Cancel"

Request Variations

```
Dialog
   request: 'What is your name?'
   initialAnswer: 'Smith'
```

```
Dialog
   request: 'What is your name?'
   initialAnswer: 'Smith'
   onCancel: ['Jones']
```

Choose

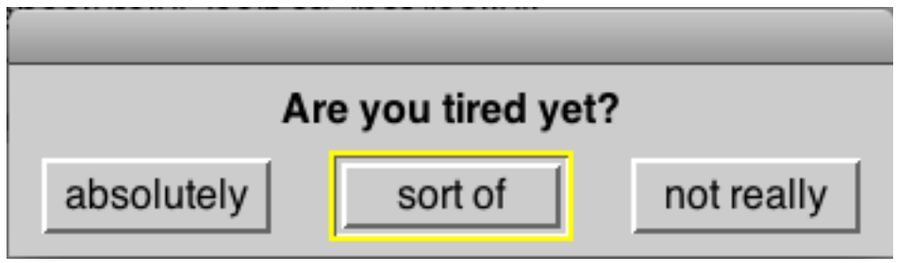
Dialog

choose: 'Are you tired yet?'

labels: #('absolutely' 'sort of' 'not really')

values: #(#yes #maybe #no)

default: #maybe

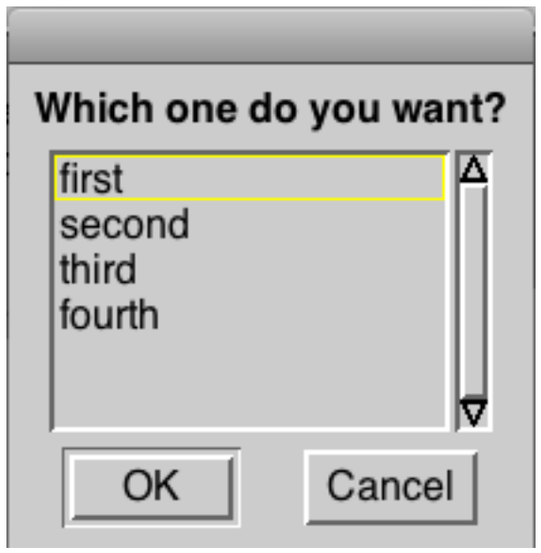


labels: what the users sees

values: value returned when user selects corresponding label

Choose from List

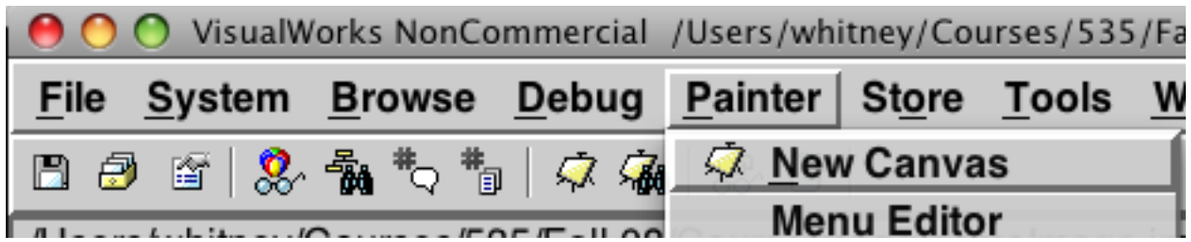
```
answer := Dialog
        choose: 'Which one do you want?'
        fromList: #('first' 'second' 'third' 'fourth')
        values: #(1 2 3 4)
        lines: 8
        cancel: [#noChoice]
```



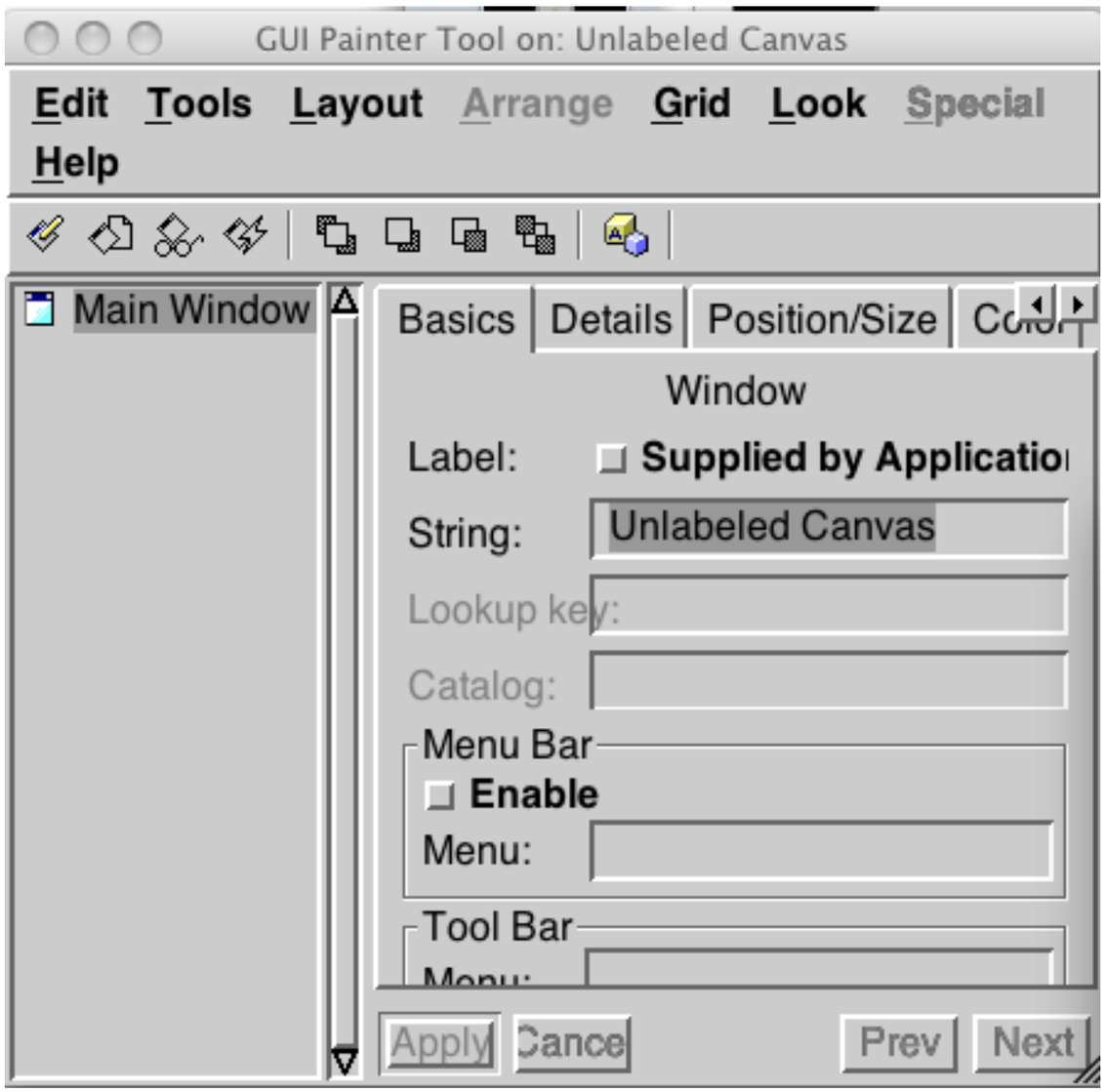
This example shows how to provide the user with a list of options. Both fromList: and values: keywords need a SequenceableCollection or subclass as an argument. The fromList: argument is the text to be displayed on the screen. The values: argument contains the value returned when the user selects an item. When the user selects the K'th item in the list, the K'th element of the values: argument is returned. The lines: keyword sets the maximum number of items that will fit in the window before the user has to scroll. The cancel: keyword requires a block as an argument. When the user cancels the dialog, the result of running the block is returned. While the block given here just returns a value, it could do more useful things like close files.

