Tutorial

ApplFLab

Application Framework Laboratory

Custom-made User Interface Components in VisualWorks

Koen De Hondt

Programming Technology Lab
Computer Science Department
Vrije Universiteit Brussel

email: kdehondt@vnet3.vub.ac.be
http://progwww.vub.ac.be/prog/pools/applflab/applflab.html

Tutorial

Overview

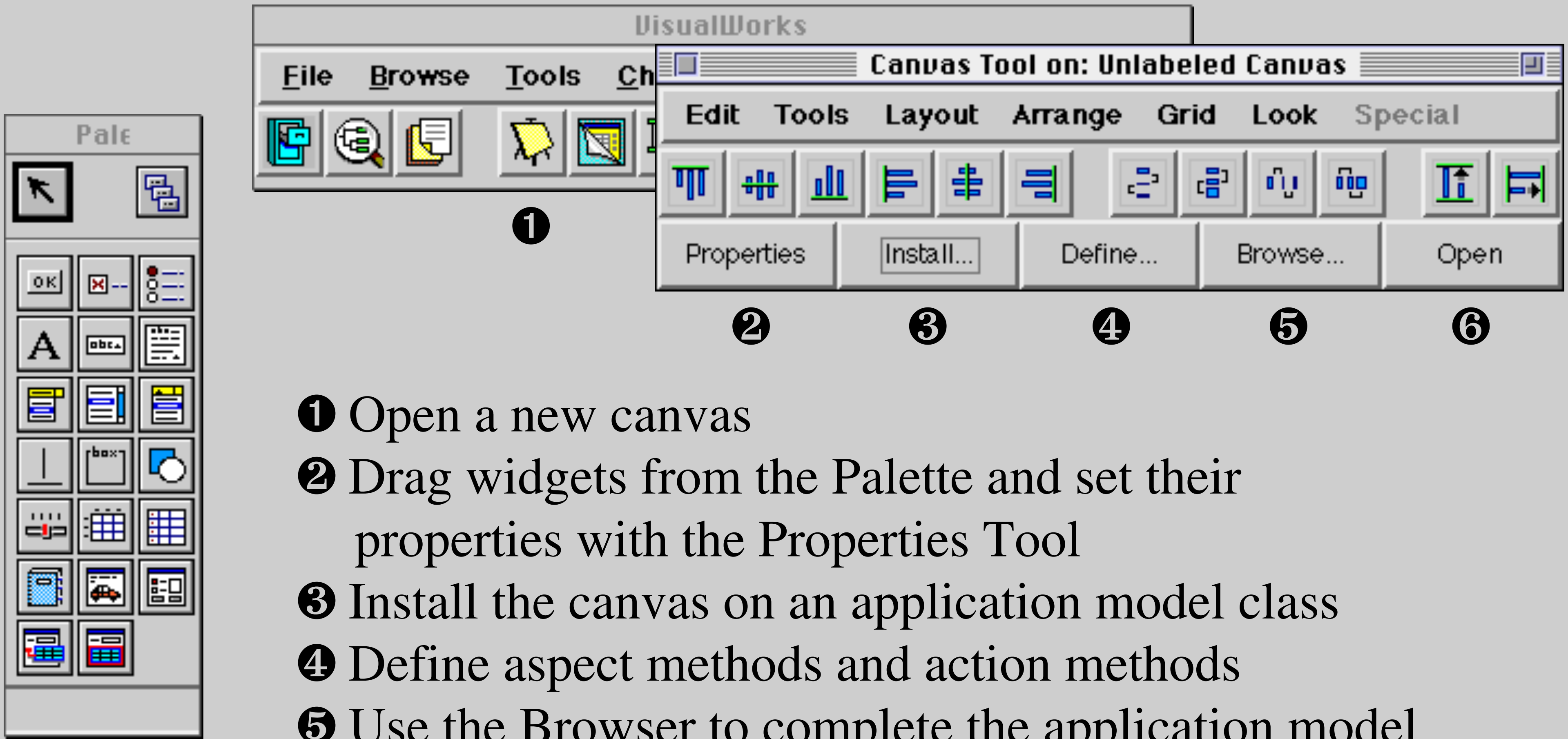
- Application building in VisualWorks
- Components
- Building new components
- ApplFLab
- Example: building a file selector component with ApplFLab
- Exercise: building components with ApplFLab

- **Application building in VisualWorks**
- Components
- Building new components
- ApplFLab
- Example: building a file selector component with ApplFLab

- Exercise: building components with ApplFLab

Tutorial | **Application Building in VisualWorks**

Application Building in VisualWorks



❶ Open a new canvas
❷ Drag widgets from the Palette and set their properties with the Properties Tool
❸ Install the canvas on an application model class
❹ Define aspect methods and action methods
❺ Use the Browser to complete the application model
❻ Open the application for testing

Important Classes

- **UIPainter**
  The application for composing a canvas.
- **UISpecification**
  Abstract superclass of all specification classes. The objects describe a user interface entity: a widget, a window, …
- **ApplicationModel**
  Abstract superclass of all application classes. On these classes window specifications are installed in literal form. These classes implement all the user interface logic and the link with the domain model.
- **UIDefiner**
  Object that is responsible for generation of aspect and action methods.
- **UILookPolicy**
  Object that knows how to setup widgets in a particular host look.
- **UIBuilder**
  Object responsible for constructing a user interface from a UISpecification and a UILookPolicy.

The Standard Palette

- The components on the standard palette are general-purpose, <u>thus</u> low-level
- An application designer needs high-level components targeted to a particular problem domain
- This can be accomplished by:
  - Building composites of standard components, i.e. applications
  - Implementing custom view classes

so-called “domain-specific components”

Reuse of Applications

- Reuse through specialisation (subclassing)
- Reuse through composition (subcanvas technology)
  - Use of application models on an "as-is" basis, no configuration possible at painting time
  - Handcoding necessary to configure the embedded application model at runtime

Reuse of Custom-Made Views

- Custom-made views can be reused in an application by means of a view holder widget
  - Use of views on an “as-is” basis, no configuration possible at painting time
  - Handcoding necessary to configure the embedded view at runtime

Problems with this Kind of Reuse

- Configuration of an embedded application model or custom-made view at runtime leads to code duplication
- Handcoding the configuration leads to coding errors when the implementation of the application model or custom-made view is not well-understood

Solution: Make Your Own Components

- When you want to put an application model or a custom-made view on the palette
- When you want to configure your components at painting time, instead of at runtime
- When you want to make applications for your problem domain in a plug & play fashion

- Application building in VisualWorks
- **Components**
- Building new components
- ApplFLab
- Example: building a file selector component with ApplFLab
- Exercise: building components with ApplFLab

What is a Component?

- At building time a component is represented by an instance of a subclass of ComponentSpec
- At runtime a component is represented by an instance of a subclass of VisualPart, usually a View

- A ComponentSpec is stored as a literal array encoding in a spec method

Component Specifications

- Specifications tell a UIBuilder what features and appearance are wanted in the interface that it is building
- Subclasses of ComponentSpec convey all of the information required in order to build and emplace an instance of a particular kind of view component, including its location, opacity, color, decoration, name, and other more specialized properties

Component Specification Classes

- Supply a name and an icon for identification in the application builder
- Supply the user interfaces to edit their instances, the so-called “properties tool slices”
- Define what default instances look like when dragged from a palette
- Define how target components are built

ComponentSpec Hierarchy

- ComponentSpec (layout)
  - NamedSpec (name, flags, isOpaque, colors)
    - ArbitraryComponentSpec (component)
    - CompositeSpec ()
    - SubCanvasSpec (majorKey, minorKey, clientKey)
    - WidgetSpec (model, callbacksSpec, tabable)
      - ButtonSpec (label, hasCharacterOrientedLabel, style)
      - MenuComponentSpec (menu, performer)

Important Spec Class Protocol

- Instance methods
  - #defaultModel
  - #dispatchTo:with:
  - #literalArrayEncoding
  - accessor and mutator for each property

Property values are held in instance variables

- Class methods
  - #componentName
  - #addBindingsTo:for:channel:
  - #specGenerationBlock
  - #placementExtentBlock
  - #paletteIcon
  - #paletteMonoIcon
  - #slices
  - slice resource methods

Visual Parts

- A VisualPart is an object that can create a visual representation of itself (via #displayOn:)
- VisualParts make up a structured picture by providing a pointer to a container, usually a Wrapper

VisualPart Hierarchy

- VisualPart (container)
  - CompositePart (components, preferredBounds)
    - DependentComposite (model)
      - CompositeView (controller)
    - SubCanvas ()
  - DependentPart (model)
    - View (controller)
      - SimpleView (state)
  - SimpleComponent (state)
  - Wrapper (component)

Active components

Passive components

Tutorial | **Overview**

- Application building in VisualWorks
- Components
- **Building new components**
- ApplFLab
- Example: building a file selector component with ApplFLab
- Exercise: building components with ApplFLab

Building a New Component

- Build a new widget class and any necessary supporting classes
- Build a corresponding component spec class
- Add the necessary behavior to UILookPolicy for constructing the component from specifications
- Edit UIPalette to make the component available during painting

Build the Widget

- Implement an MVC triad
- Passive components
  - ➡ V is a subclass of SimpleComponent
- Active components
  - ➡ V is a subclass of SimpleView
- C is a subclass of an appropriate controller class. No input ➡ use class NoController
- M subclass of ValueModel or related

Build Component Spec Class

- Choose a superclass
- Add component description protocol
- Define the default model
- Add the component construction behavior
- Build the properties tool slices
- Add the interface bindings
- Create the spec generation block
- Create the palette icons
- Add other spec class protocol

Build Component Spec Class

- Example code taken from InputFieldSpec

Choose a Superclass

- Passive component
  - ➡ subclass NamedSpec
- Active component
  - ➡ subclass WidgetSpec, MenuComponentSpec, or ButtonSpec
- Specialisation of existing component
  - ➡ subclass that component’s spec class
- Naming convention: all spec class names end with “Spec”, e.g “MyComponentSpec”

Choose a Superclass

- An input field is an active component with a menu

```
WidgetSpec
   MenuComponentSpec
        TextEditorSpec
                InputFieldSpec
```

Add Component Description Protocol

- Add an instance variable for each property of the component

  Good idea to include a default value through lazy initialization

- Add an accessor and a mutator for each instance variable

Add Component Description Protocol

```
TextEditorSpec subclass: #InputFieldSpec
   instanceVariableNames: 'numChars type formatString '


numChars
   ^numChars

formatString
   ^formatString

type
   ^type == nil ifTrue: [#string] ifFalse: [type]
```

Define the Default Model

- Implement the #defaultModel method, answer a ValueModel or related object
- Serves as the model at editing time
- Serves as the model at runtime when no model is specified
- Used by UIDefiner to generate the aspect method

Define the Default Model

```
defaultModel

  type == nil ifTrue: [^ValueHolder with: String new].
  type == #number
      ifTrue: [^ValueHolder with: 0].
  (type == #string or: [type == #password])
      ifTrue: [^ValueHolder with: String new].
  type == #text
      ifTrue: [^ValueHolder with: Text new].
  ^ValueHolder with: nil
```

Add Construction Behavior

- Implement the #dispatchTo:with: method

```
dispatchTo: policy with: builder

  policy myComponent: self into: builder
```

Use an appropriate name for your component

Add Construction Behavior

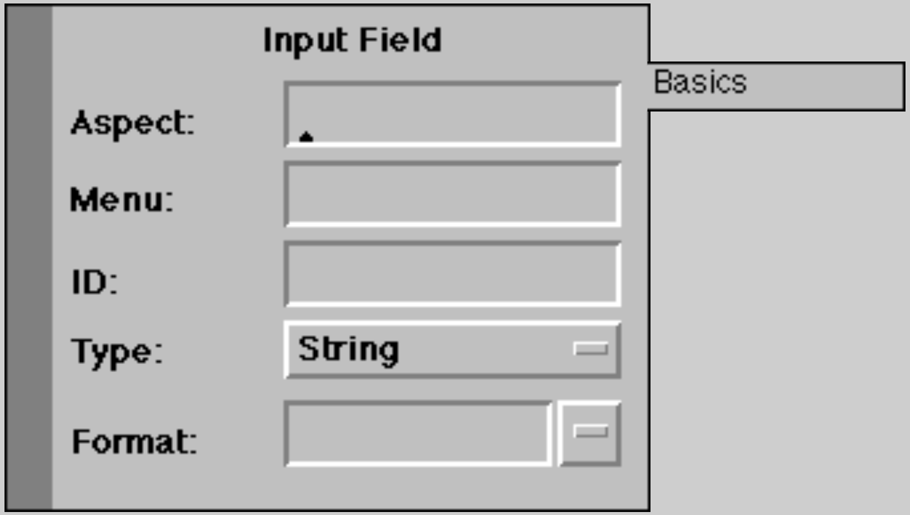
```
dispatchTo: policy with: builder

  policy inputBox: self into: builder
```

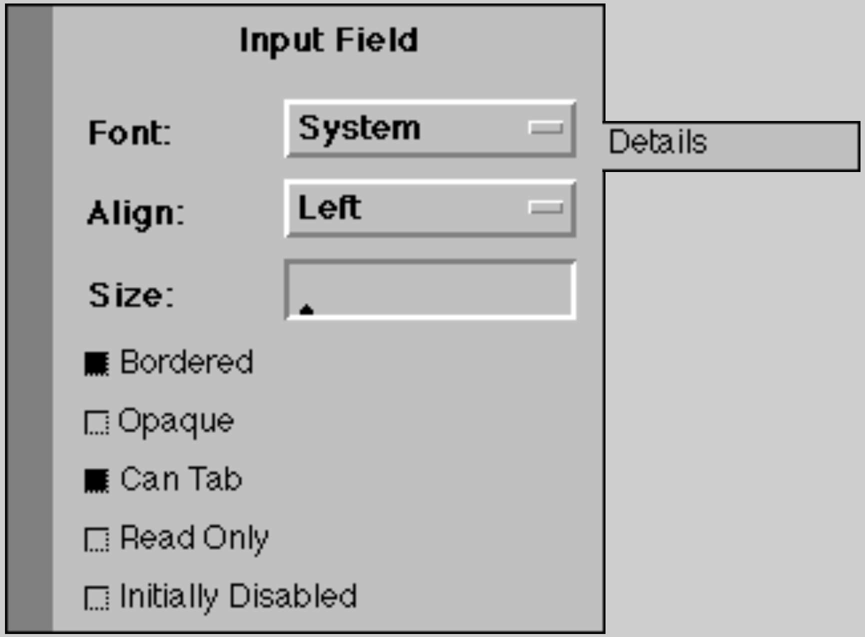
Build the Properties Tool Slices

- Paint each slice with the Painter
- Install them as class methods in protocol *interface specs*
- Naming convention:
  slice named “slice” installed in method
  named “slice<u>EditSpec</u>”
  e.g. “basicsEditSpec”, “detailsEditSpec”

Build the Properties Tool Slices

- InputFieldSpec implements #basicsEditSpec and #detailsEditSpec





Add the Interface Bindings

- Implement the #addBindingsTo:for:channel: class method in protocol *private-interface building*
- Add an entry for each property as follows:

```
addBindingsTo: env for: inst channel: aChannel

   super addBindingsTo: env for: inst channel: aChannel.
   env  at: #property
        put: (self  adapt: inst
                    forAspect: #property
                    channel: aChannel)
```

