

VisualAge Smalltalk



# Ultra Light Client Guide and Reference

*Version 5.5*

**Note**

Before using this document, read the general information under “Notices” on page vii.

**May 2001**

This edition applies to Version 5.5 of the VisualAge Smalltalk products, and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product. The term “VisualAge”, as used in this publication, refers to the VisualAge Smalltalk product set.

Portions of this book describe materials developed by Object Technology International Inc. of Ottawa, Ontario, Canada. Object Technology International Inc. is a subsidiary of the IBM Corporation.

If you have comments about the product or this document, address them to: IBM Corporation, Attn: IBM Smalltalk Group, 621-107 Hutton Street, Raleigh, NC 27606-6324. You can fax comments to (919) 828-9633.

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1999, 2000. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

---

# Contents

<b>Notices</b>	<b>vii</b>
Trademarks	vii

<b>About this book</b>	<b>ix</b>
Who this book is for	ix
Conventions used in this book	ix
Tell us what you think	x

<b>What's new</b>	<b>xi</b>
What's new in ULC for Version 5.5.2	xi
What's new in ULC for Version 5.5 and Version 5.5.1	xii
What's new in this guide	xiv

---

## Part 1. User's Guide . . . . . 1

<b>Chapter 1. Overview of ULC</b>	<b>3</b>
Half objects.	3
How the UI Engine works.	5

<b>Chapter 2. Setting up ULC</b>	<b>7</b>
What you need for ULC desktop workstations	7
Setting up Java support.	9
Setting up the sample ULC HTTP server	10
Configuring Internet Explorer for ULC	10
Configuring Netscape Navigator for ULC	11

<b>Chapter 3. Running ULC components</b>	<b>13</b>
Using the UI Engine	13
Running the UI Engine as a standalone application	13
Running the UI Engine as an applet	13
Running UI Engine as a helper application	14
UI Engine command options	14
Using the sample ULC HTTP server	15
Using Application Controller	17

<b>Chapter 4. About building ULC applications</b>	<b>19</b>
How ULC compares with common widget protocol	19
ULC class overview	19
The simple widgets.	20
ULC layout	20
ULC layout design tips	22
Using layout widgets	23
Shells	24
About model-based widgets.	24
Using models with model-based widgets	25
ULC and Server Smalltalk	26
Opening new ULC views.	27
Handling Strings in ULC applications	28
Running applications in Compatibility Mode	29
Migrating to Unicode	29

Enabling national language support in ULC applications	30
Differences between ULC and standard	
VisualAge NLS mechanisms	31
Implementing NLS support for ULC applications	31

## Chapter 5. Building ULC applications visually . . . . . 33

Visual composition pitfalls in ULC	34
Using ULC visual parts	34
Defining layout	34
Adding widgets	35
Setting layout properties	35
Changing the ULC layout grid	36
Building the To-Do List with ULC	36
Creating a ULC application	36
Creating a new ULC visual part	36
Setting the layout	37
Adding the remaining parts	37
Connecting the parts	38
Testing the application.	39
Adding support for enablers.	39
Enabling reuse with ULC composite parts	40
Using ULC nonvisual parts	40
Working with Variable parts.	40
Working with Form Model parts	41
Working with Table Model parts	42
Working with Tree Model parts.	42
Working with Data Models	45
Working with Row Models	46

## Chapter 6. Deploying ULC-based applications . . . . . 49

Packaging ULC-based applications in XD	49
Preparing ULC visual and composite parts	49
Registering the ULC visual application class	49
Creating and populating the passive image.	50
Creating the packaging instructions and outputting the image	50
Setting up a ULC development image to run in production mode	51
About running ULC applications from a command prompt.	52

## Chapter 7. Implementing ULC objects 55

Implementing the widget.	55
Implementing the UI half.	55
Implementing the faceless half	57
Implementing the faceless half in Java	57
Implementing the faceless half in Smalltalk.	58