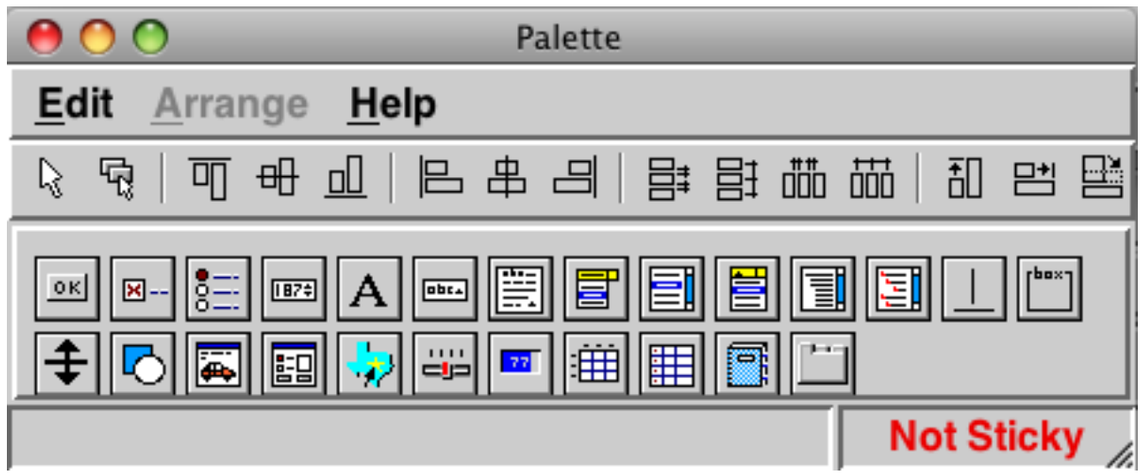
UIPainter

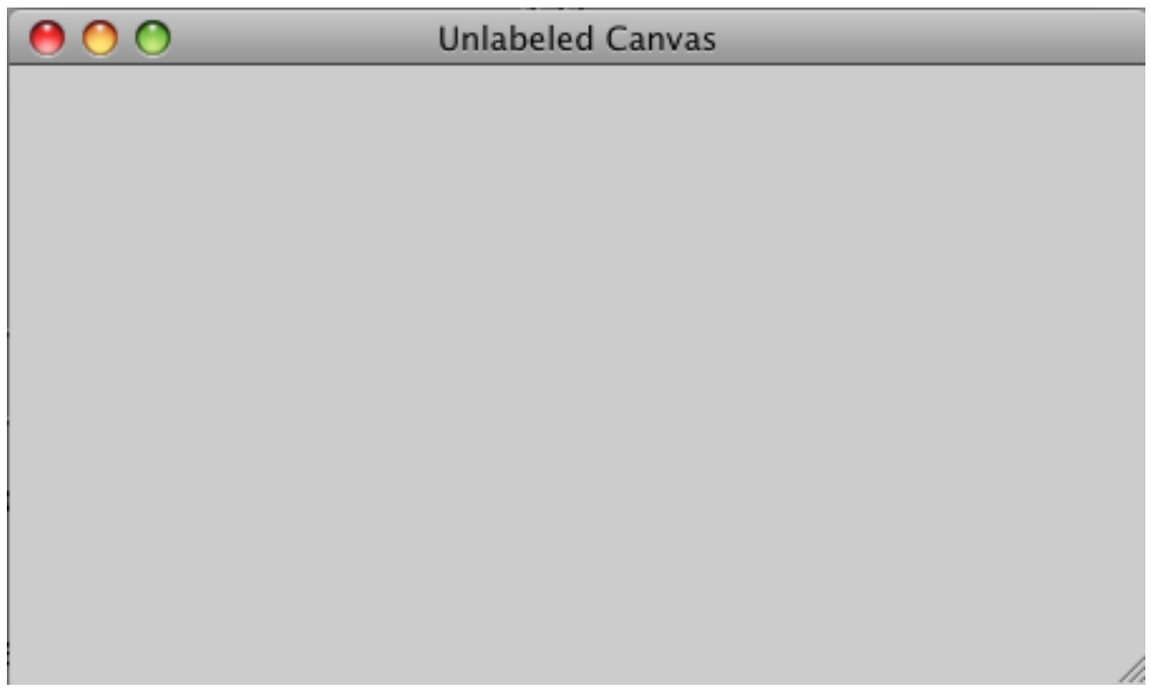
Graphical Tool used to develop GUIs



UI Painter Windows



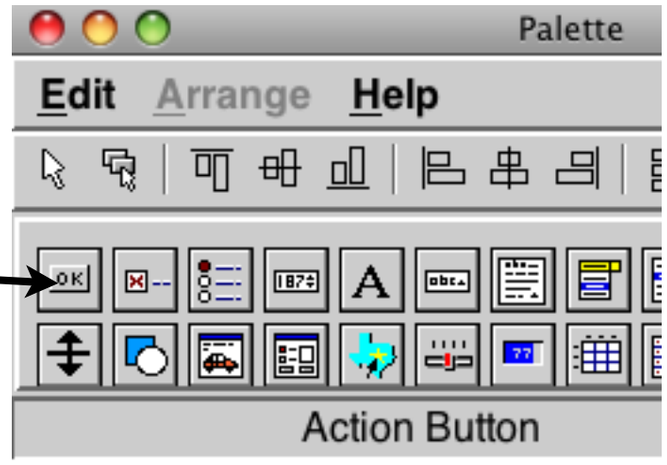




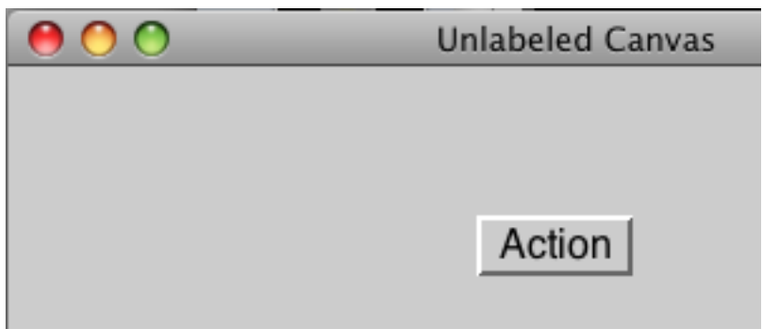
Palette - Widgets that we can put in the window
Unlabeled Canvas - Window we are constructing
GUI Painter Tool - Details about the widgets in our new window