Add the Interface Bindings

```
addBindingsTo: env for: inst channel: aChannel

super addBindingsTo: env for: inst channel: aChannel.
env at:#numChars
  put: (TypeConverter onNumberValue: (self adapt:inst
           forAspect:#numChars channel:aChannel)).
env at:#type
  put: (self adapt:inst forAspect:#type channel: aChannel).
env at:#formatString
  put: (self adapt:inst forAspect:#formatString
           channel: aChannel).
env at:#typeMenu put:self typeMenu.
... <and more> ...
```

Create the Spec Generation Block

- Implement the #specGenerationBlock class method in protocol *private-interface building*
- Answer a block, taking 2 arguments, a controller and a point, returning a default instance of the spec class

default size
when painted

```
specGenerationBlock
  ^[:ctrl :point| MyComponentSpec
      layout:((ctrl gridPoint: point) extent:100@50)]
```

Create the Spec Generation Block

- Inherited from ComponentSpec

```
specGenerationBlock

^[:ctrlr :point | self layout:
  ((ctrlr gridPoint: point) extent:
   (ctrlr currentMode value class
      placementExtentFor: self inBuilder: ctrlr builder))]
```

Create the Palette Icons

- Use the Image Editor to paint a black & white icon and a color icon with extent 26@26
- Include the look of the button: border, background and 3D effect
- Install the icons in protocol *resources*, in methods paletteMonoIcon and paletteIcon respectively

Create the Palette Icons

Add Other Spec Class Protocol

- In protocol *private-interface building*:
  - #componentName
    returns a String identifying the component
  - #placementExtentBlock
    returns a block, returning a Point describing the component's default extent
  - #slices
    returns an array of arrays describing the slices in the component's properties tool

Add Other Spec Class Protocol

```
componentName
   ^'Input Field'

placementExtentBlock
   ^[:bldr | 100 @ (TextAttributes defaultLineGrid + 4)]

slices                              (inherited from WidgetSpec)
   ^#( (Basics basicsEditSpec)
       (Details detailsEditSpec)
       (Validation validationEditSpec nil callbacks)
       (Notification notificationEditSpec nil callbacks)
       (Color propSpec ColorToolModel)
       (Position propSpec PositionToolModel)
       (#'Drop Target' dropTargetSpec) )
```

Change UILookPolicy

- Implement #myComponent:into: in class UILookPolicy
- This method interprets the spec argument, sets up a VisualPart accordingly and puts it in the builder argument

```
myComponent: spec into: builder

   ...
```

Change UILookPolicy

```
inputBox: spec into: builder
   | component model menu performer alignment |
  model := spec modelInBuilder:builder.
  component := self inputBoxClass new.
...
  component model:model.
...
  component controller:self inputBoxControllerClass new.
  (menu := spec getMenuIn:builder) == nil
      ifFalse: [component controller menuHolder:menu].
...
  spec numChars == nil ifFalse: [component controller
      maxChars:spec numChars]
...
  builder wrapWith:
      (self simpleWidgetWrapperOn:builder spec: spec)
```

Edit UIPalette

- Edit method #standardSpecsForPalette
- This method returns the collection of component specification classes on the palette

```
standardSpecsForPalette

  ^#(#ActionButtonSpec #CheckBoxSpec  #RadioButtonSpec
      #LabelSpec #InputFieldSpec #TextEditorSpec
      #MenuButtonSpec #SequenceViewSpec #ComboBoxSpec
      #DividerSpec #GroupBoxSpec #RegionSpec  #SliderSpec
      #TableViewSpec  #DataSetSpec #NoteBookSpec
      #ArbitraryComponentSpec #SubCanvasSpec)
```

Edit UIPalette

```
standardSpecsForPalette

   ^#(#ActionButtonSpec #CheckBoxSpec  #RadioButtonSpec
       #LabelSpec #InputFieldSpec #TextEditorSpec
       #MenuButtonSpec #SequenceViewSpec #ComboBoxSpec
       #DividerSpec #GroupBoxSpec #RegionSpec  #SliderSpec
       #TableViewSpec  #DataSetSpec #NoteBookSpec
       #ArbitraryComponentSpec #SubCanvasSpec)
```

Tutorial | Overview

- Application building in VisualWorks
- Components
- Building new components
- **ApplFLab**
- Example: building a file selector component with ApplFLab
- Exercise: building components with ApplFLab

ApplFLab

- First conceived to support the creation of custom-made <u>composite</u> components for domain-specific frameworks
  - Turns an application model into a component on the palette
  - Allows associating properties with application models
  - Uses subcanvas technology

ApplFlab

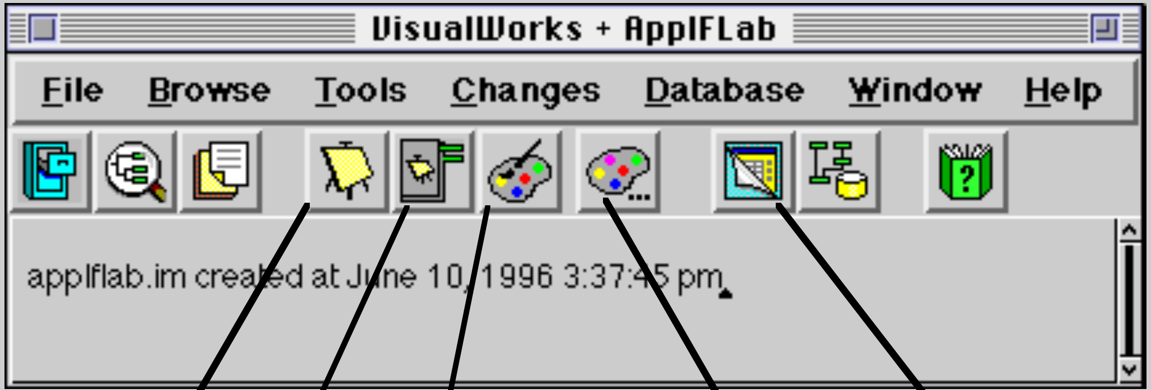
- Is now able to make subclasses of standard specification classes, as well as spec classes for custom-made VisualPart classes

- This tutorial only focusses on <u>composite</u> components

ApplFLab Implementation

- ApplFLab is an extension of VisualWorks, ~~it does not change the standard classes~~
- ApplFLab provides its own version of each VisualWorks tool
- ApplFLab's implementation is subject to change with every upgrade of VisualWorks
- ApplFLab is not a finished product, but is continuously under development

ApplFLab Features



Enhanced painter

Component editor

Palette editor

Multiple palettes

Extended resource finder

+ Enhanced properties tool
+ Extended ApplicationModel

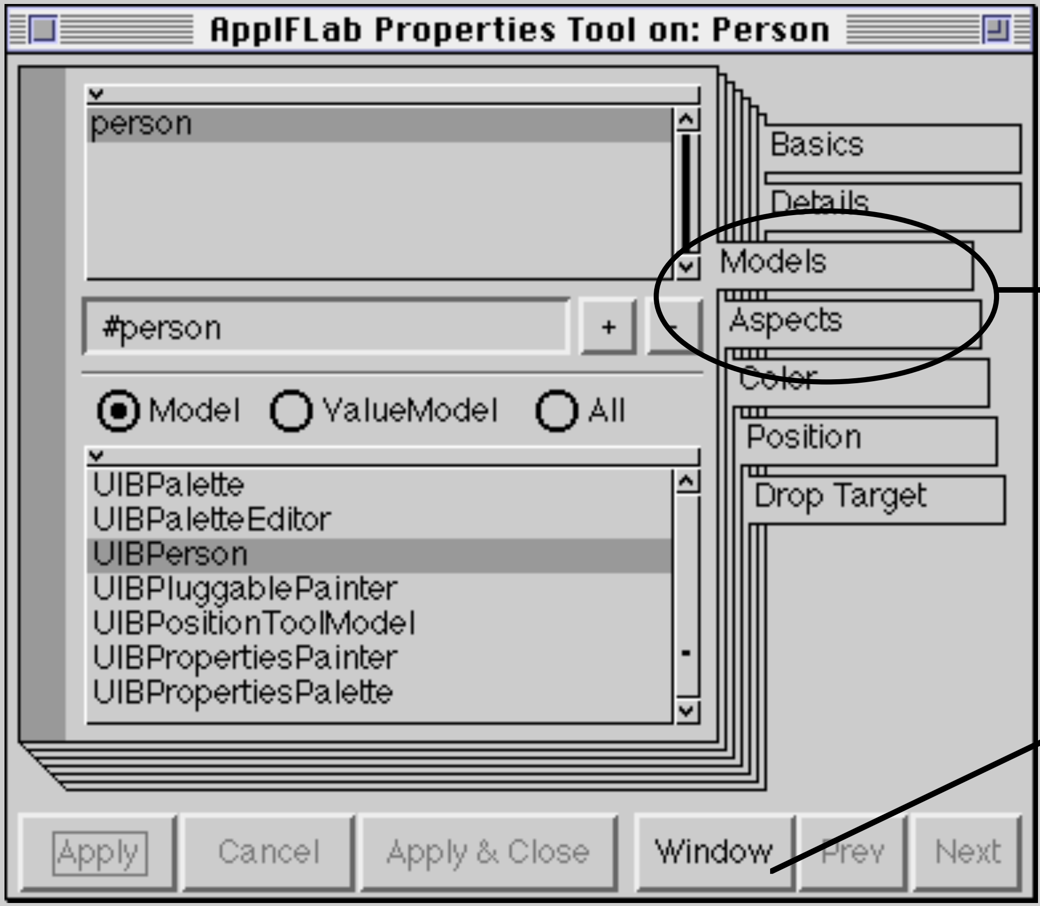
Enhanced Painter

- Pick a palette when opening a new canvas
- Extended widget handles for resizing in one direction
- “Define All...” command to define all aspects

Extended ApplicationModel

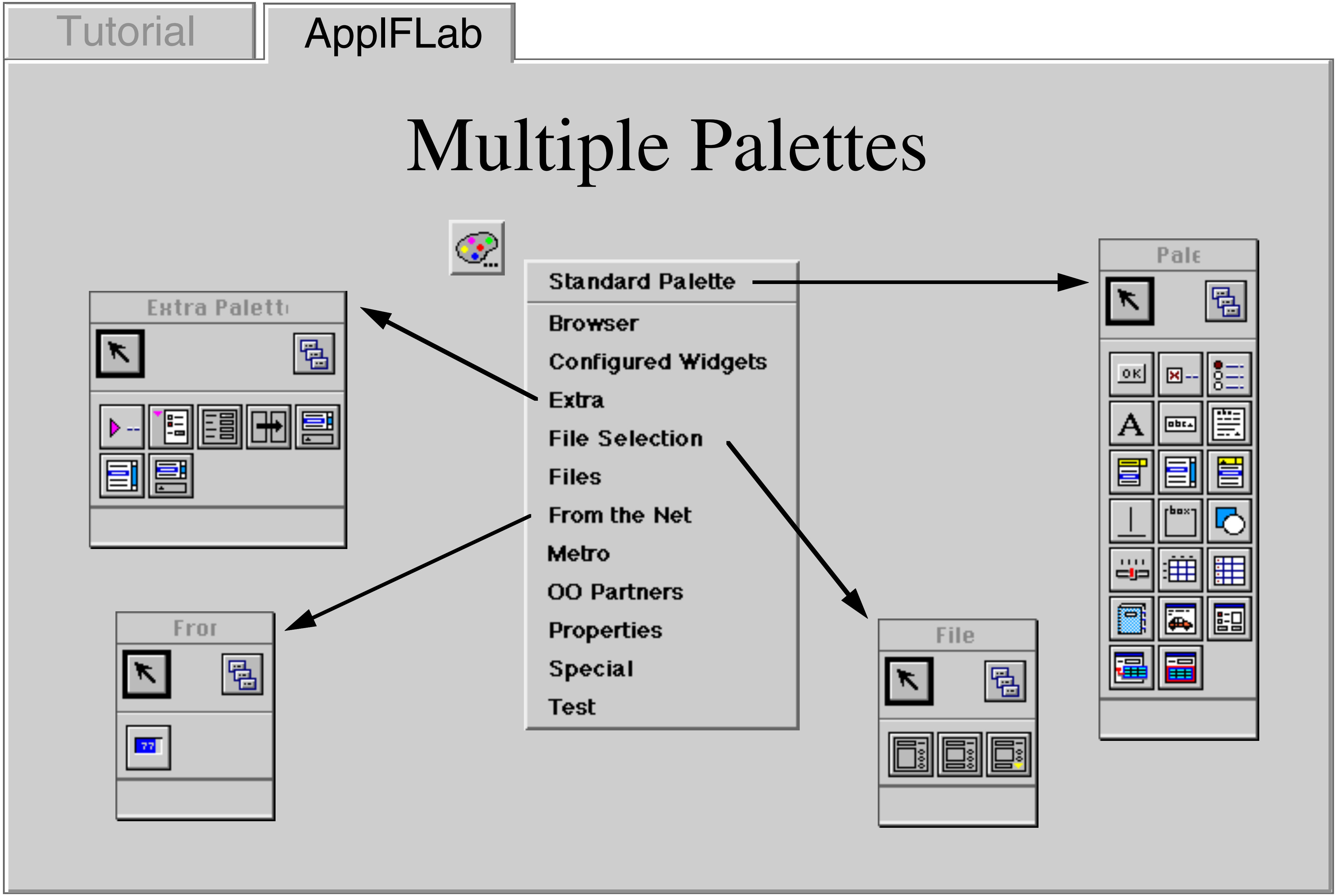
- Class UIBApplicationModel
- Other builder: UIBBuilder
- Convenience methods for:
  - component and widget access
  - component enabling and disabling
  - component visibility control
- Binding of non-unary action methods
- Release of dependencies when window closes