## Chapter 8. Advanced Techniques . . . 61

ULC Client Reconnect Feature	61
Introduction	61

	Keep-Alive Timeout . . . . .	61
	ULC Reconnect Token Strategy . . . . .	61
	How to Write a ULC Reconnect Token Strategy . . . . .	62
	Enable Client Reconnect in Runtime Time . . . . .	63
	Enable Client Reconnect in Development Time . . . . .	64
	Packaging . . . . .	64
	Known Problems . . . . .	64
	Server Support . . . . .	64
	Introduction . . . . .	64
	Monitoring at Development Time . . . . .	65
	Including the Monitor in Packaged Images . . . . .	65
	How to write a Custom UI for the Monitor . . . . .	65
	Support for ENVY/Stats . . . . .	66
	Overview . . . . .	66
	Using the ULC Profiler . . . . .	66
	Test Workbench . . . . .	66
	Standardized Test Format. . . . .	66
	Workbench . . . . .	67
	Create Test Suite. . . . .	68
	Create Test . . . . .	68
	Edit Test . . . . .	69
	Test Overview . . . . .	69

## Chapter 9. Troubleshooting ULC applications . . . . . 71

Named ULC contexts . . . . .	71
Cleaning up the ULC system . . . . .	71
Using the Debugger window with ULC . . . . .	72
Configuring #debugPrintString . . . . .	72
Tracing inside the Smalltalk image. . . . .	72
Default ULC debugging aspects . . . . .	73
Defining application-specific debugging aspects . . . . .	73
Customizing exception handling by context . . . . .	74
Default exception handling in ULC . . . . .	74
Implementing custom exception handling . . . . .	75
Customization considerations . . . . .	76

## Chapter 10. Frequently asked questions . . . . . 77

General Questions . . . . .	77
How do I run the UI Engine as an applet? . . . . .	77
How do I increase the Environment under Win95? . . . . .	77
How do I run ULC on a standalone machine? . . . . .	77
How can I request a remote ULC server to exit? . . . . .	78
How much code is downloaded when connecting to a ULC application? . . . . .	78
What kind of applications is ULC designed for? . . . . .	78
How can I convert my standalone application to a ULC server application? . . . . .	78
What happens when a connection fails? . . . . .	79
Why do I get an error "ULC Transport Unable to load defaults"? . . . . .	79
Why am I not able to connect a sample ULC application to a UIEngine on the same machine? . . . . .	79
In ULC lists (Table, TreeTable), how is addPreloadColumns different from addPreloadAttributes? . . . . .	80
What can I do to tune the performance of my ULC List widgets (Tree, Table, TreeTable) ? . . . . .	80

My long running ULC Java server seems to be leaking memory. What can I do? . . . . .	80
Smalltalk Questions . . . . .	81
Why does the Test button in the Composition Editor stop working? . . . . .	81
How do I inspect the objects in a ULC application? . . . . .	81
Why do I get a Debugger window when saving the public interface of an object? . . . . .	81
Why does the development image not respond when I start a ULC application? . . . . .	81
How can I add a new menu to an existing menu or menubar? . . . . .	81
How do you inspect the Objects being used within a Smalltalk ULC Application? . . . . .	82
Why do I get a debugger when saving the public interface of an object? . . . . .	82
How can I find a widget that has disappeared? . . . . .	82
How can I execute application specific image startup tasks? . . . . .	82
How can I open new ULC views? . . . . .	82
Packaging when using non-ULC parts in a variable. . . . .	83
Why do I see two connections when starting a Smalltalk example (exe)? . . . . .	83
Can I have equal objects in a list? . . . . .	83

## Part 2. Programmer's Reference . . . . . 85

### Chapter 11. Resource classes . . . . . 87

### Chapter 12. General classes . . . . . 89

### Chapter 13. Alert . . . . . 91

### Chapter 14. Box parts . . . . . 93

Box attributes. . . . .	93
Box events. . . . .	94
Box general advice . . . . .	94

### Chapter 15. Browser Context . . . . . 97

Browser Context attributes . . . . .	97
Browser Context events . . . . .	97

### Chapter 16. Button . . . . . 99

Button attributes. . . . .	99
Button events. . . . .	99
Button general advice . . . . .	99