Adding A Button

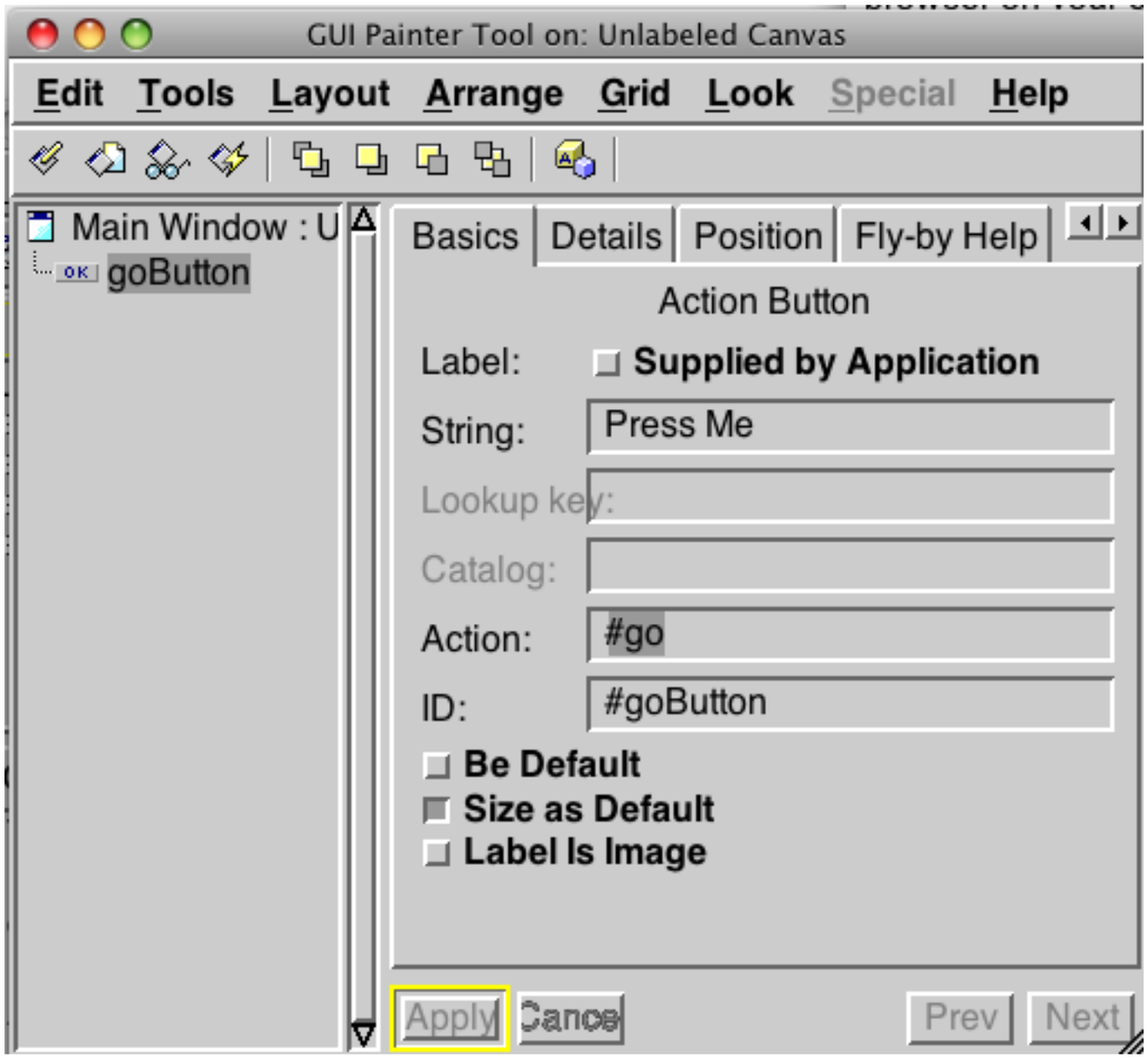
Click on "button" icon



Click in the canvas where you want the button

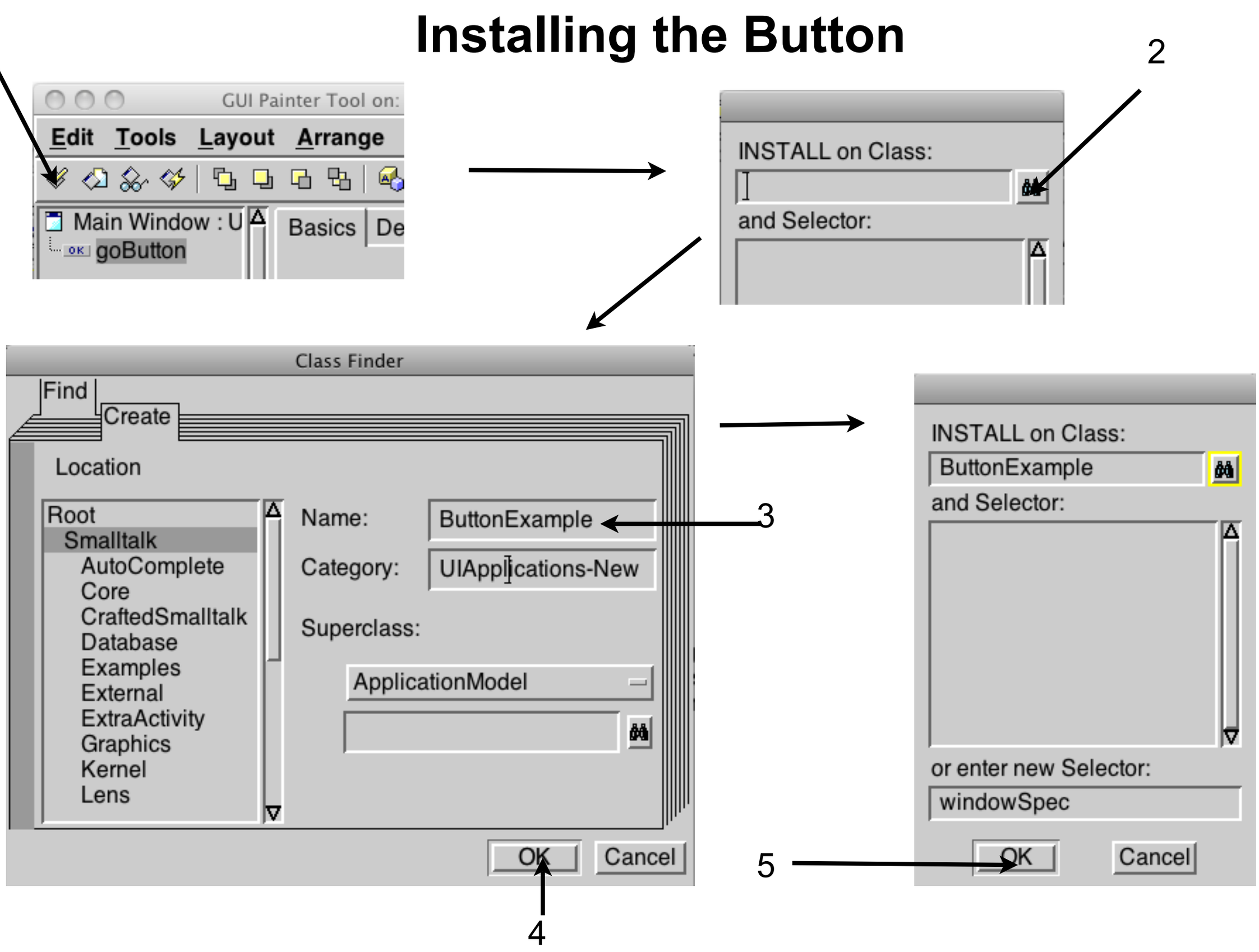


Configuring Button



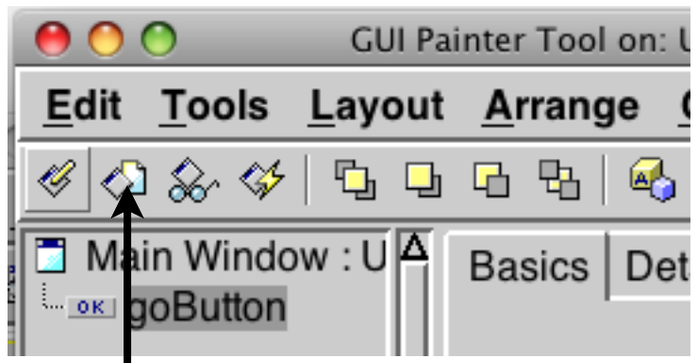
The GUI Painter Tool will also show information about button. In the GUI Painter Tool, change the String to "Press Me" and the Action: to "go". Actions must be symbols, but if you enter a string the tool will change it to a symbol. Now click on the "Apply" button. The string is the label of the button. The action is the method that will be called when the button is pressed.

Installing the Button

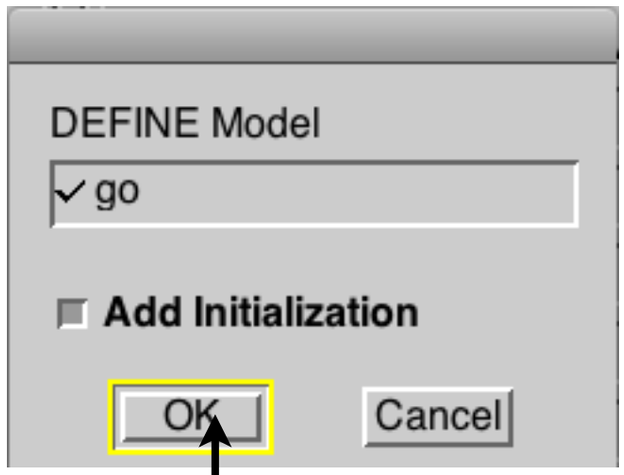


1. In the GUI Painter Tool click on the "Install..." button.
2. Click on the magnifying glass to open the class finder. We will create a new class.
3 Click on the "Create" tab. Click on the Examples namespace. Then enter the name of the class. tab out of the field
4 & 5. Click "OK"