Enhanced Properties Tool

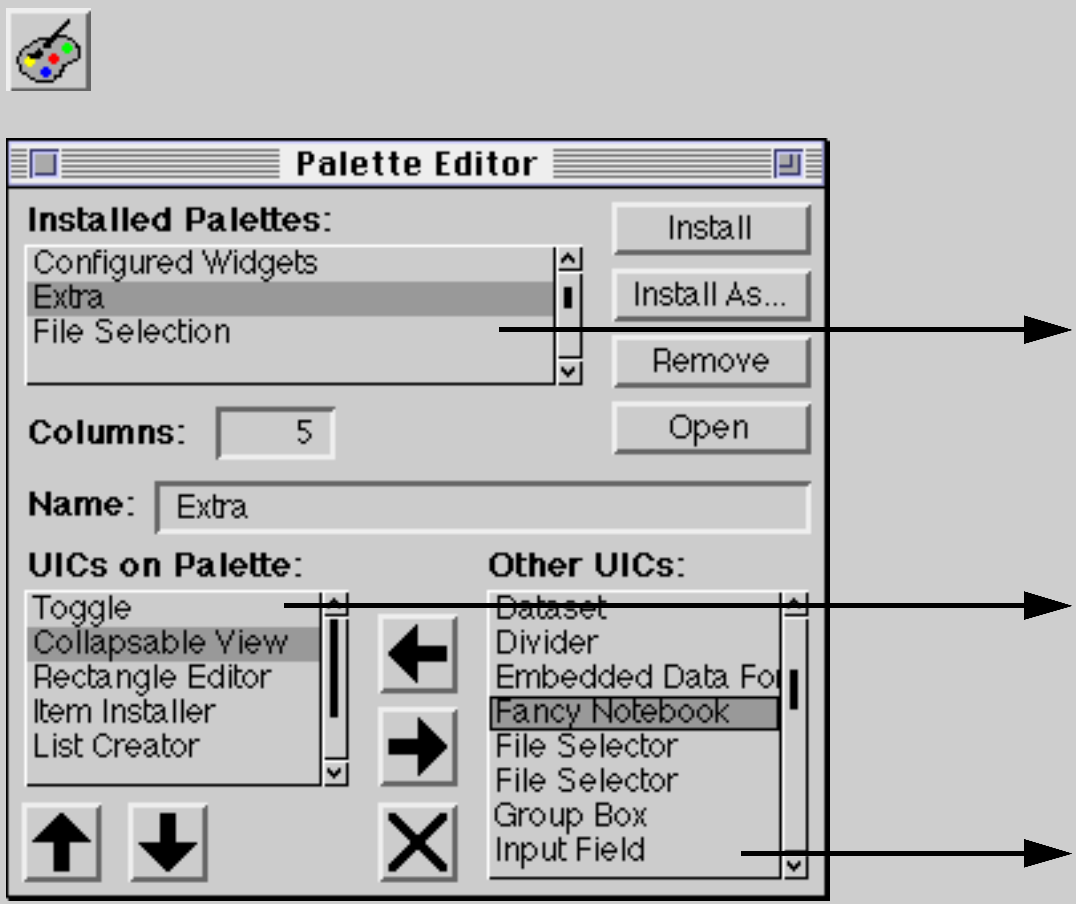


Models and Aspects slices to configure the aspects

Window button to display the properties of the canvas


Tutorial
ApplFLab
Multiple Palettes
Standard Palette
Browser
Configured Widgets
Extra
File Selection
Files
From the Net
Metro
OO Partners
Properties
Special
Test
Extra Palett
Fror
File
Pale
OK
A
abc
box

Palette Editor

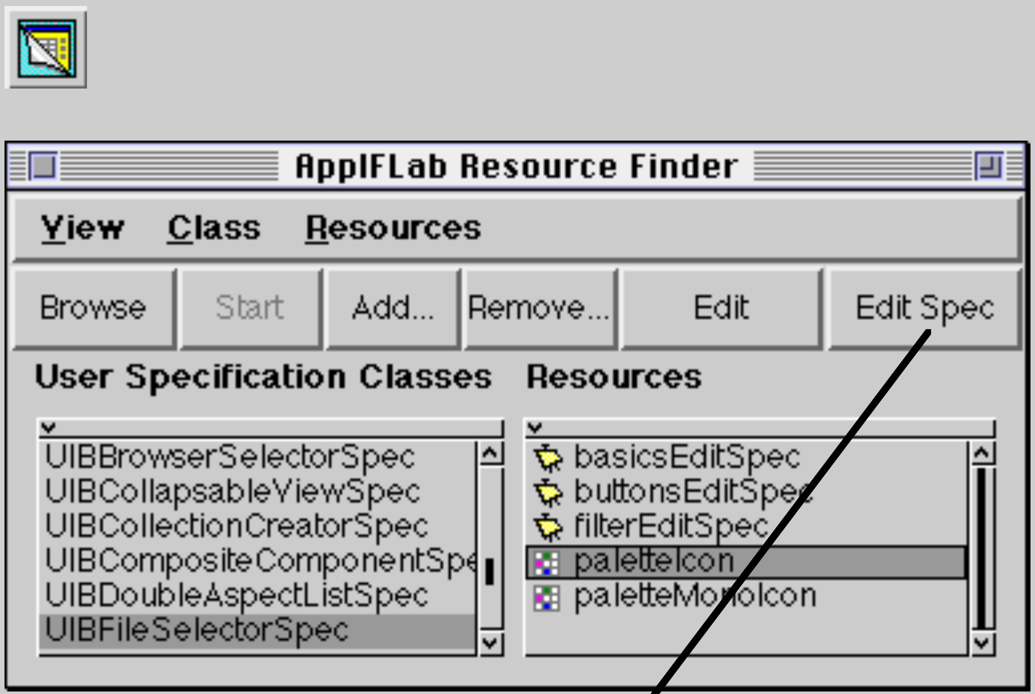


All palettes in the image

Components on the palette after installation

All other components in the image

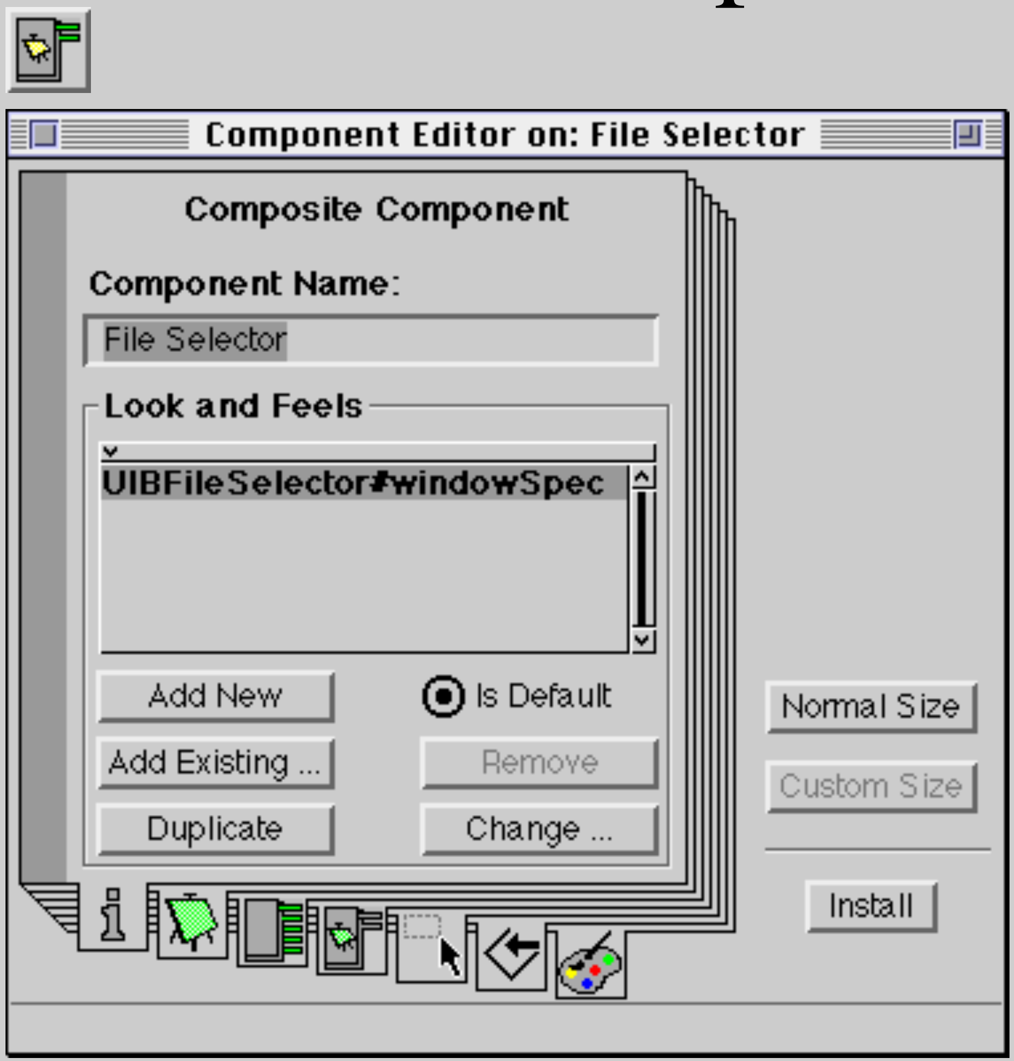
Extended Resource Editor



Open a component editor on the selected specification class

View the (user) specification classes

Component Editor



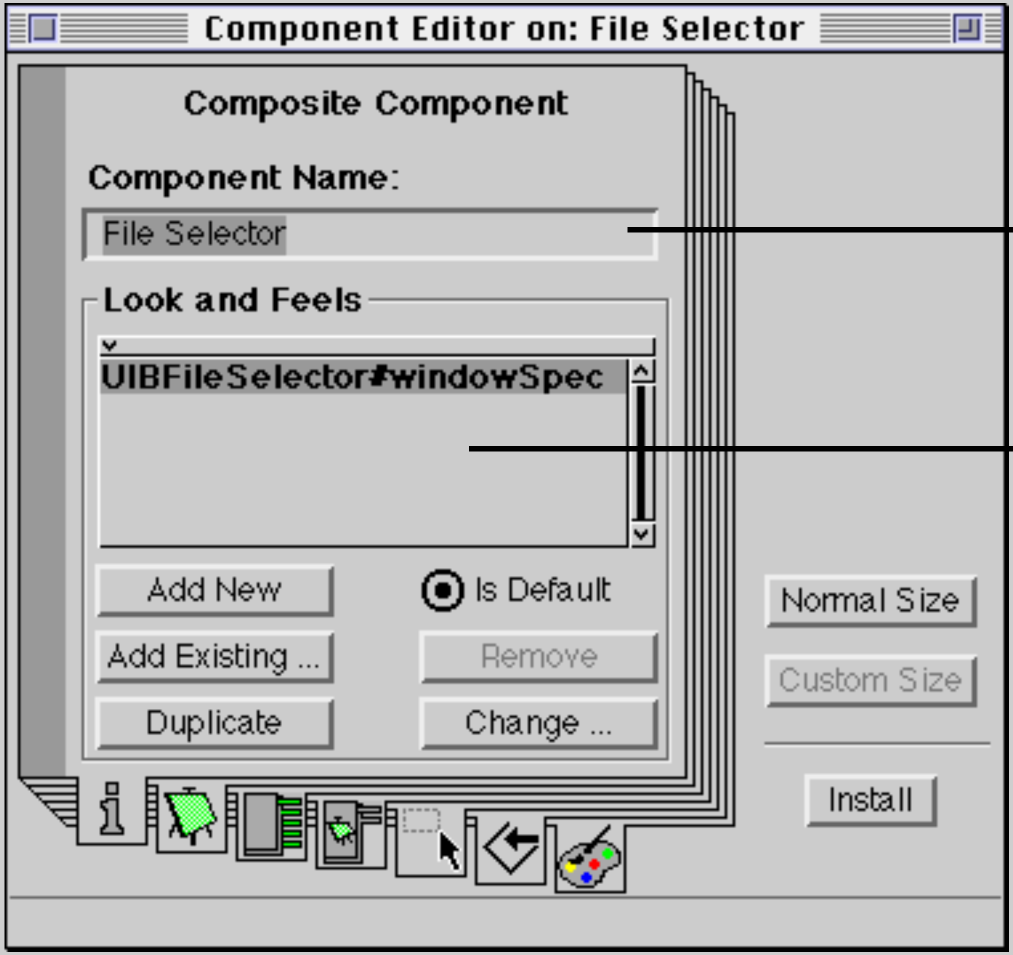
- Central tool of ApplFLab
- Supports creation of new user interface components
  - Composite components
  - Specialisations of standard components
- Can be used to edit standard components (not recommended)

Component Editor—Functionality

❶ ❷ ❸ ❹ ❺ ❻ ❼

❶ Registering the component's application models
❷ Building the application models
❸ Defining the component's properties tool slices
❹ Painting the properties tool slices
❺ Supplying painting information
❻ Installing the component in the system
❼ Putting the component on a palette

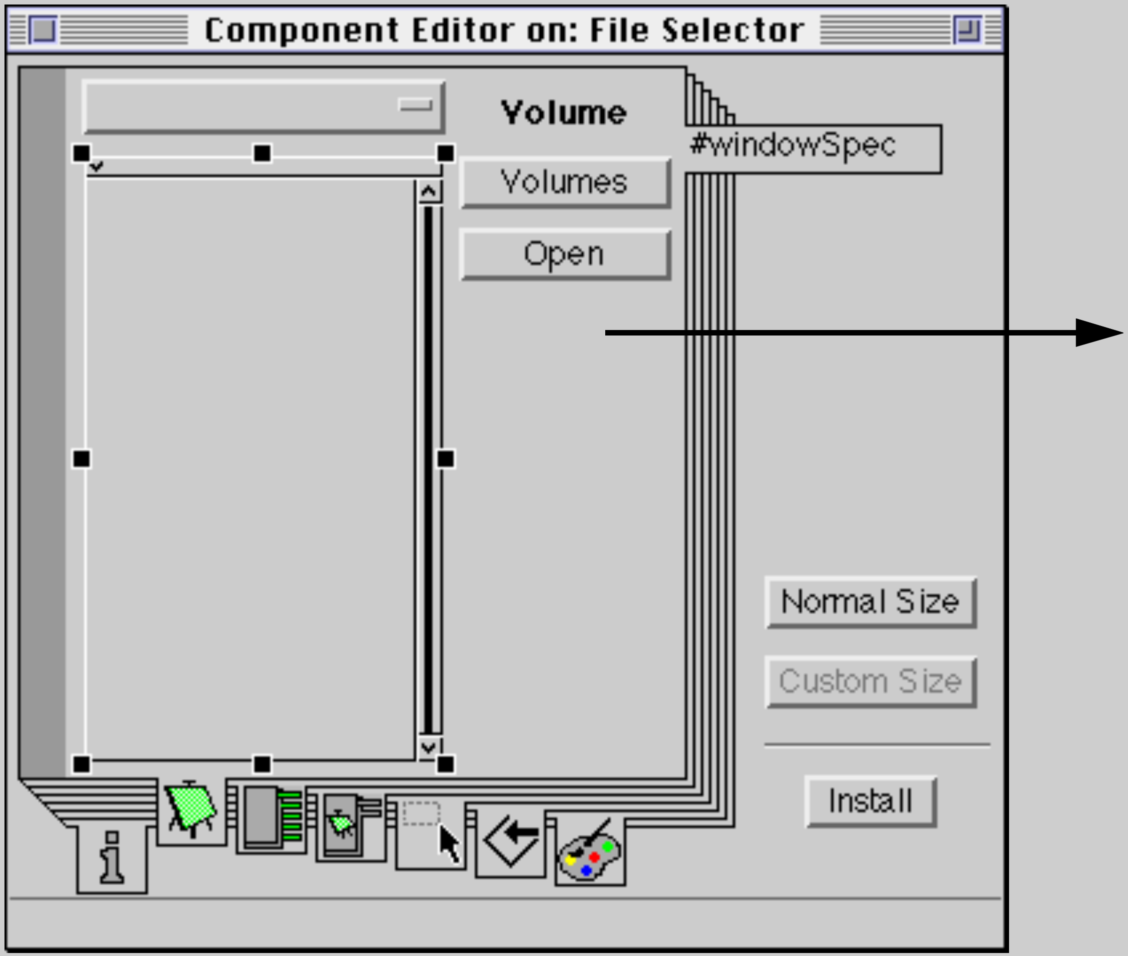
❶ Registering the Application Models



Name of component

List of all application model – user interface pairs that implement this component