### Chapter 17. CheckBox . . . . . 101

CheckBox attributes . . . . .	101
CheckBox events . . . . .	102
CheckBox general advice . . . . .	102

### Chapter 18. CheckBox Menu Item. . . . . 103

CheckBox Menu Item attributes . . . . .	103
CheckBox Menu Item events . . . . .	103
CheckBox Menu Item general advice . . . . .	104

<b>Chapter 19. Column.</b>	<b>105</b>	<b>Chapter 32. Menubar</b>	<b>135</b>
Column attributes . . . . .	105	Menubar attributes . . . . .	135
Column events . . . . .	105	Menubar events . . . . .	135
Column general advice . . . . .	105	Menubar general advice . . . . .	135
<b>Chapter 20. ComboBox</b>	<b>107</b>	<b>Chapter 33. Menu Item</b>	<b>137</b>
ComboBox attributes . . . . .	107	Menu Item attributes . . . . .	137
ComboBox events . . . . .	108	Menu Item events . . . . .	137
<b>Chapter 21. Container.</b>	<b>109</b>	Menu Item general advice . . . . .	137
Container attributes . . . . .	109	<b>Chapter 34. Menu Separator</b>	<b>139</b>
Container events . . . . .	109	Menu Separator attributes . . . . .	139
Container general advice . . . . .	110	Menu Separator events . . . . .	139
<b>Chapter 22. Data Model</b>	<b>111</b>	Menu Separator general advice . . . . .	139
<b>Chapter 23. Enabler.</b>	<b>113</b>	<b>Chapter 35. Notebook</b>	<b>141</b>
Enabler general advice . . . . .	113	Notebook attributes . . . . .	141
<b>Chapter 24. Field parts</b>	<b>115</b>	Notebook events . . . . .	142
Field attributes . . . . .	115	Notebook general advice . . . . .	142
Field events . . . . .	116	<b>Chapter 36. Page</b>	<b>143</b>
Field general advice . . . . .	116	Page attributes . . . . .	143
<b>Chapter 25. Filler</b>	<b>119</b>	Page events . . . . .	143
Filler attributes . . . . .	119	<b>Chapter 37. Pagebook.</b>	<b>145</b>
Filler events . . . . .	119	Pagebook attributes . . . . .	145
Filler general advice . . . . .	119	Pagebook events . . . . .	146
<b>Chapter 26. Form Model</b>	<b>121</b>	Pagebook general advice . . . . .	146
Form Model attributes . . . . .	121	<b>Chapter 38. Progress Bar</b>	<b>149</b>
Form Model events . . . . .	121	Progress Bar attributes . . . . .	149
Form Model general advice . . . . .	121	Progress Bar events . . . . .	149
<b>Chapter 27. GroupBox</b>	<b>123</b>	Progress Bar general advice . . . . .	149
GroupBox attributes . . . . .	123	<b>Chapter 39. RadioButton</b>	<b>151</b>
GroupBox events . . . . .	123	RadioButton attributes . . . . .	151
GroupBox general advice . . . . .	123	RadioButton events . . . . .	152
<b>Chapter 28. Html Pane</b>	<b>125</b>	RadioButton general advice . . . . .	152
Html Pane attributes . . . . .	125	<b>Chapter 40. Radio Group</b>	<b>153</b>
Html Pane events . . . . .	126	<b>Chapter 41. Row Model</b>	<b>155</b>
Html Pane general advice . . . . .	126	Row Model attributes . . . . .	155
<b>Chapter 29. Label.</b>	<b>127</b>	Row Model general advice . . . . .	155
Label attributes . . . . .	127	<b>Chapter 42. Shell</b>	<b>157</b>
Label events . . . . .	128	Shell attributes . . . . .	157
Label general advice . . . . .	128	Shell events . . . . .	157
<b>Chapter 30. List</b>	<b>129</b>	Shell general advice . . . . .	158
List attributes . . . . .	129	<b>Chapter 43. Slider</b>	<b>159</b>
List events . . . . .	130	Slider attributes . . . . .	159
List general advice . . . . .	130	Slider events . . . . .	160
<b>Chapter 31. Menu.</b>	<b>133</b>	Slider general advice . . . . .	160
Menu attributes . . . . .	133	<b>Chapter 44. Split Pane</b>	<b>161</b>
Menu events . . . . .	133	Split Pane attributes . . . . .	161
Menu general advice . . . . .	133	Split Pane events . . . . .	162