Define the "go" method

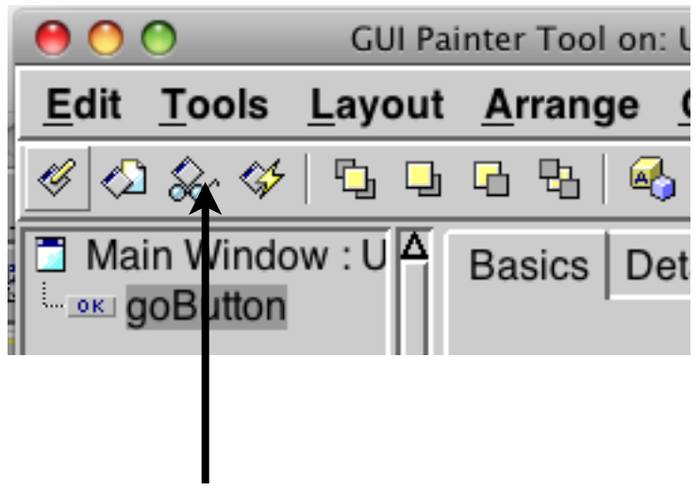


1



2

Edit the "go" method



1

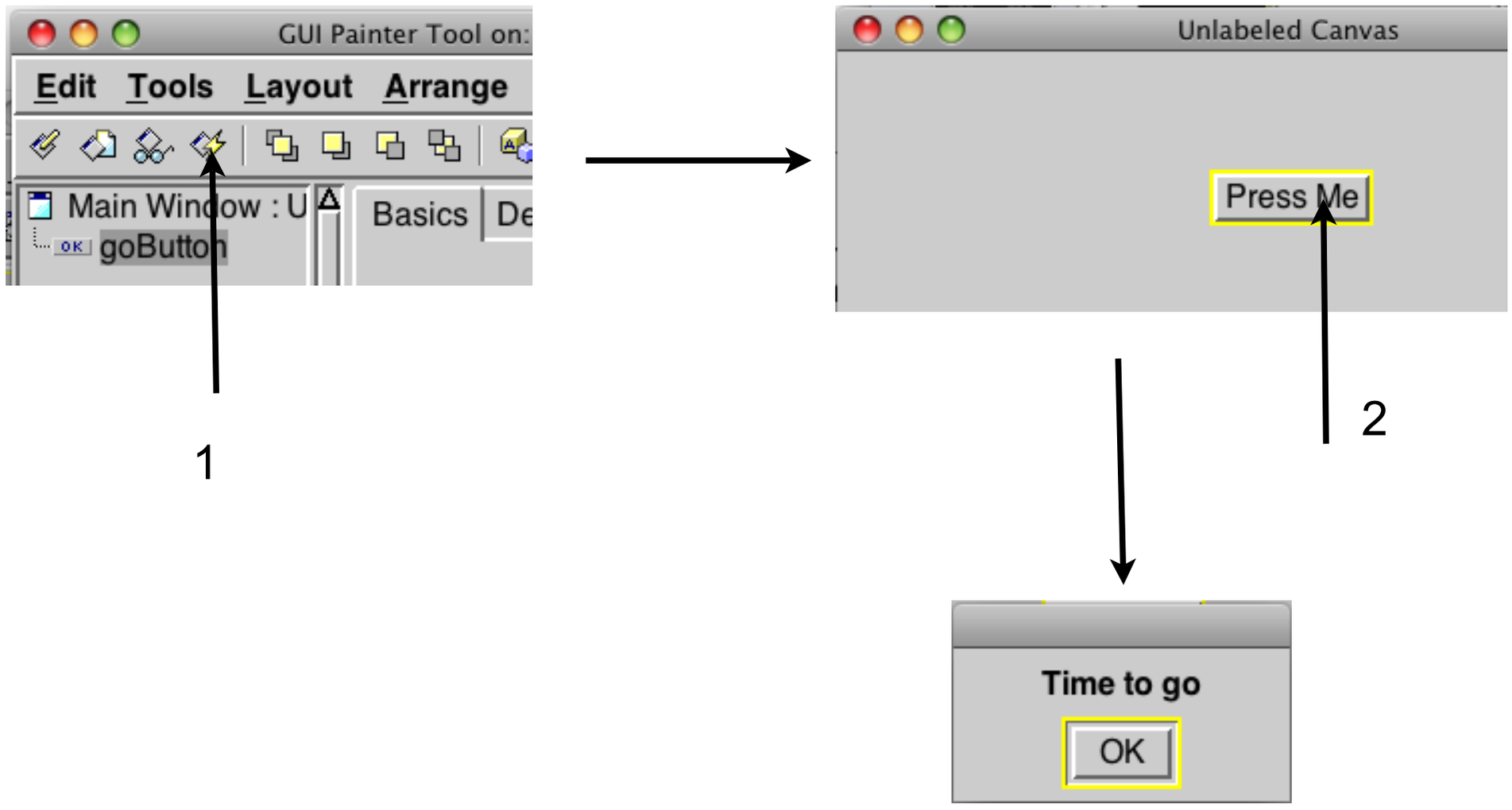
2. Edit go method in browser to be:

go

```
Dialog warn: 'Time to go'.
^self
```


1. Opens a browser on the go method

Running the Example



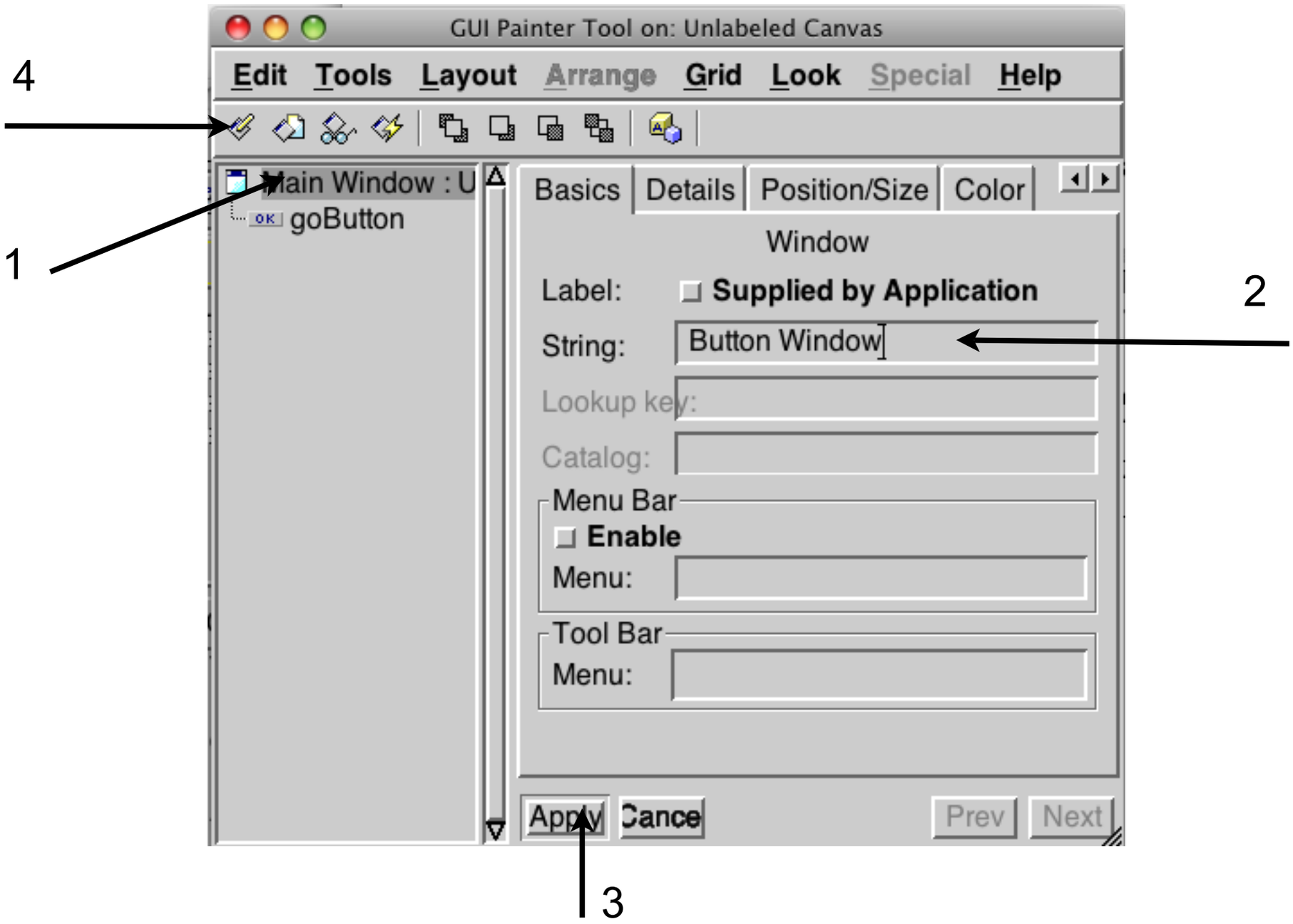
Using Code to Run the Example

ButtonExample open

WindowSpec

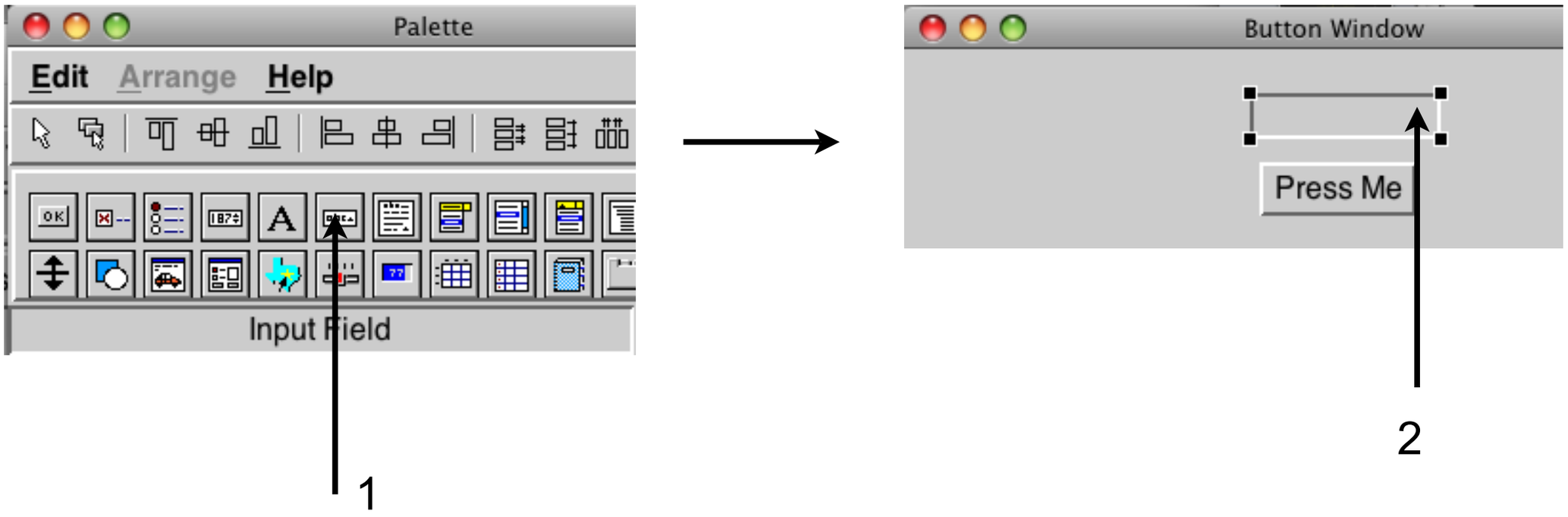
```
ButtonExample>>windowSpec
    "Tools.UIPainter new openOnClass: self andSelector: #windowSpec"

    <resource: #canvas>
    ^#(#{UI.FullSpec}
        #window:
        #(#{UI.WindowSpec}
            #label: 'Unlabeled Canvas'
            #bounds: #(#{Graphics.Rectangle} 556 524 1008 770 ) )
        #component:
        #(#{UI.SpecCollection}
            #collection: #(
                #(#{UI.ActionButtonSpec}
                    #layout: #(#{Graphics.Rectangle} 183 56 271 90 )
                    #name: #goButton
                    #model: #go
                    #label: 'Press Me'
                    #defaultable: true ) ) ) )
```