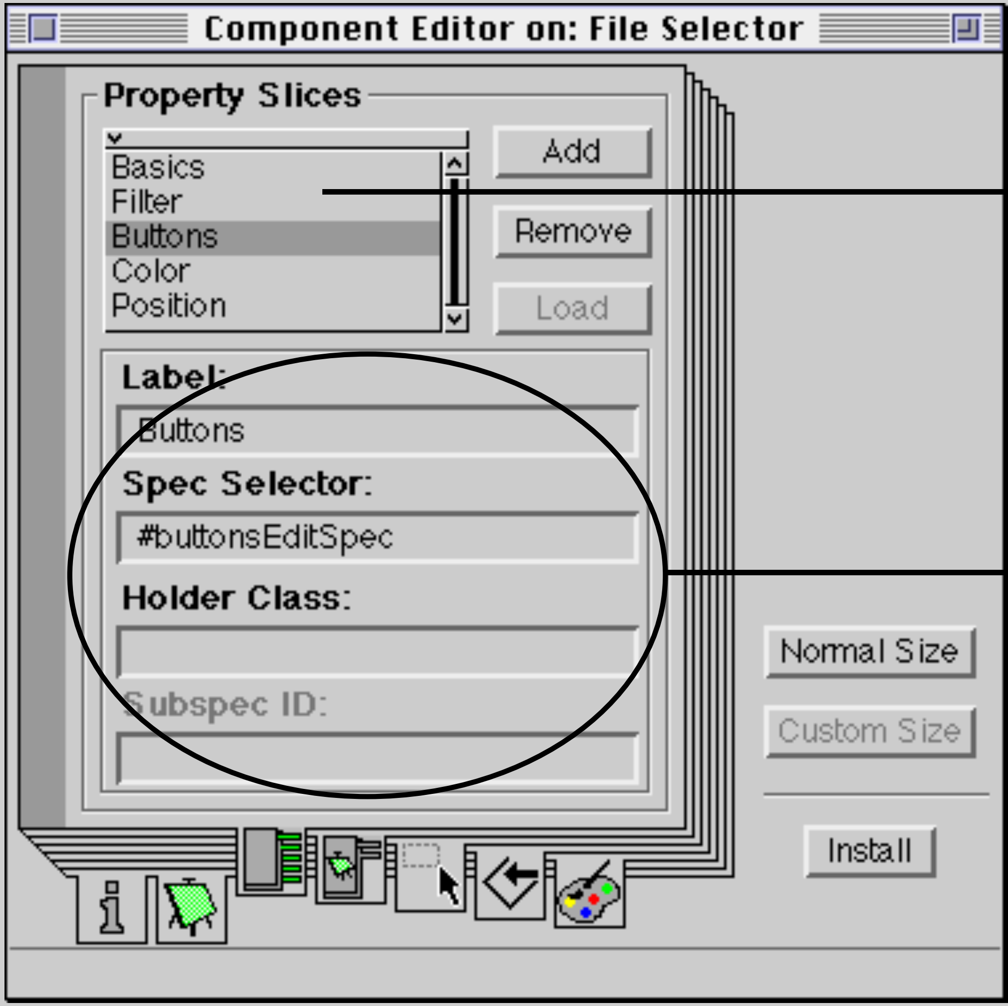
❷ Building the Application Models



Painter embedded in the component editor’s notebook

Perform all the standard application building steps

❸ Defining the Properties Tool Slices

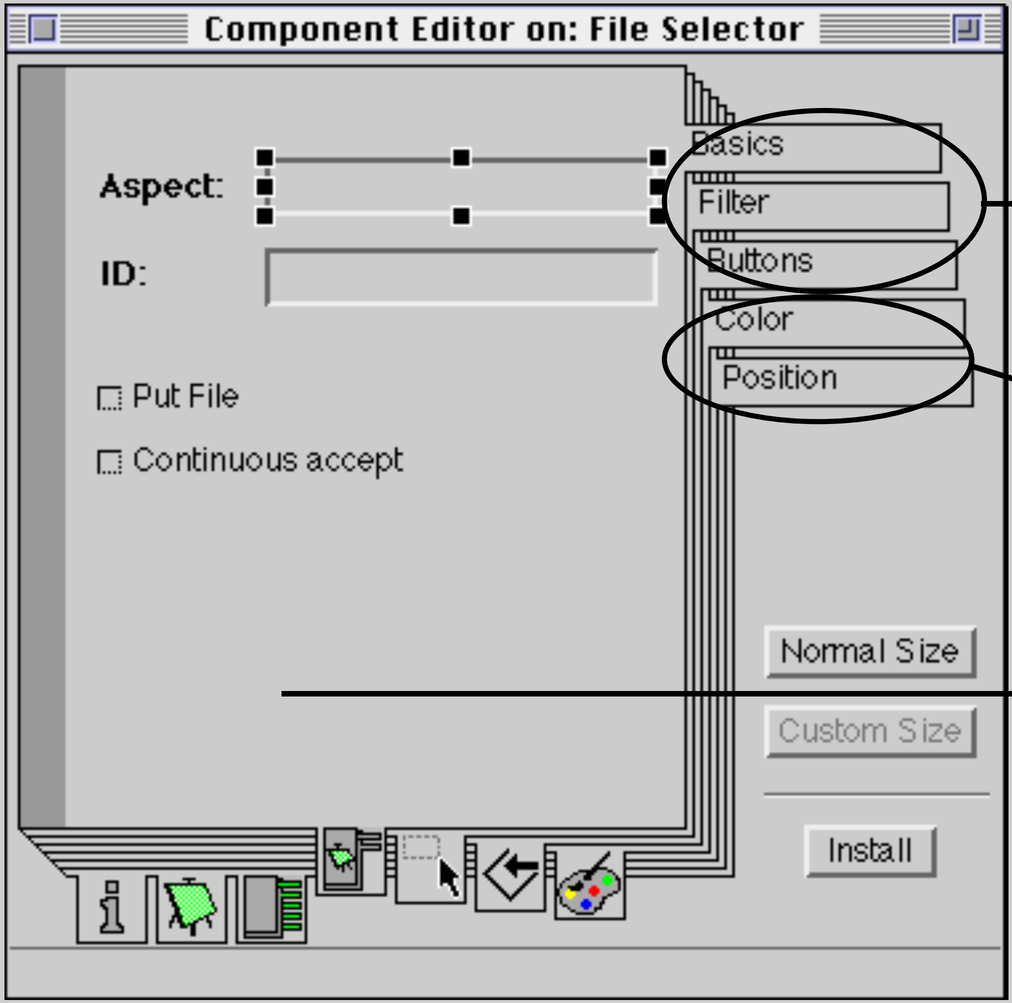


List of slices in the component’s properties tool

Editor for selected slice

Color and position slices cannot be edited but can be removed

❹ Painting the Properties Tool Slices



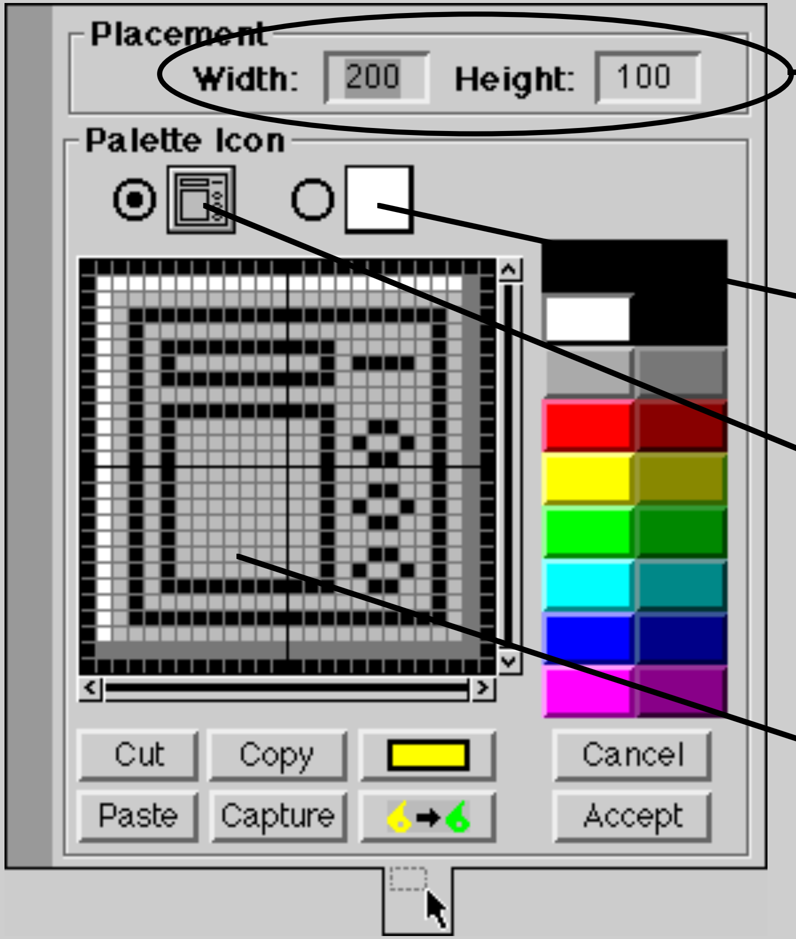
Slices defined in previous step

Standard slices (cannot be edited)

Painter embedded in the component editor's notebook

No install, no define

❺ Supplying Painting Information



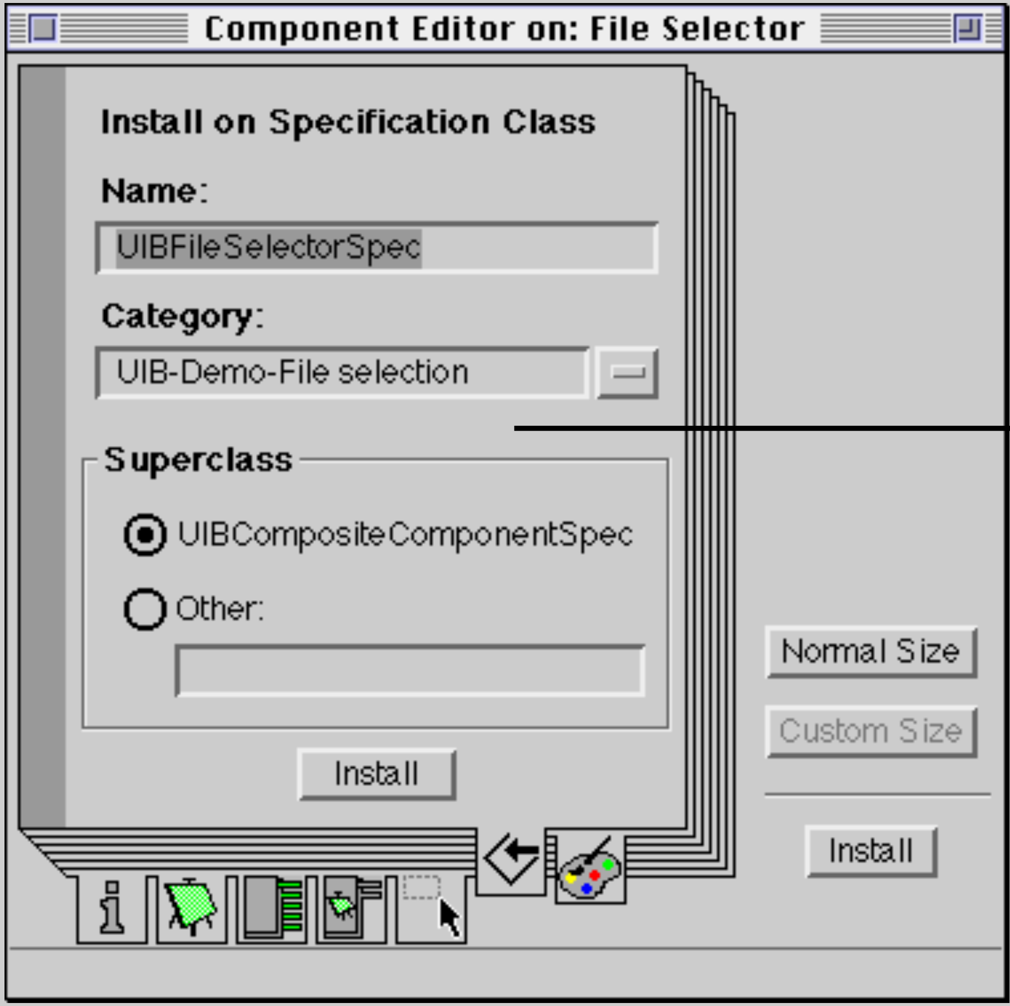
Height and width of component when dragged from a palette

Black & white icon

Color icon

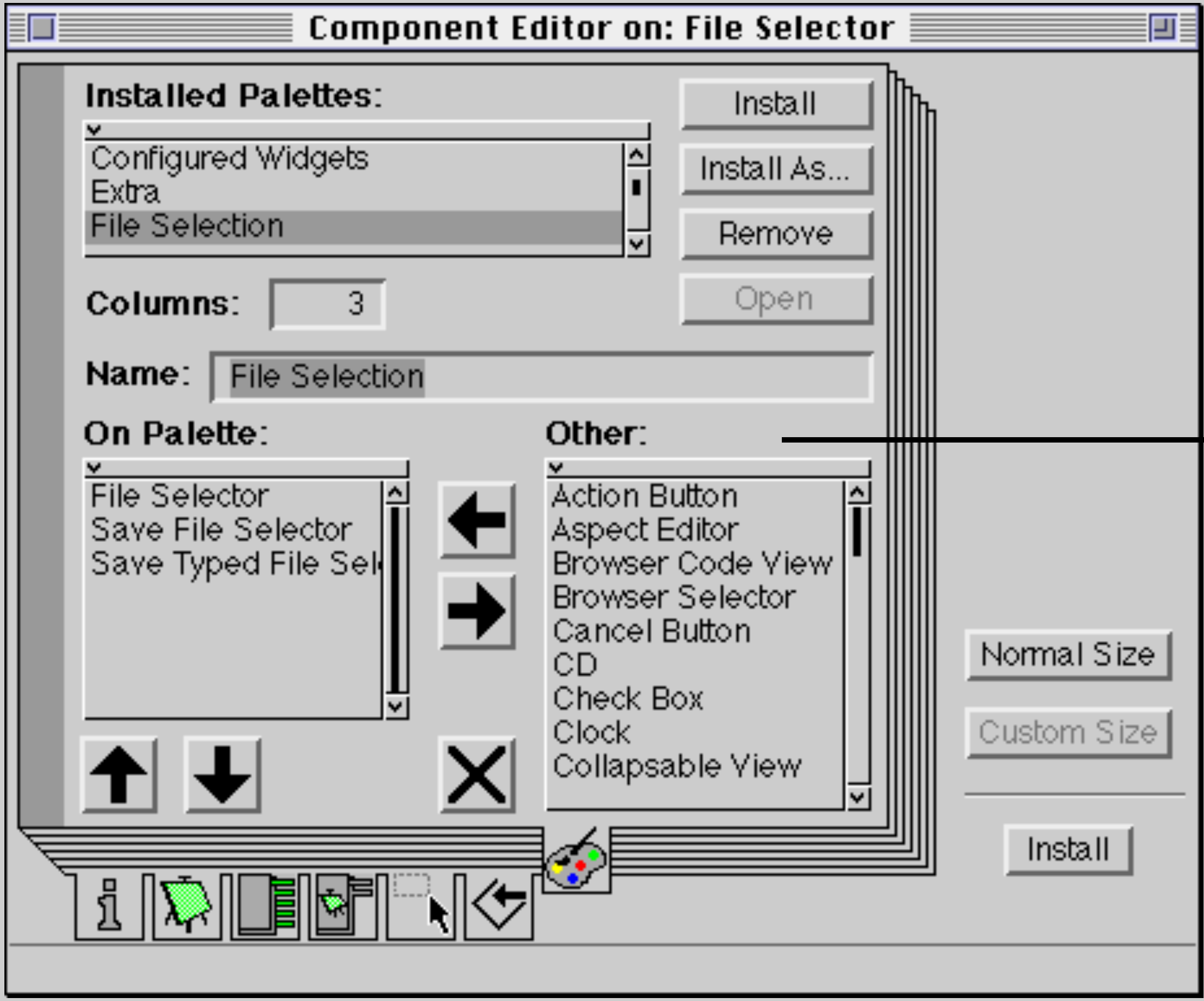
Embedded icon editor

❻ Installing the Component



Embedded installation dialog in component editor’s notebook

❼ Putting the Component on a Palette



Embedded palette editor in component editor’s notebook

More on Properties

- Properties are attributes of a component
- They define the component's
  - look and feel (minorKey and majorKey)
  - link with the domain model (aspect, client, action)
  - visual appearance (label, image, menu)
  - state