<b>Chapter 45. Table.</b>	<b>163</b>	<b>Chapter 52. Resize Window</b>	<b>181</b>
Table attributes . . . . .	163	<b>Chapter 53. Open &lt;BoxClassName&gt; Settings.</b>	<b>183</b>
Table events . . . . .	164	<b>Chapter 54. Align.</b>	<b>185</b>
Table general advice . . . . .	164	<b>Chapter 55. Hide Box Margin/Show Box Margin</b>	<b>187</b>
<b>Chapter 46. Table Model.</b>	<b>167</b>	<b>Chapter 56. Row</b>	<b>189</b>
Table Model attributes . . . . .	167	<b>Chapter 57. Column.</b>	<b>191</b>
Table Model events . . . . .	167	<b>Chapter 58. Switch to page</b>	<b>193</b>
Table Model general advice. . . . .	168	<b>Chapter 59. Font prompt</b>	<b>195</b>
<b>Chapter 47. ToolBar.</b>	<b>169</b>	<b>Chapter 60. Icon property</b>	<b>197</b>
ToolBar attributes . . . . .	169	<b>Chapter 61. Converter prompt</b>	<b>199</b>
ToolBar events . . . . .	170	<b>Chapter 62. Accelerator prompt</b>	<b>201</b>
<b>Chapter 48. Tree</b>	<b>171</b>	<hr/>	
Tree attributes . . . . .	171	<b>Part 4. Appendixes</b>	<b>203</b>
Tree events . . . . .	172	<b>Index</b>	<b>205</b>
<b>Chapter 49. Tree Model</b>	<b>173</b>		
Tree Model attributes. . . . .	173		
Tree Model events. . . . .	173		
Tree Model general advice . . . . .	173		
<b>Chapter 50. TreeTable.</b>	<b>175</b>		
TreeTable attributes . . . . .	175		
TreeTable events . . . . .	176		
<hr/>			
<b>Part 3. Reference</b>	<b>177</b>		
<b>Chapter 51. Hide Margin/Show Margin</b>	<b>179</b>		

---

## Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of the intellectual property rights of IBM may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY, USA 10594.

IBM may change this publication, the product described herein, or both.

---

## Trademarks

The following terms are trademarks of the IBM Corporation in the United States, other countries, or both:

VisualAge

AIX	IMS	S/390
AS/400	MVS	SOM
CICS	OS/2	VisualAge
IBM	Object Connection	

The following terms are trademarks or registered trademarks of other companies:

Unicode                      Unicode, Inc.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product or service names may be the trademarks or service marks of others.





---

## About this book

The purpose of this book is to introduce you to the following:

- The basic concepts and terms needed for using Ultra Light Client (ULC)
- The fundamentals you need to know to create an application using ULC

This book is divided into the following parts:

- “Part 1. User’s Guide” on page 1 describes how ULC and how to use it.
- “Part 2. Programmer’s Reference” on page 85 describes the parts you use to visually construct ULC applications.

---

## Who this book is for

This book is written for anybody who wants to become familiar with the basic use of ULC. Basic understanding of VisualAge Smalltalk and visual construction is required to use this book. For developing industrial-strength applications, an understanding of the Server Smalltalk (SST) feature is also required.