Window label



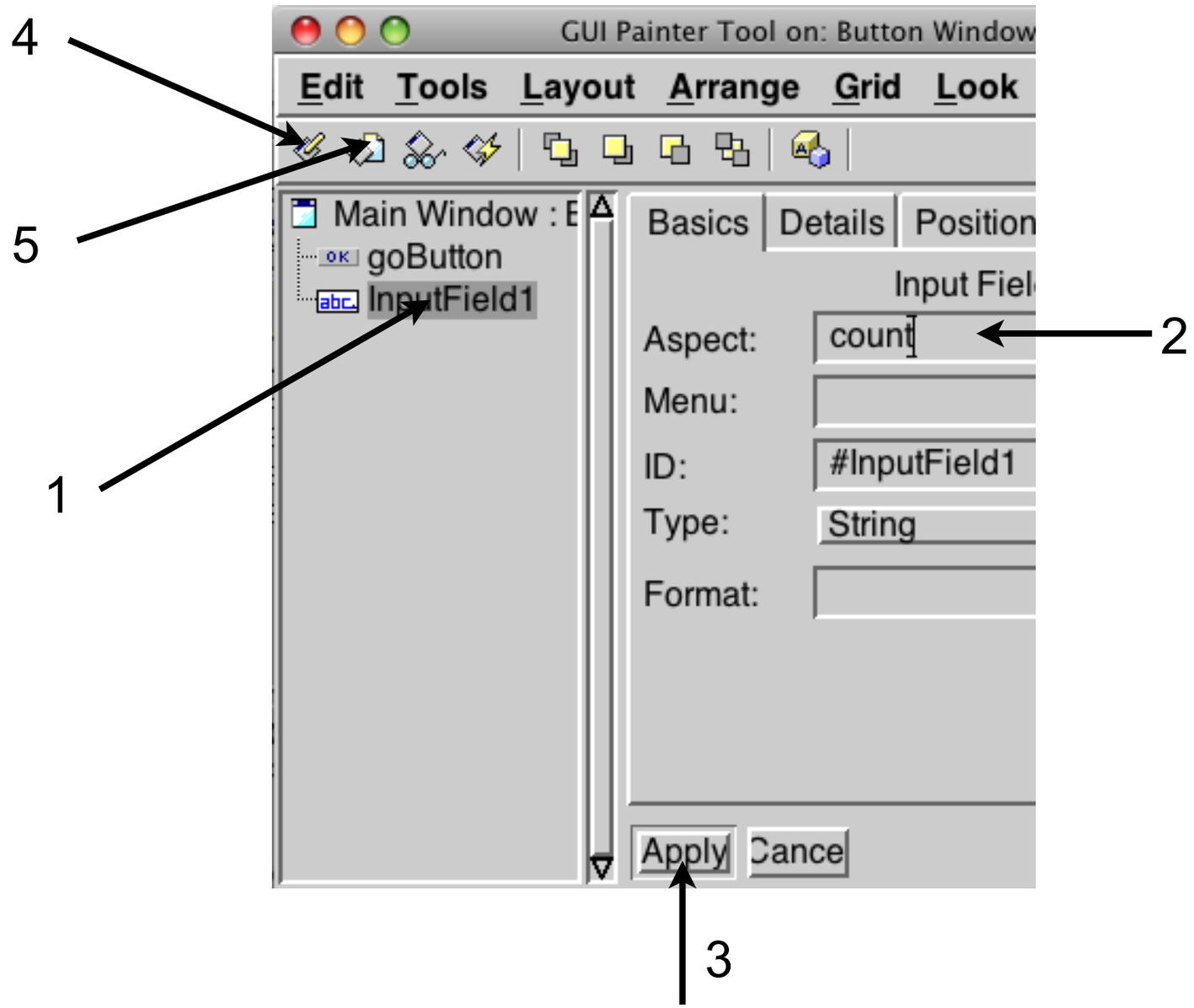
1. Select "Main window"
2. Enter Window label
3. Apply the change
4. Install the change in the ButtonExample class

Adding Text Input



1. Click on the "input Field" icon
2. Click in the Button Window where you would like the input field

Configuring Input Field



count method

```
ButtonExample>>count
	"This method was generated by UIDefiner.  Any edits made here
	may be lost whenever methods are automatically defined.  The
	initialization provided below may have been preempted by an
	initialize method."

	^count isNil
		ifTrue:
			[count := '0' asValue]
		ifFalse:
			[count]
```

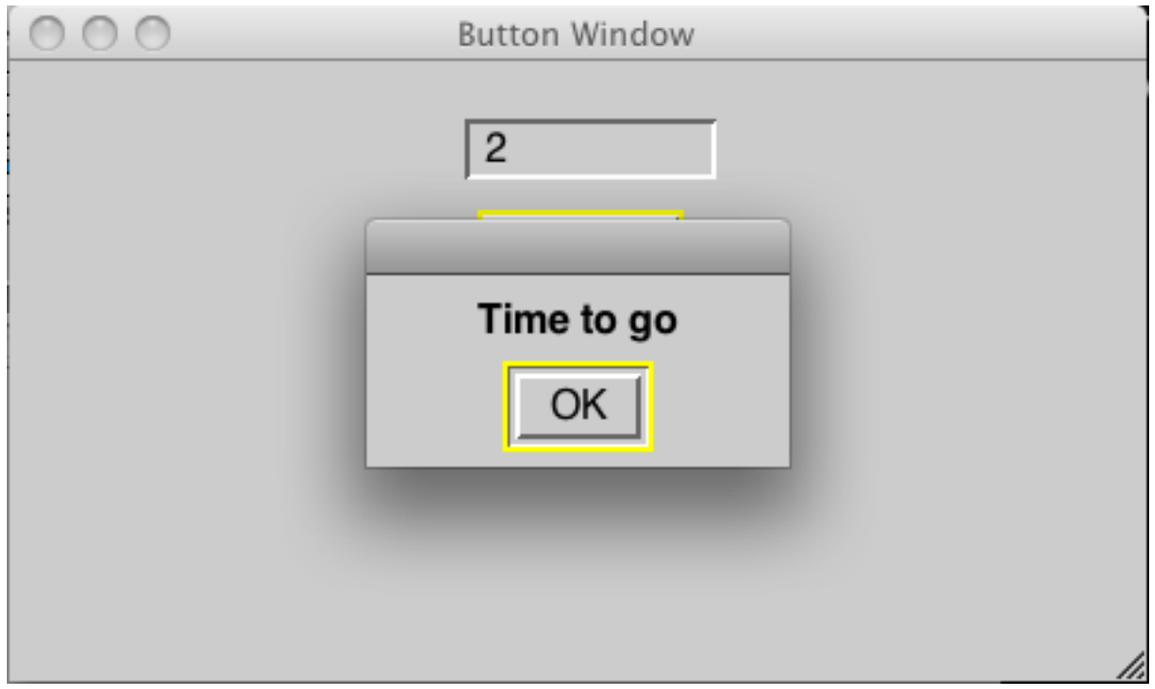

Modify the count method to be as above. That is change "String new asValue" to " '0' asValue"