Property-Entry Components

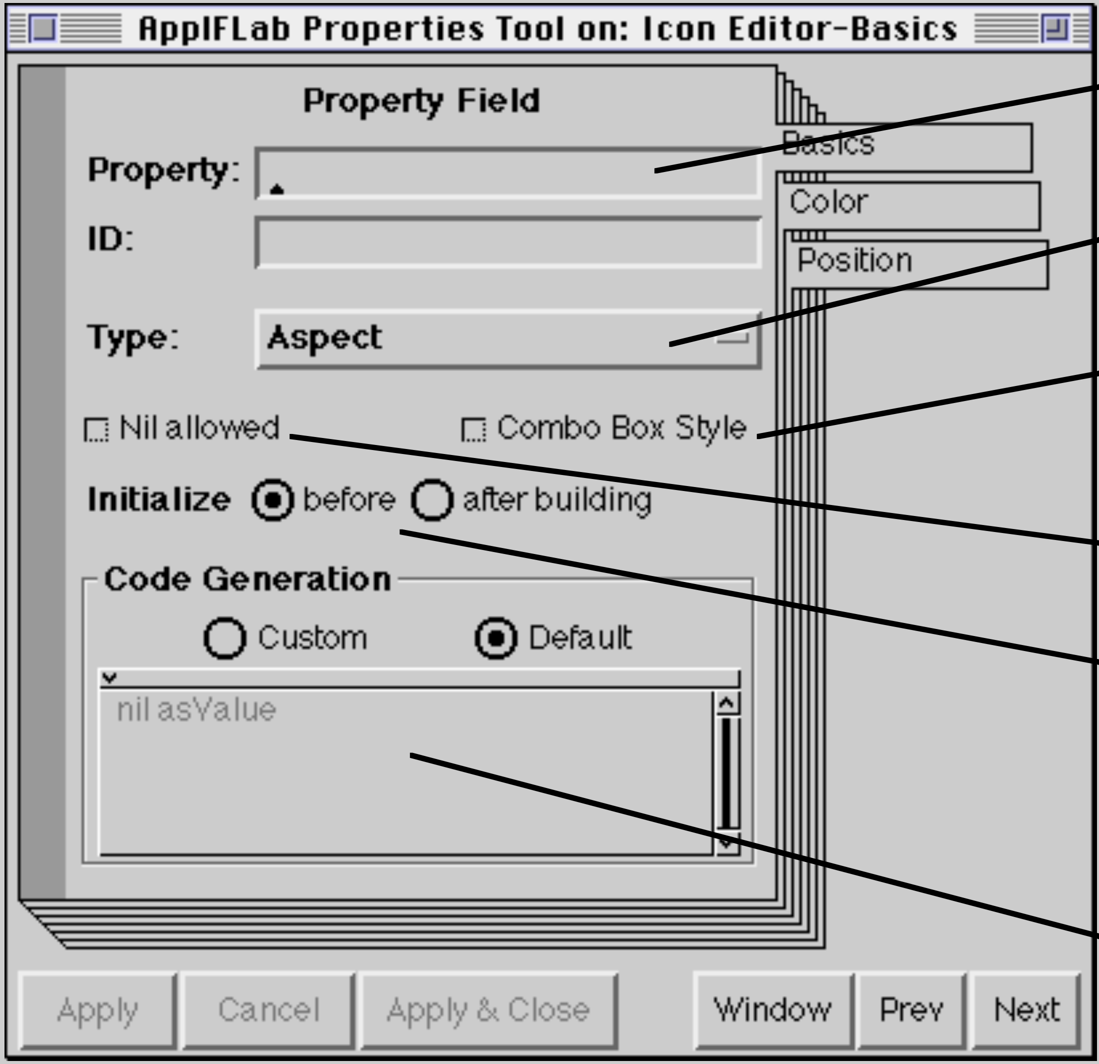
- Property fields
  for properties retrieved from an application model at building time
- Subproperty fields
  for overriding properties of subcomponents
- Standard components
  for simple (state) properties
- Special-purpose property-entry components
  specialisations of standard components for use on properties slices
  now only Property Input Field, probably more later
- Custom-made components
  for properties that require a complex interface to enter them, e.g. labels

Property Field

- A UI component specifically built for use in a slice of the properties tool
- Can only be used when painting properties tool slices!
- Property fields configure a component’s aspect, action, label, visual, menu, client
- Property values are fetched from an application model at building time

Properties palette

Properties of the Property Field



Property name

Property type

Combo Box or
plain input field?

Can the field be left empty?

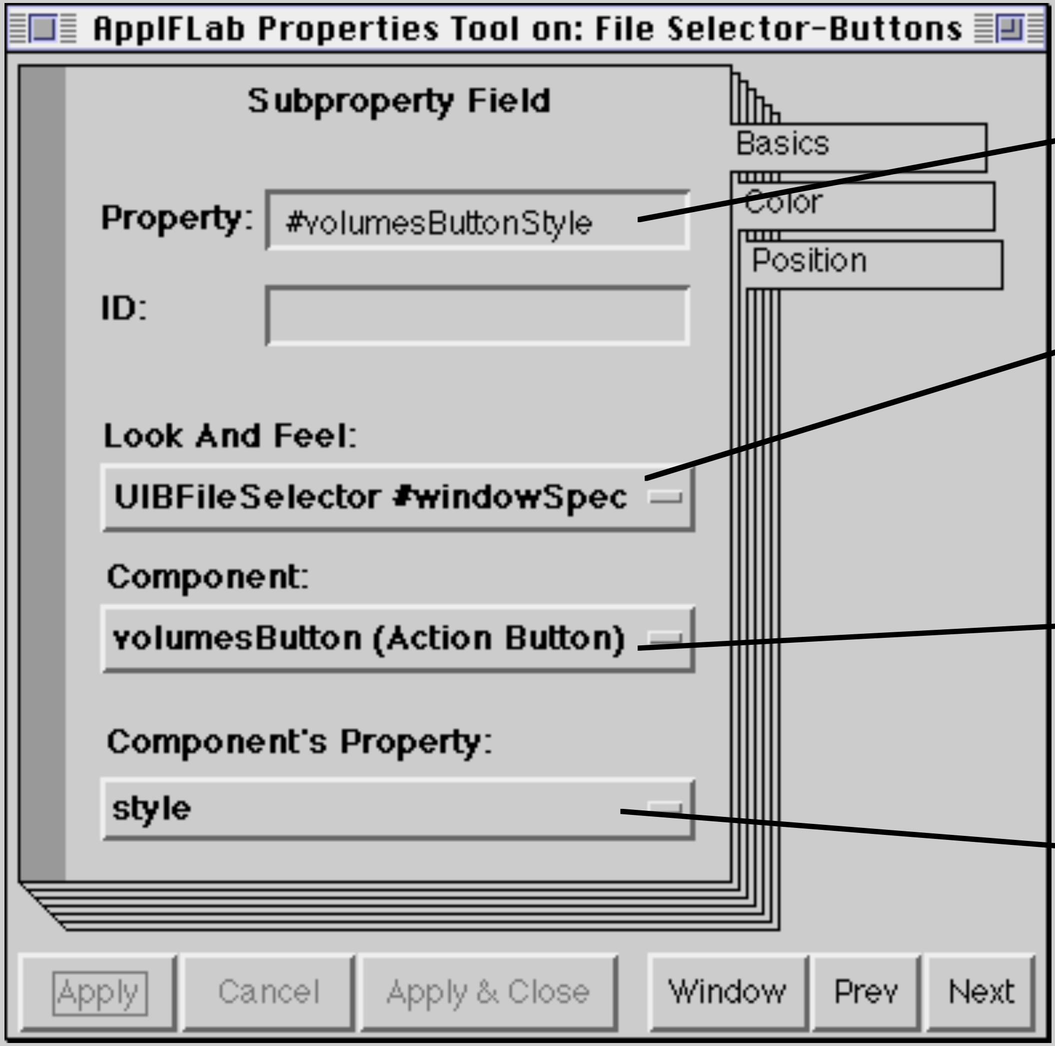
Should this property's
value be set before or after
building the component?

Initialization code
generated by the Definer

Subproperty Fields

- A UI component specifically built for use in a slice of the properties tool
- Can only be used when painting properties tool slices!
- Subproperty fields allow overriding of properties of subcomponents
- Can only be used on properties slices of <u>composite</u> components

Properties of the Subproperty Field



Property name

The look & feel of the component for which this property is valid

The component in the canvas selected above

Property name of the component selected above

Other Property-Entry Components

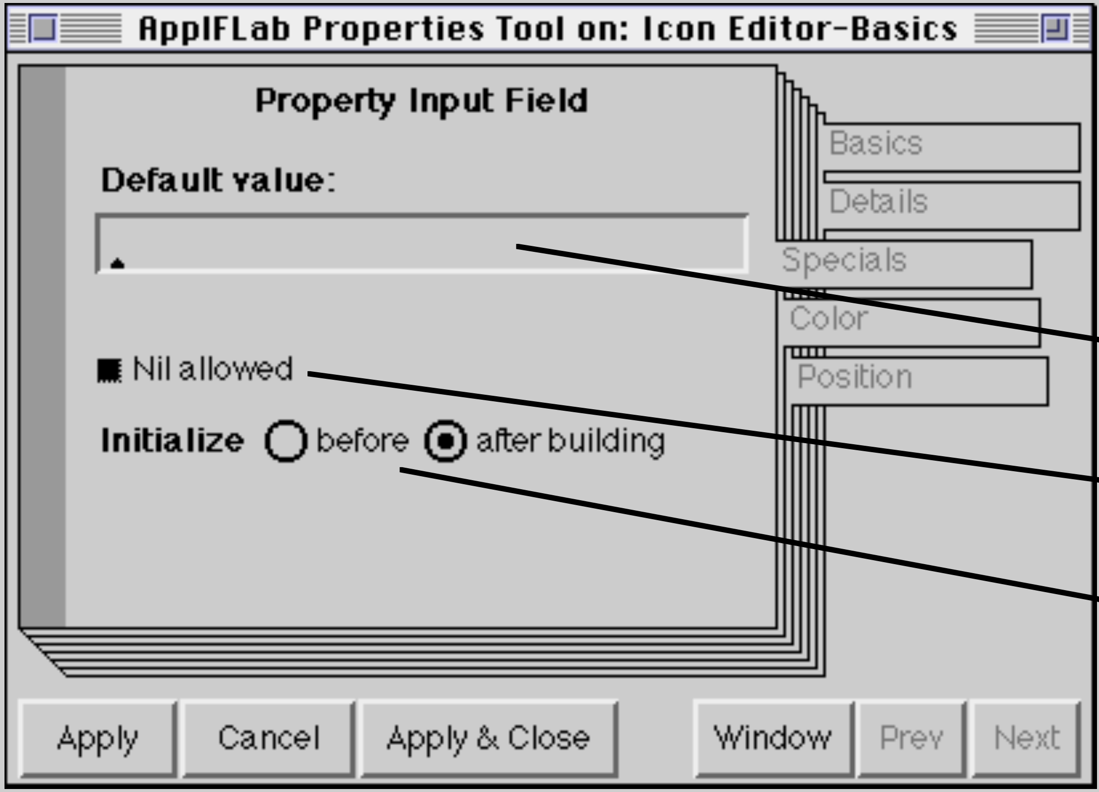
- A standard component can be put on a properties slice when it has only one model and its model understands #literalArrayEncoding
- The same holds for a custom-made component

Use simple models or implement #literalArrayEncoding on your models

Special Property-Entry Components

- Specialisations of standard components
- Created specifically for use on properties slices
- For now only Property Input Field, more later

Properties of Special Property-Entry Components



All special property-entry components share this page

Default value for property

Can the field be left empty?

Should this property's value be set before or after building the component?

How are Property Values Registered?