go method

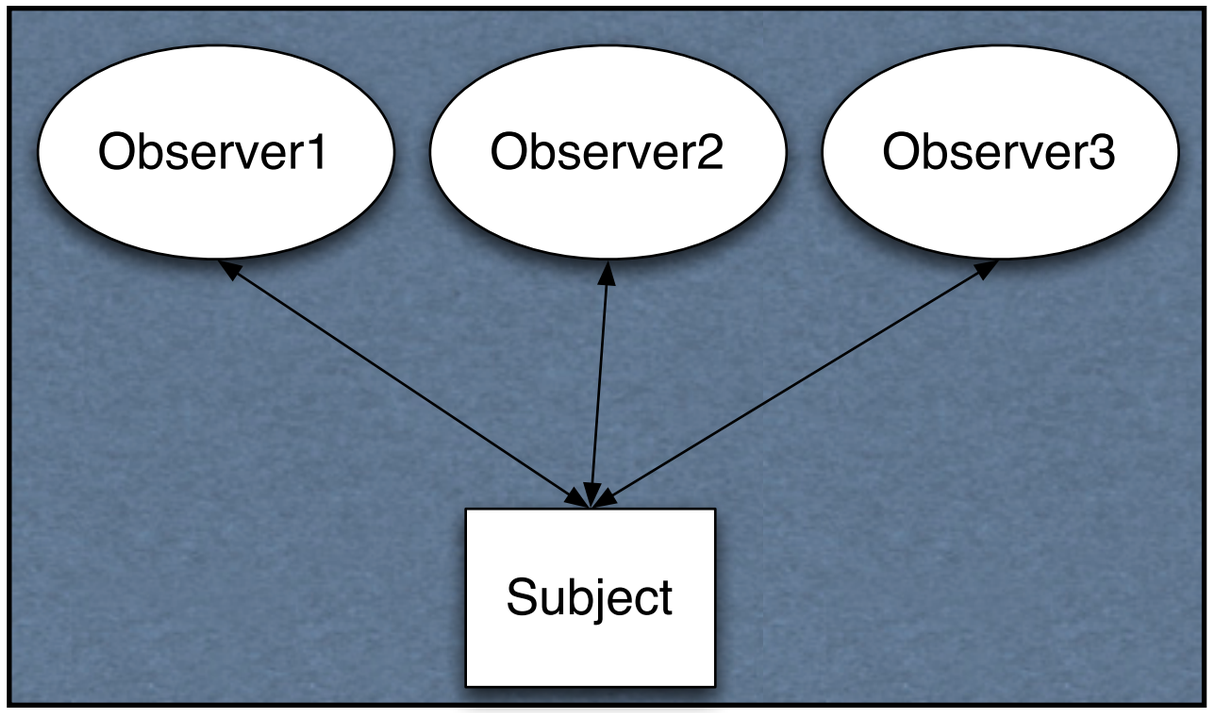
```
ButtonExample>>go

	self count value: ((self count value asNumber) + 1) printString.
	Dialog warn: 'Time to go'.
	^self
```


The method is modified to increase the count each time the "Press Me" button is clicked on. The count is displayed in the window.

Window in Action



Observer



Subject notifies all observers when it changes

Coupling

Measure of the interdependence among modules

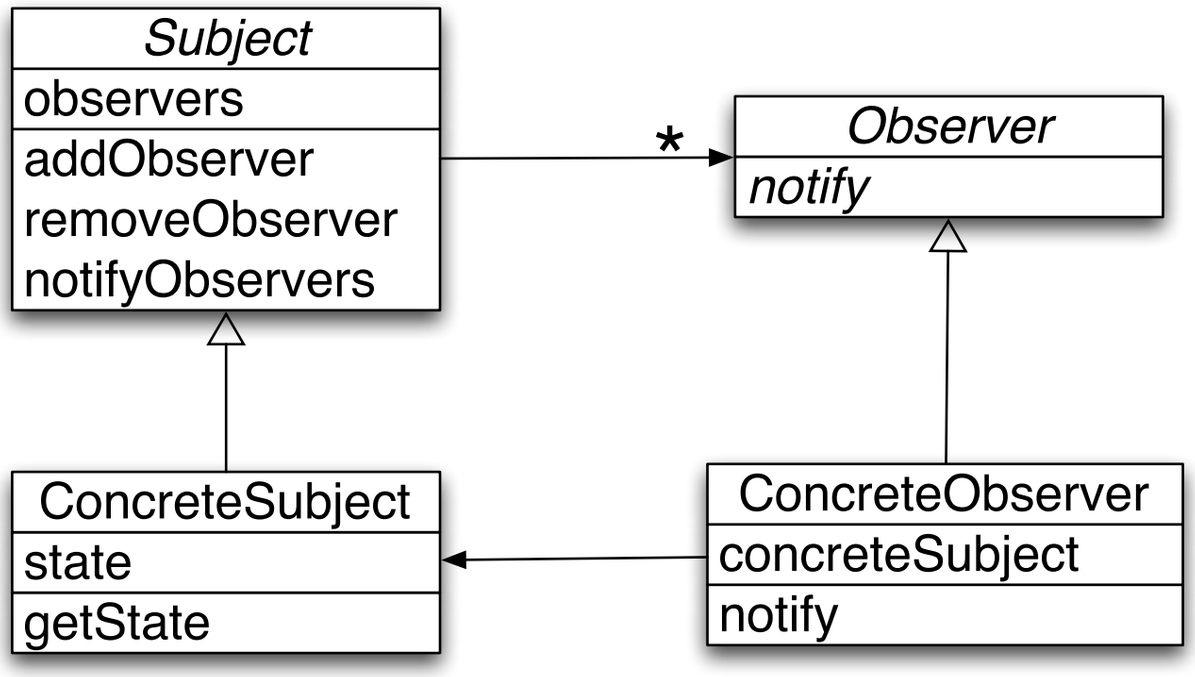
"Unnecessary object coupling needlessly decreases the reusability of the coupled objects "

"Unnecessary object coupling also increases the chances of system corruption when changes are made to one or more of the coupled objects"

Notify With Coupling

```
Subject>>notifyObservers
    observer1 myNewValue: self myValue.
    observer2 displayValue: self myValue.
    observer3 dispalyAndSave: self myValue.
```

Keeping it Flexible




```
Subject>>notifyObservers
    observers do: [:each | each notify]
```

ValueHolder

A subject

When value changes it notifies observers

```
foo asValue
    Returns ValueHolder on foo

valueHolder value: newValue
    Changes the value
    Notifies observers
```

Button Example

```
ButtonExample>>count

	^count isNil
		ifTrue:
			[count := '0' asValue]
		ifFalse:
			[count]
```

```
ButtonExample>>go

	self count value: (self count value asNumber + 1) printString.
	Dialog warn: 'Time to go'.
	^self
```


In count we create count as a value holder on the string '0'. The text field is configured to register itself as an observer on count and expects the value to be a string. In go we change the value of count. When that happens the text field is notified and updates the display.

Coupling and Transcript

```
Smalltalk.CS535 defineClass: #Customer
   superclass: #{Core.Object}
   instanceVariableNames: 'name phone id '
```

```
Customer>>display
   Transcript
      show: 'Customer(';
      print: name;
      show: ', ';
      print: phone;
      show: ', ';
      print: id;
      show: ')'
```

```
foo := Customer new.
...
foo display.
```

Separate display device from Customer

```
Customer>>printOn: aStream
  aStream
    print: 'Customer(';
    print: name;
    print: ', ';
    print: phone;
    print: ', ';
    print: id;
    print: ')'
```

```
foo := Customer new.
...
Transcript
    show: foo printString.

bar := 'bar' asFilename writeStream.
bar
    nextPutAll: foo printString
```


By separating the output device from the class we gain flexibility on where the output goes.

GUIs & Coupling

Domain information
- Customer records
- Inventory
- Names
- Reports
- Addresses

Application/GUI information
- Menus
- Error Messages
- Help information
- Labels

Keep domain and application information separate

Application information changes faster

Often there is multiple view of domain information

Model-View-Controller (MVC)