- Property values are registered before or after the Builder sets up the component (cfr. #preBuildWith: and #postBuildWith:)
- Each property value is copied from the custom component's spec to the application model that implements the component
- This only holds for new properties, not for properties associated with SubCanvasSpec and superclasses

Registration of properties

- The value of property *p* is passed to the application model by sending it the message *#p:*
- Such a method *#p:* is typically found in protocol *initialize-release*

Choosing Between *Before* and *After*

- Choose *Initialize before building*
  - for all properties needed during building by the Builder
  - for most aspect properties!
- Choose *Initialize after building*
  - for properties that are only needed after building
  - typically action properties and simple (state) properties

Tutorial

Overview

- Application building in VisualWorks
- Components
- Building new components
- ApplFLab
- **Example: building a file selector component with ApplFLab**
- Exercise: building components with ApplFLab

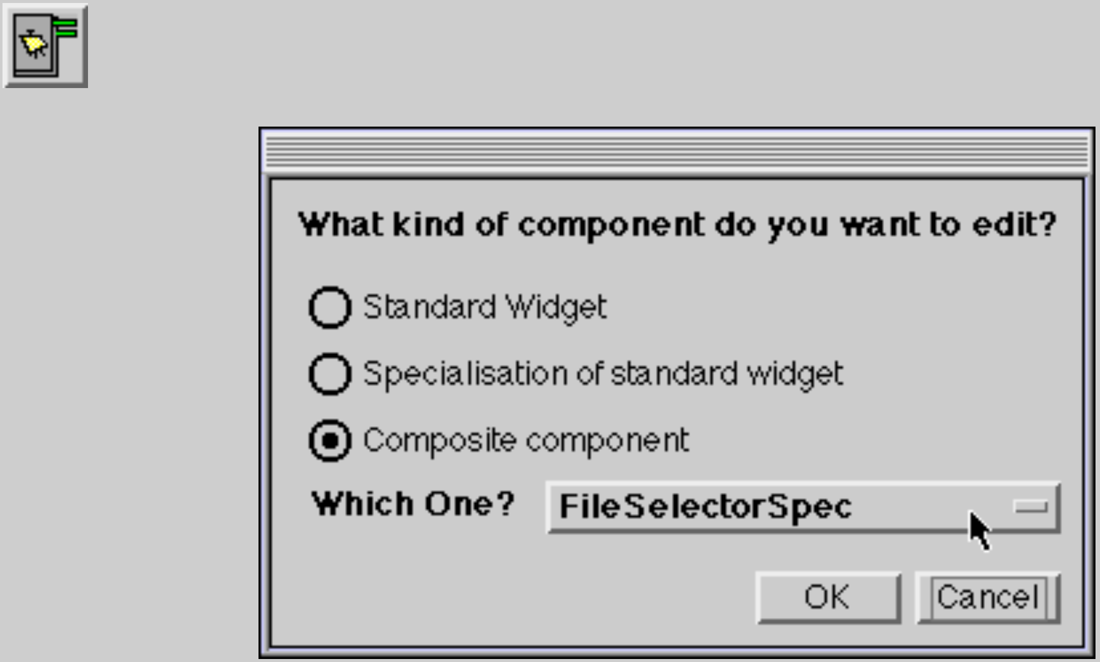
Tutorial | Example

Example: Building a File Selector Component with ApplFLab

- Step by step illustration of the Component Editor on a file selector à la Macintosh

Tutorial

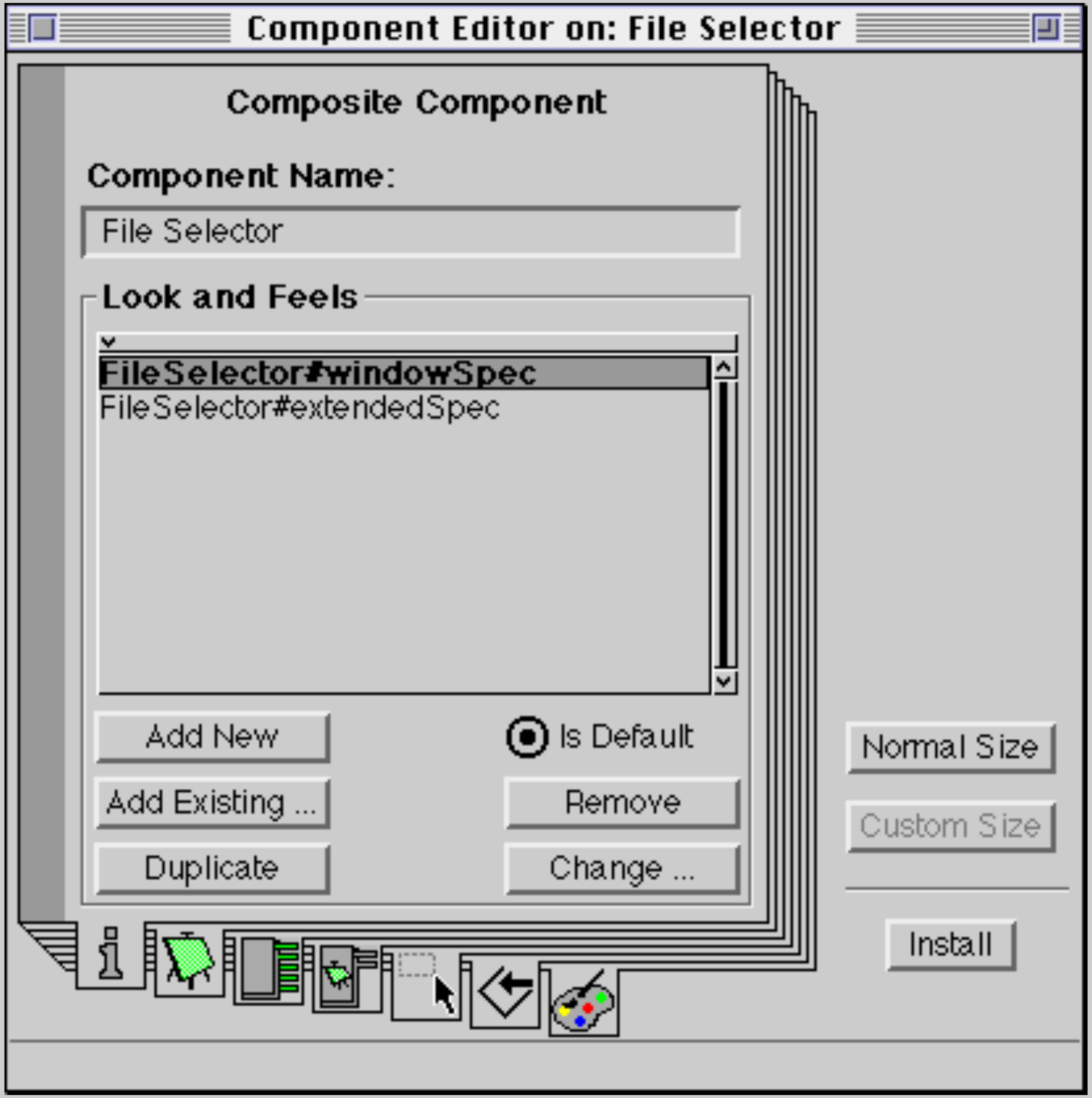
Example

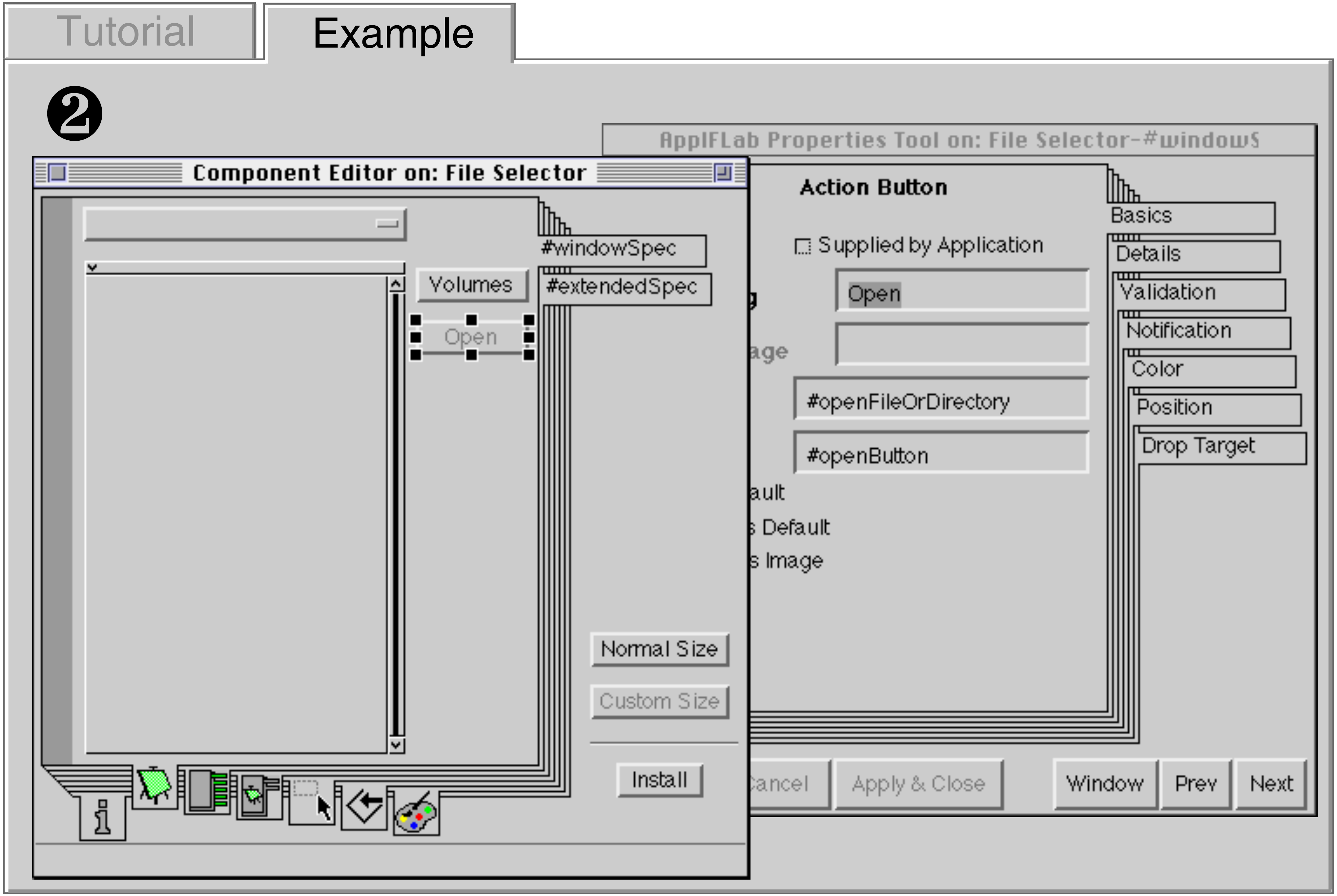


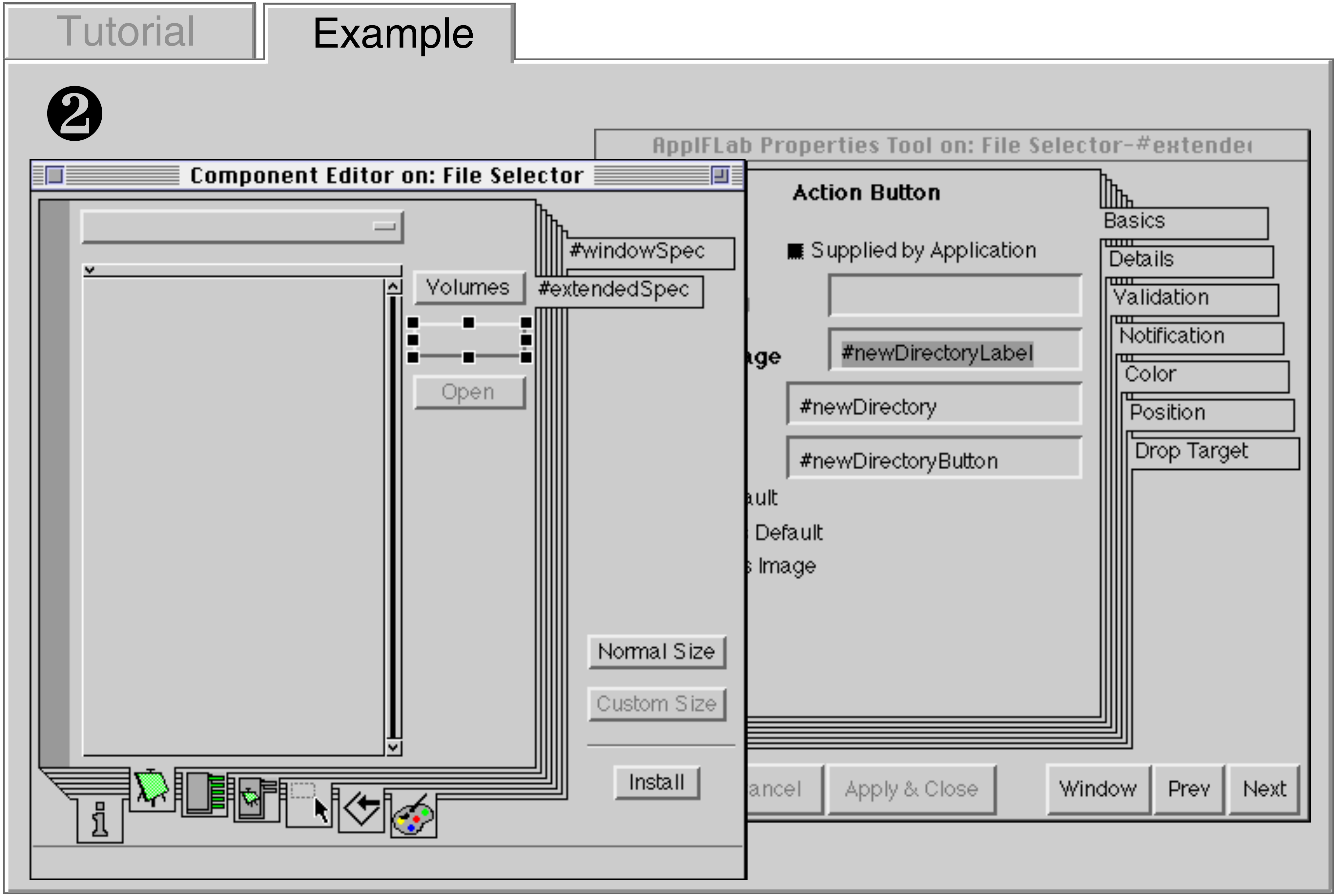
Tutorial

Example

❶

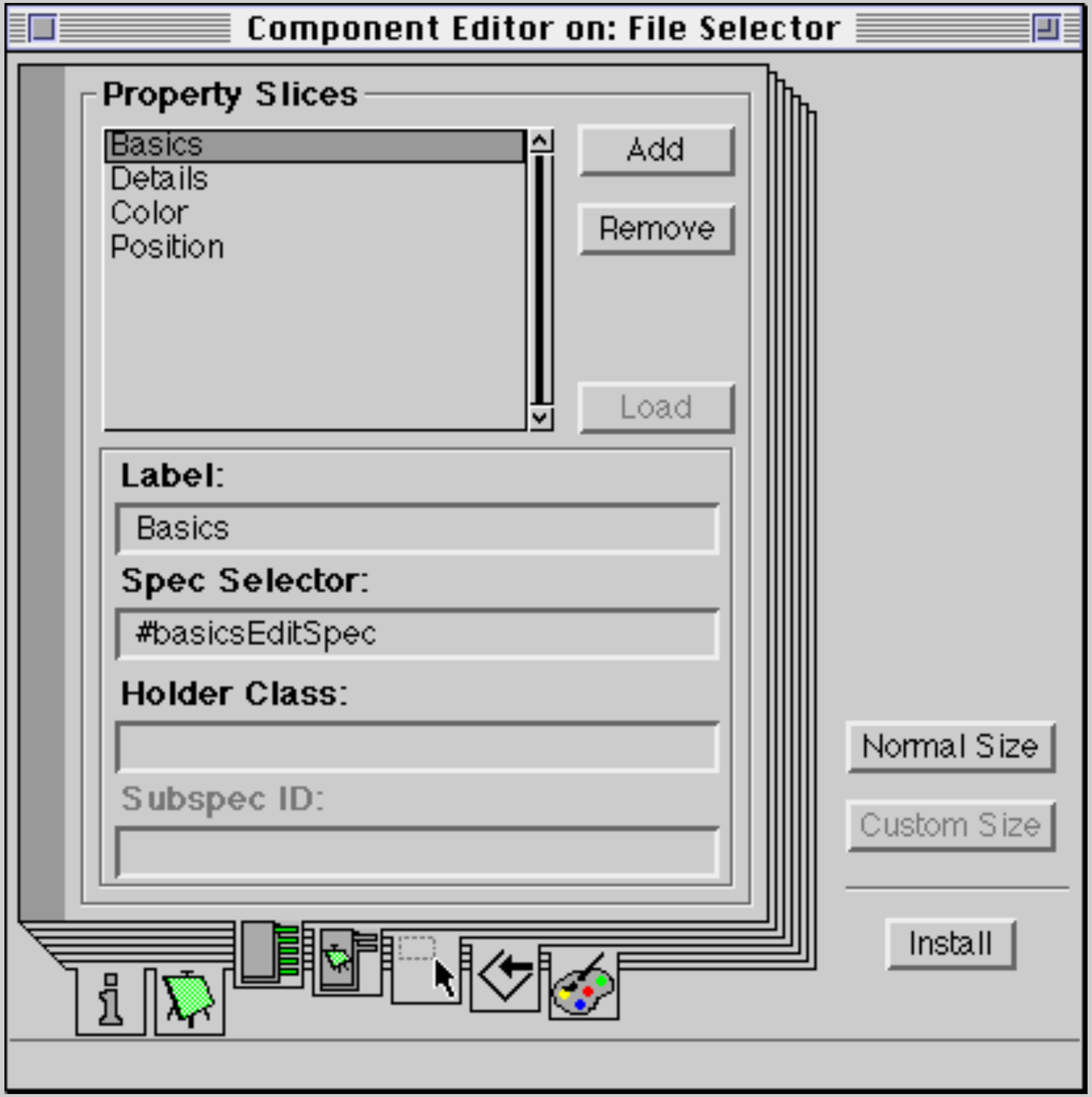


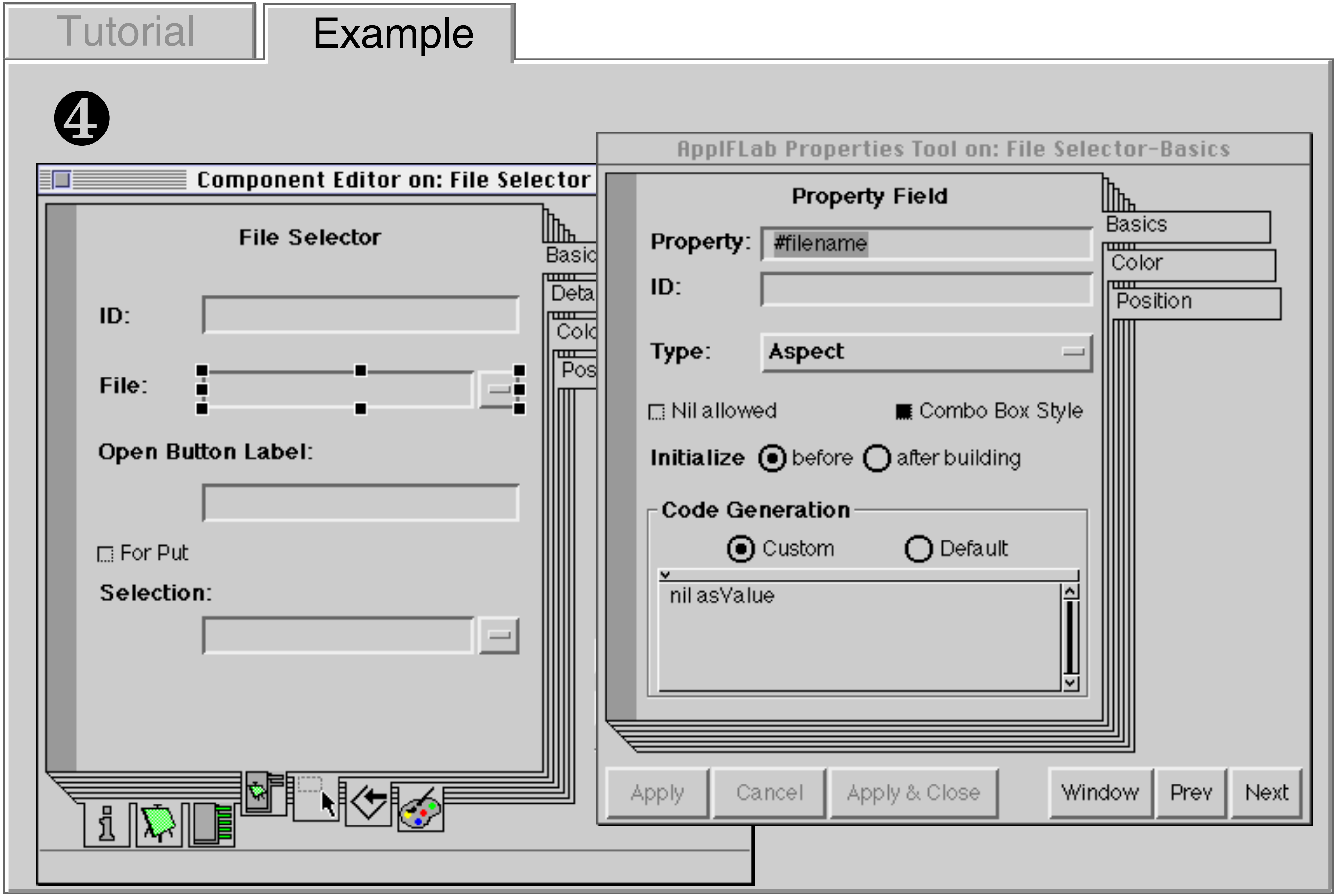

Tutorial
Example
❷
ApplFLab Properties Tool on: File Selector-#windowS
Component Editor on: File Selector
Volumes
Open
#windowSpec
#extendedSpec
Normal Size
Custom Size
Install
Action Button
Supplied by Application
Open
#openFileOrDirectory
#openButton
Basics
Details
Validation
Notification
Color
Position
Drop Target
Apply & Close
Window
Prev
Next

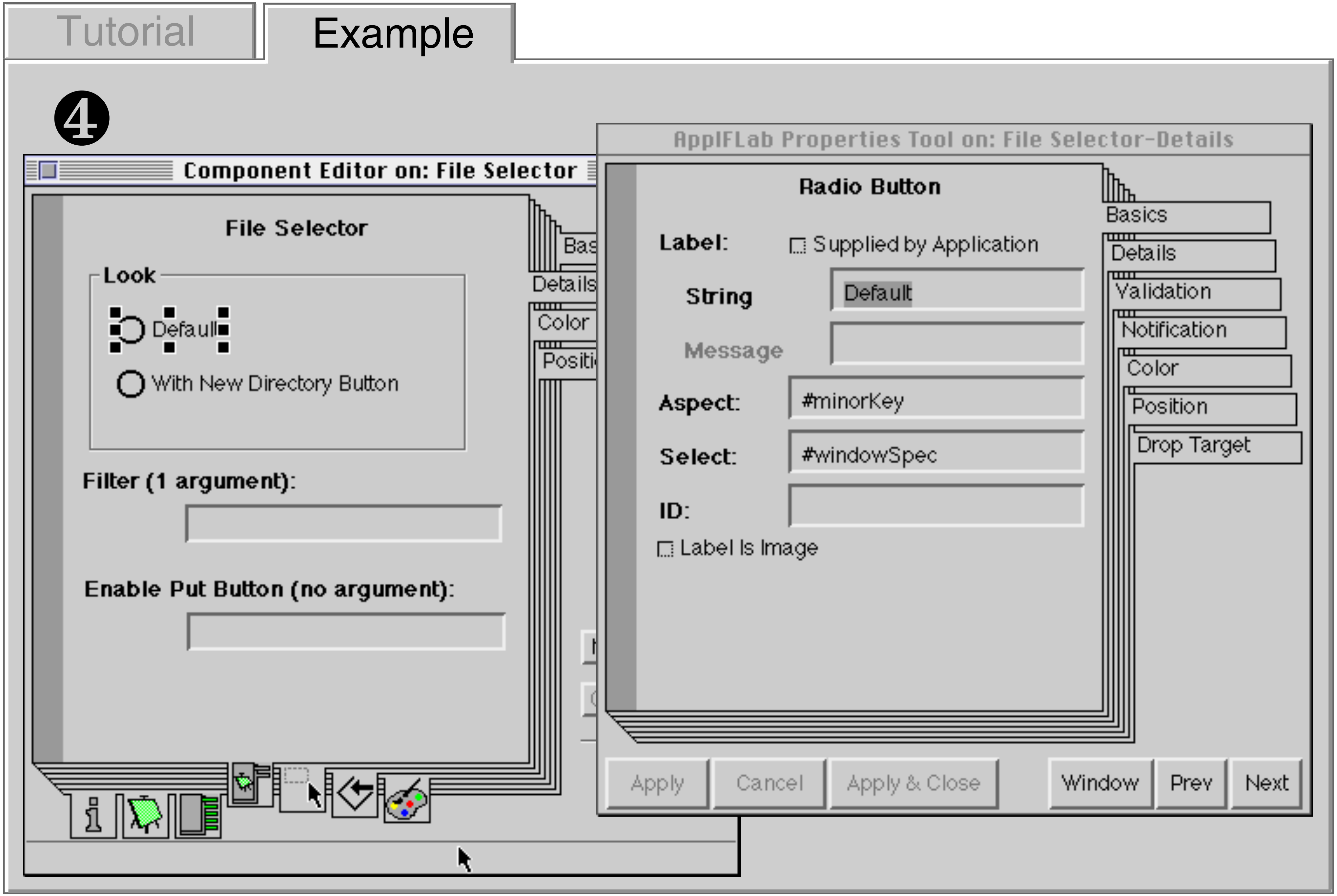

Tutorial
Example
❷
Component Editor on: File Selector
Volumes
Open
#windowSpec
#extendedSpec
Normal Size
Custom Size
Install
ApplFLab Properties Tool on: File Selector-#extende
Action Button
Supplied by Application
#newDirectoryLabel
#newDirectory
#newDirectoryButton
Default
Image
Basics
Details
Validation
Notification
Color
Position
Drop Target
Apply & Close
Window
Prev
Next

Tutorial | Example

❸



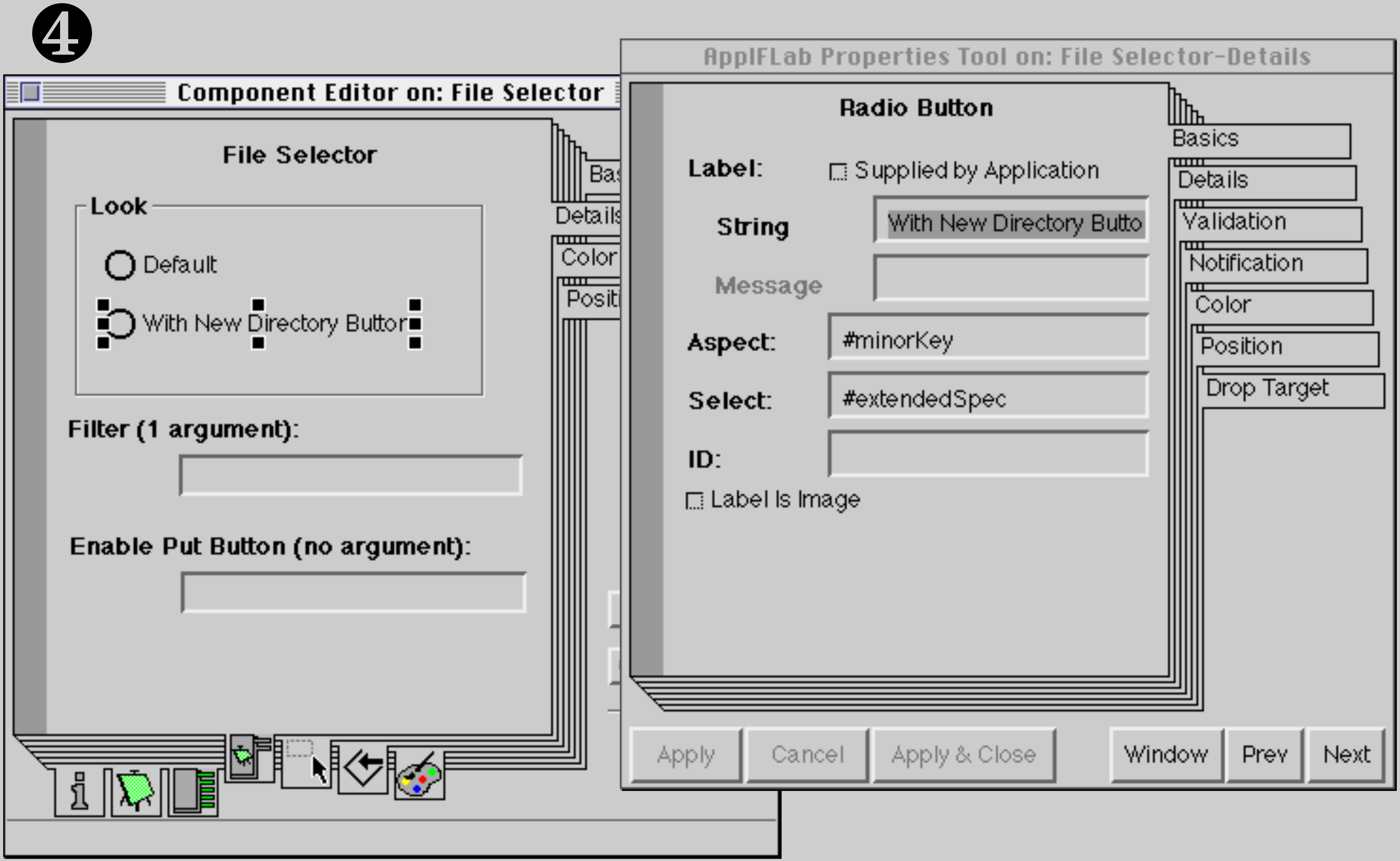

Tutorial
Example
❹
Component Editor on: File Selector
File Selector
ID:
File:
Open Button Label:
For Put
Selection:
Basic
Deta
Colo
Pos
ApplFLab Properties Tool on: File Selector-Basics
Property Field
Property:
#filename
ID:
Type:
Aspect
Nil allowed
Combo Box Style
Initialize
before
after building
Code Generation
Custom
Default
nil asValue
Basics
Color
Position
Apply
Cancel
Apply & Close
Window
Prev
Next