Model

Encapsulates

Domain information
Core data and functionality

Independent of

Specific output representations
Input behavior

View

Display data to the user

Obtains data from the model

Multiple views of the model are possible

Controller

Handles input

- Mouse movements and clicks
- Keyboard events

Each view has it's own controller

Programmers commonly don't see controllers

Button Example

Model

ButtonExample

View

Created dynamically from window spec

Controller

Hidden

Issues with ButtonExample

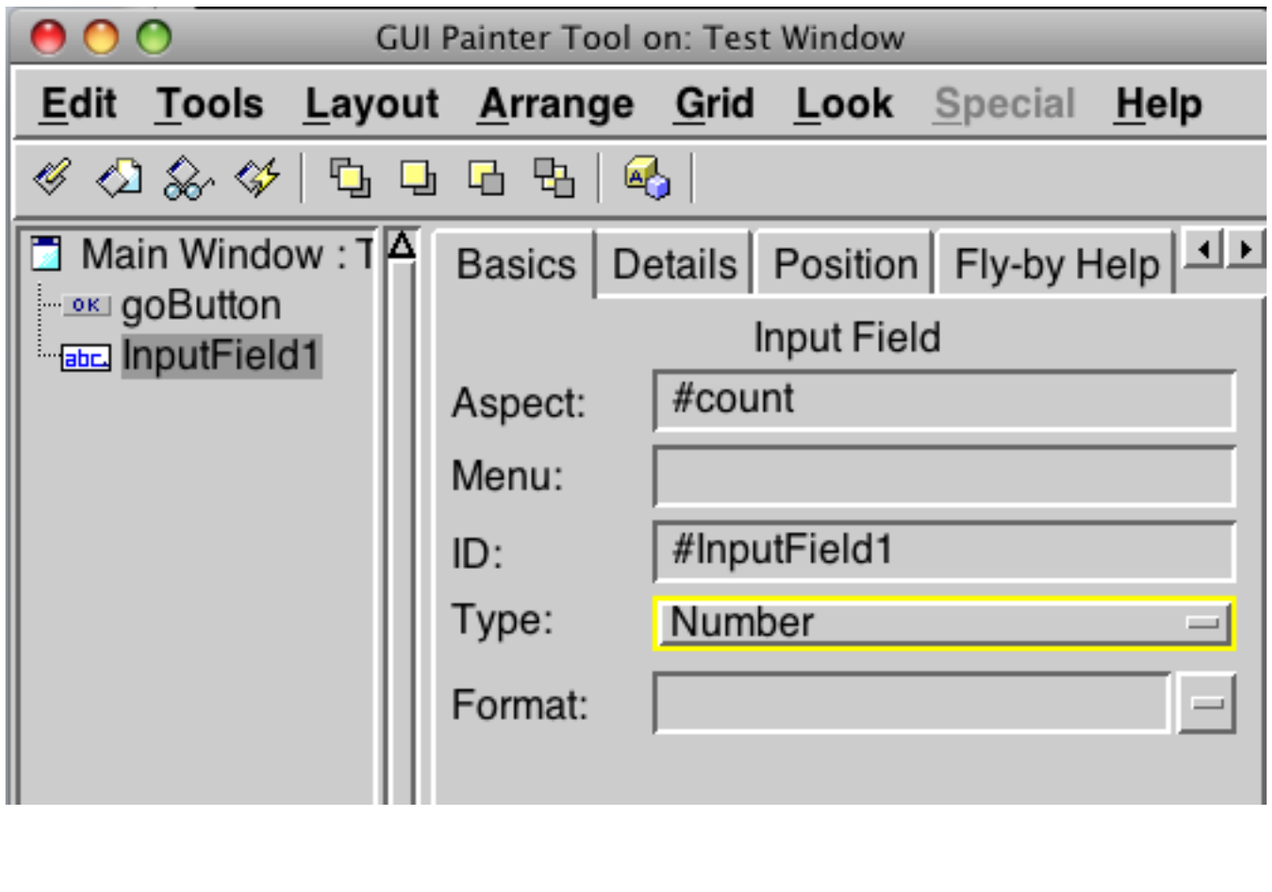
Strings rather than numbers

Dealing with ValueHolders rather than with values

View code/logic in domain code

Well there really is no domain objects as the example is so simple, but the issue needs to be addressed.

Configuring the Widget for Numbers



Using Numbers in Button Example

ButtonExample>>count

```
^count isNil
    ifTrue:
        [count := 0 asValue]
    ifFalse:
        [count]
```

ButtonExample>>go

```
self count value: (self count value  + 1) .
Dialog warn: 'Hi'.
^self
```

Adapters

ButtonExample Adapter

```
ButtonExample>>countAdapter

	| countAdapter |
	countAdapter := AspectAdaptor subject: self.
	countAdapter
		forAspect: #count;
		subjectSendsUpdates: true.
	^countAdapter
```

```
ButtonExample>>initialize

	count := 0
```

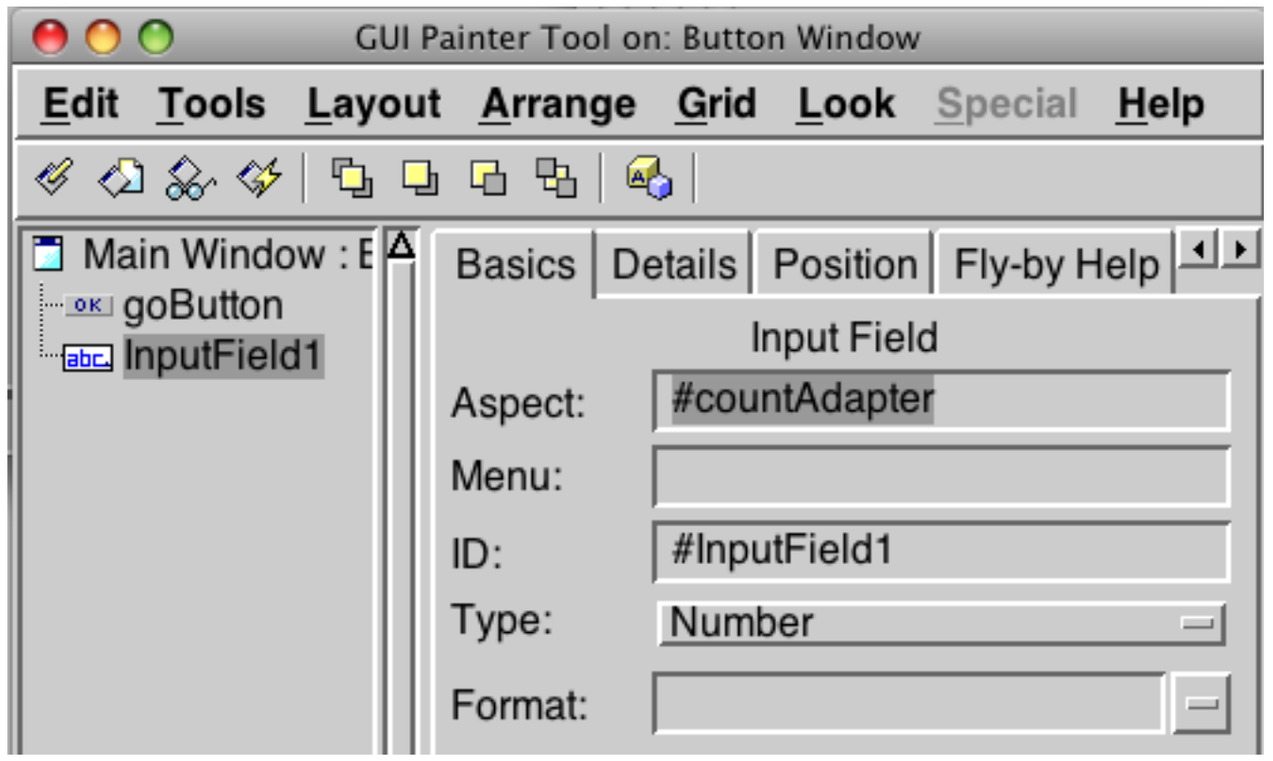
```
ButtonExample>>count

	^count
```

```
ButtonExample>>go

	count := count + 1.
	self changed: #count.
	Dialog warn: 'Time to go'.
	^self
```

```
ButtonExample>>count: anInteger

	count := anInteger
```


The text widget wants a ValueHolder, we want to have an integer. So we use an adapter that looks like a ValueHolder to the text widget. However when it needs to access the value it uses the count and count: methods.

Using a Number rather than a String



View & Domain Logic Mixed

ButtonExample

- Simple example
- Designed to show how to use a widget

It handles both view logic and domain logic

Simple Domain Class

```
Smalltalk defineClass: #Counter
	superclass: #{Core.Object}
	instanceVariableNames: 'count '

Counter class>>new
	^super new initialize

Counter>>count
	^count

Counter>>count: anInteger
	count := anInteger

Counter>>increment
	self count: count + 1

Counter>>initialize
	count := 0
```

Using the Domain Object

```
Smalltalk defineClass: #ButtonExample
        superclass: #{UI.ApplicationModel}
        instanceVariableNames: 'count '
```

```
initialize
        count := Counter new
```

```
countAdapter
        | countAdapter |
        countAdapter := AspectAdaptor subject: count.
        countAdapter
                forAspect: #count;
                subjectSendsUpdates: true.
        ^countAdapter
```

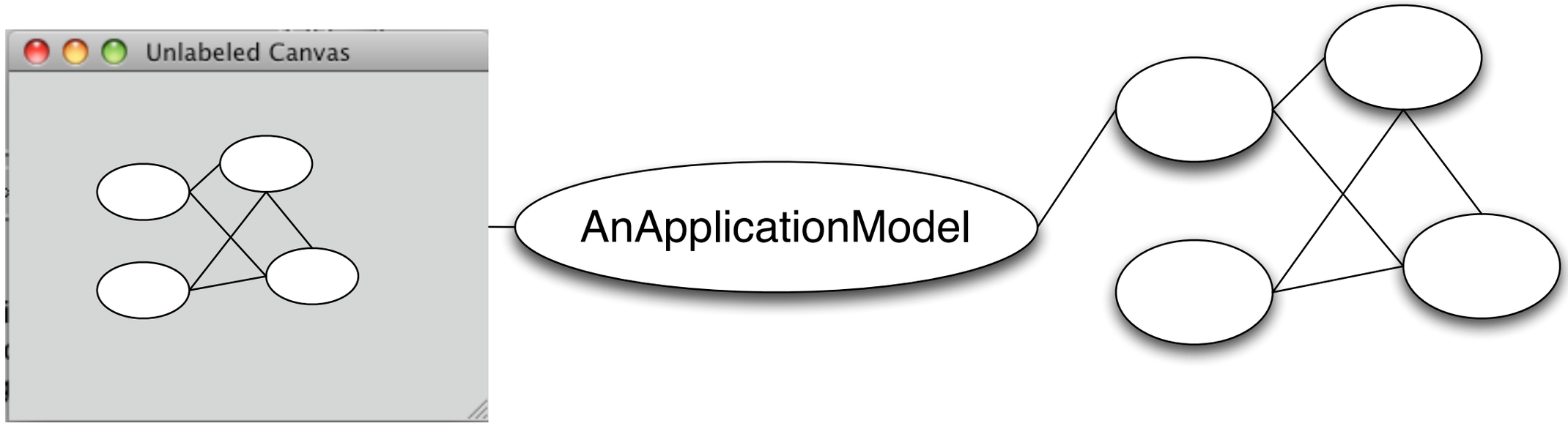
```
go
        count increment.
        count changed: #count.
        Dialog warn: 'Time to go'.
        ^self
```

Issue - Who changes count?

ButtonExample class controls when count changes

ButtonExample can then inform window of changes
Keeps Counter class independent of GUI

Issue - Who changes count?



What if other objects can change count?

ButtonExample will not be able to inform window of changes

When we have multiple domain objects a message to one may result in changes to one or more objects. If these objects contribute to the display it may not be possible for the application model to notify the view how to update itself correctly.

Domain Objects Updated

```
Smalltalk defineClass: #Counter
        superclass: #{Core.Object}
        instanceVariableNames: 'count '

Counter class>>new
        ^super new initialize

Counter>>count
        ^count

Counter>>count: anInteger
        count := anInteger

Counter>>increment
        self count: count + 1.
    self changed: #count

Counter>>initialize
        count := 0
```

ButtonExample Updated

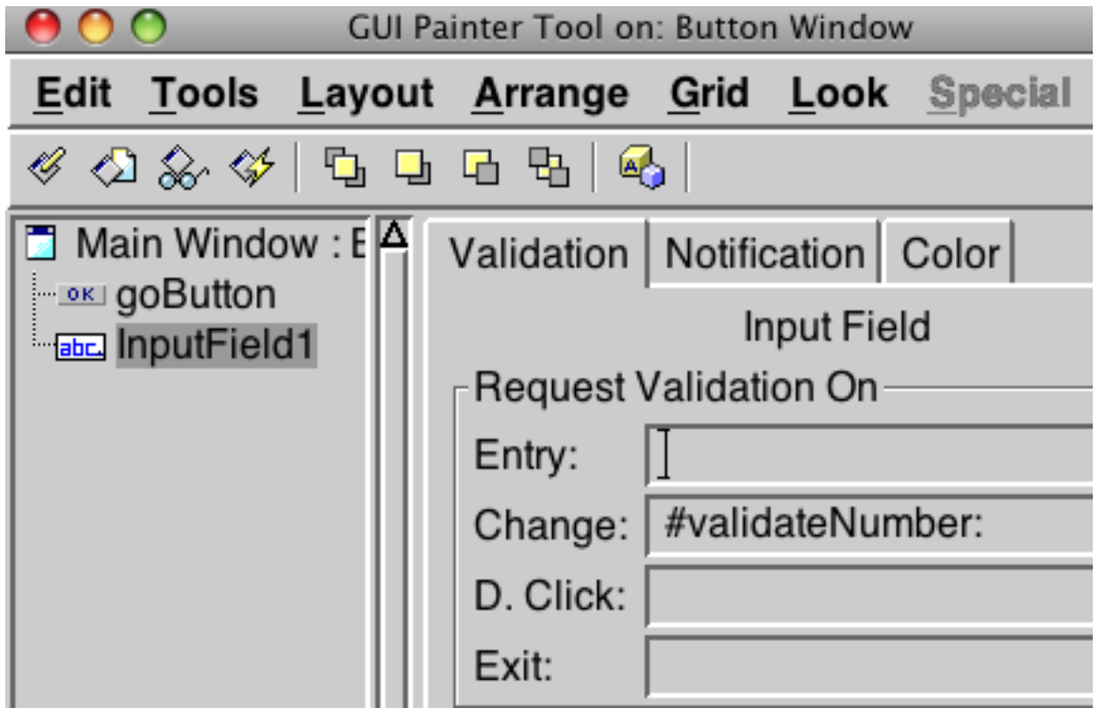
```
Smalltalk defineClass: #ButtonExample
        superclass: #{UI.ApplicationModel}
        instanceVariableNames: 'count '
```

```
initialize
        count := Counter new
```

```
countAdapter
        | countAdapter |
        countAdapter := AspectAdaptor subject: count.
        countAdapter
                forAspect: #count;
                subjectSendsUpdates: true.
        ^countAdapter
```

```
go
        count increment.
        Dialog warn: 'Time to go'.
        ^self
```

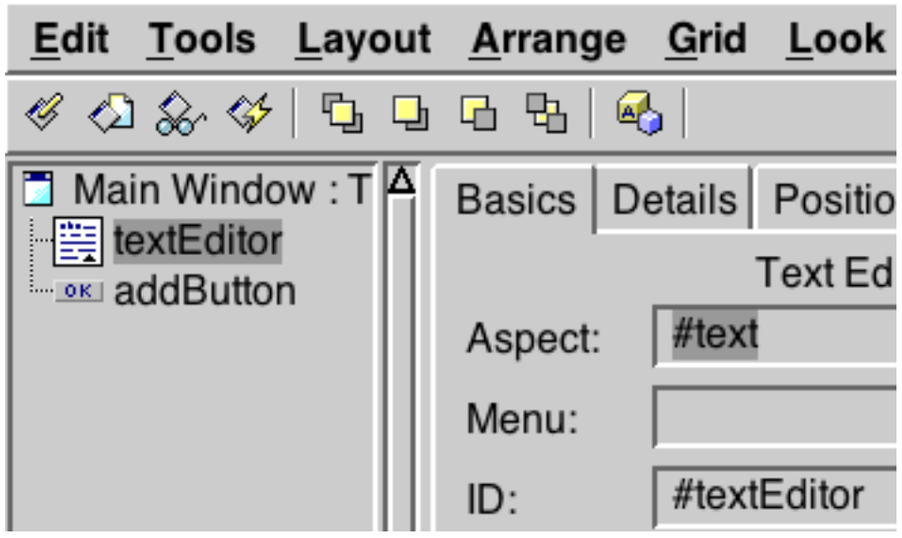
Validating Input



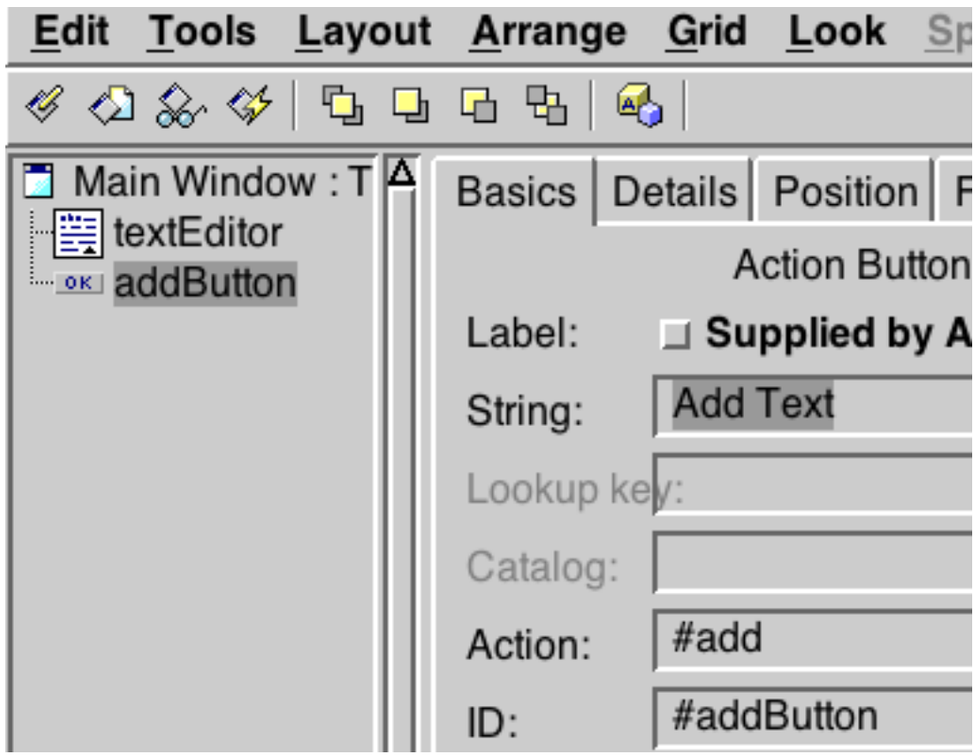

```
ButtonExample>>validateNumber: aController
        | entry |
        entry := aController editValue.
        ^entry >= 0
```

Text Editor Example

Text Editor Example




```
TextExample>>text
        ^text isNil
                ifTrue:
                        [text := 'Hi Mom' asValue]
                ifFalse:
                        [text]
```




```
TextExample>>add
        self text
      value: self text value , '\Add more text\' withCRs
```