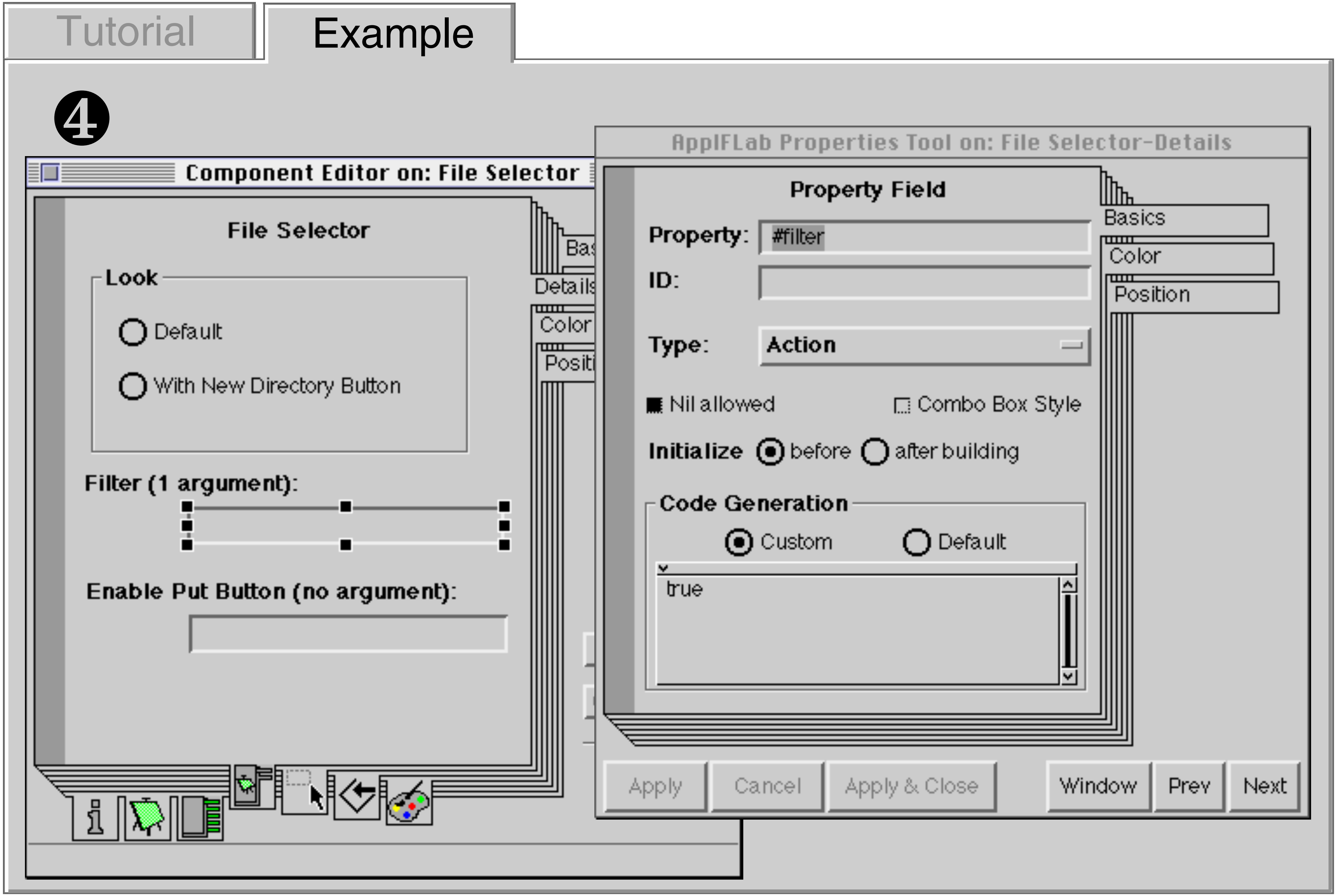


Tutorial
Example
4
Component Editor on: File Selector
File Selector
Look
Default
With New Directory Button
Filter (1 argument):
Enable Put Button (no argument):
Details
Color
ApplFLab Properties Tool on: File Selector-Details
Radio Button
Label:
Supplied by Application
String
Default
Message
Aspect:
#minorKey
Select:
#windowSpec
ID:
Label Is Image
Basics
Details
Validation
Notification
Color
Position
Drop Target
Apply
Cancel
Apply & Close
Window
Prev
Next

Tutorial

Example

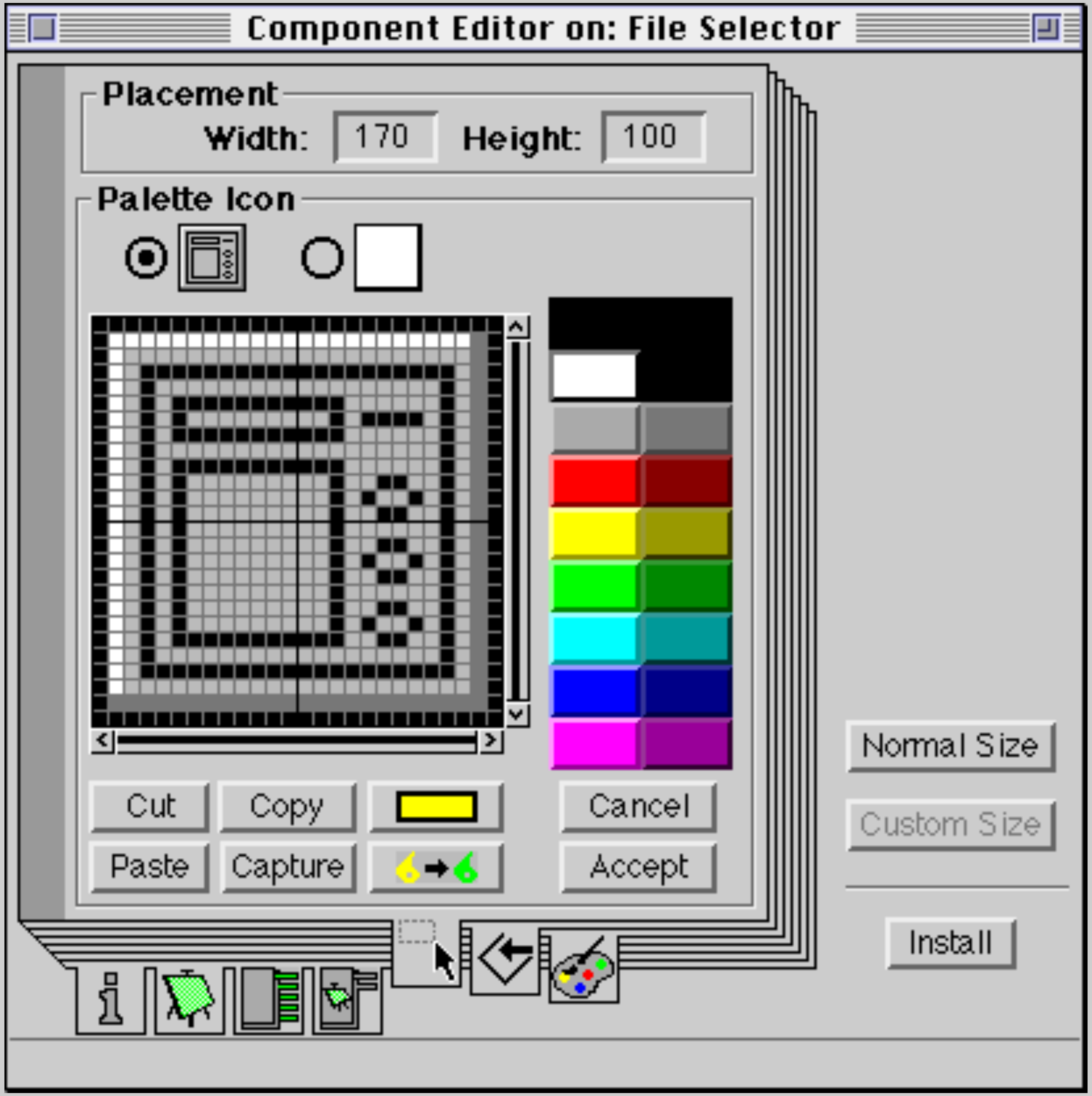



Tutorial
Example
❹
Component Editor on: File Selector
File Selector
Look
Default
With New Directory Button
Filter (1 argument):
Enable Put Button (no argument):
Basics
Details
Color
Position
ApplFLab Properties Tool on: File Selector-Details
Property Field
Property:
#filter
ID:
Type:
Action
Nil allowed
Combo Box Style
Initialize
before
after building
Code Generation
Custom
Default
true
Basics
Color
Position
Apply
Cancel
Apply & Close
Window
Prev
Next

Tutorial

Example

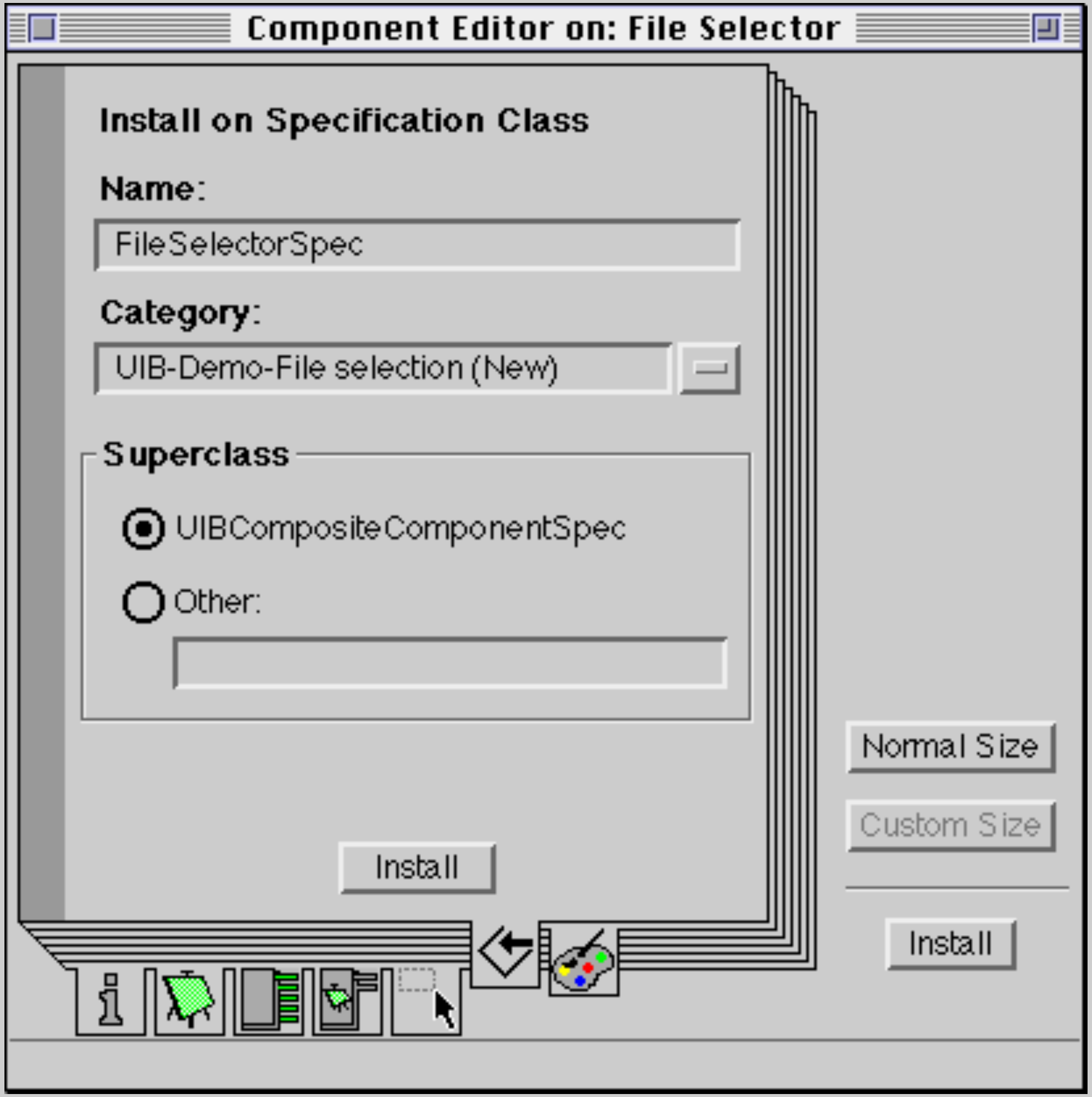
❺



Tutorial

Example

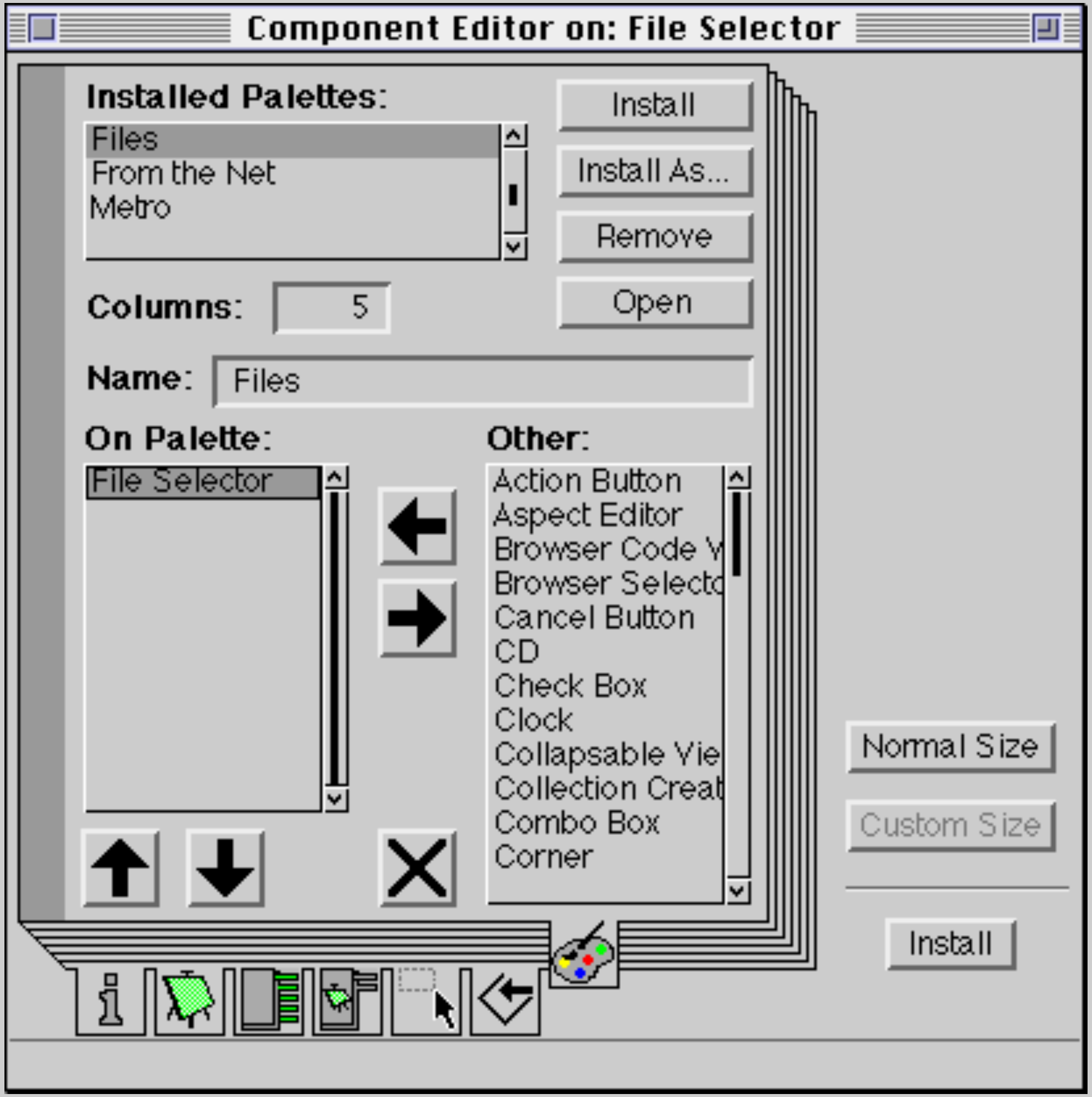
❻



Tutorial

Example

❼



Tutorial

Summary

- Building a new component in VisualWorks is not easy without proper guidance
- ApplFLab’s component editor guides the component designer through all the steps that need to be taken in order to add a new component to the system

Tutorial | **Overview**

- Application building in VisualWorks
- Components
- Building new components
- ApplFLab
- Example: building a file selector component with ApplFLab

- **Exercise: building components with ApplFLab**

Exercise

- With detailed notes:
  - Build a new component starting from an existing application model
  - Test in an example application
  - Extend the component and test it again
- On your own:
  - Build a component from scratch
  - Further exploration of ApplFLab