---

## Conventions used in this book

These highlighting conventions are used in the text:

Highlight style	Used for	Example
Boldface	New terms the first time they are used	VisualAge uses <b>construction from parts</b> to develop software by assembling and connecting reusable components called <b>parts</b> .
	Items you can select, such as push buttons and menu choices	Select <b>Add Part</b> from the <b>Options</b> pull-down. Type the part’s class and select <b>OK</b> .
<i>Italics</i>	Special emphasis	Do <i>not</i> save the image.
	Titles of publications	Refer to the <i>VisualAge Smalltalk User’s Guide</i> .
	Text that the product displays	The status area displays <i>Category: Data Entry</i> .
	VisualAge programming objects, such as <i>attributes, actions, events, composite parts, and script names</i>	Connect the window’s <i>aboutToOpenWidget</i> event to the <i>initializeWhereClause</i> script.
Monospace font	VisualAge scripts and other examples of Smalltalk code	<pre>doSomething   aNumber aString   aNumber := 5 * 10. aString := 'abc'.</pre>
	Text you can enter	For the customer name, type John Doe

---

## Tell us what you think

Please take a few moments to tell us what you think about this book. The only way for us to know if you are satisfied with our books or if we can improve their quality is through feedback from customers like you. There is an online reader's comment form on the VisualAge Smalltalk web page.

---

## What's new

The Ultra Light Client feature and this guide have changed in Version 5.5.2. This chapter includes descriptions of these changes.

---

### What's new in ULC for Version 5.5.2

The ULC feature of VisualAge Smalltalk Version 5.5.2 has changed since Version 5.5.1. VisualAge Smalltalk Version 5.5 and Version 5.5.1 supported ULC 3.0. VisualAge Smalltalk Version 5.5.2 supports ULC 3.1. Here is a list of what is new for ULC 3.1 in Version 5.5.2:

#### UI Engine

- The UI Engine supports JRE 1.2.2 and JRE 1.3. Previous releases of the JRE are no longer supported.
- The API for requesting UI Engine properties has been enhanced by the following methods which will be executed synchronously:
  - `getUIAvailableFontFamilyNames`
  - `getUIScreenHeight`
  - `getUIScreenResolution`
  - `getUIScreenWidth`
  - `getUISystemProperty`
  - `getUISystemProperties`
- Column renderers now honor the following properties (where available):
  - `backgroundColor`
  - `foregroundColor`
  - `horizontalAlignment`
  - `verticalAlignment`
  - `font`
- All ULC enabler parts i.e. subclasses of *UlcEnablerAbstract* (ULCEnabler for ULC Java) now offer the setter property. This property holds the name (String) of a boolean setter method. If this property is set (i.e. not nil) the enabler will not send `setEnabled(state)` to its targets but will dynamically invoke the specified setter method with the current state as parameter on them. This is done using the Java Reflection API.

**Note:** Normally you only specify the setter property on terminal enablers.

- The visible content of an ULC shell can now be printed via the `printScreen` API.
- The UI started in client mode can now also be requested to auto-reconnect by specifying the `-autoReconnect` parameter when starting the UI.
- `UITree` and `UITreeTable` have been improved.
- The date validation now supports the specified format (as described by Java's `SimpleDateFormat` class) correctly but will no longer filter the input (i.e. all characters can be entered and validation is done when the field is left).

## ULC Smalltalk API

- A dialog box will be displayed when the connection to the UI gets lost if the class method *getConnDownErrorMsg* is implemented in the class which is registered in the ULC system (see *registerInUlcSystem*).

**Note:** When using IIOP connection the dialog box will not appear until the next event will be sent to the ULC server.

- The profiler API has been simplified by removing the enabled/disabled state. The API is now simply start/stop.
- An optional feature enables ULC applications started in server mode to keep the contexts (i.e. state) if a connection fails. This allows the UI Engine (started in client mode) to reconnect to those contexts (which would normally have been terminated). See ULC Client Reconnect feature for more details.
- A ProgramStarter part has been added which allows to execute programs on the UI side
- *ulcWhenLookAndFeelReceivedSend*: has been deprecated and should no longer be used. As of R3.1 *getLook* directly returns the result.

---

## What's new in ULC for Version 5.5 and Version 5.5.1

The ULC feature of VisualAge Smalltalk Version 5.5 changed from Version 5.0. VisualAge Smalltalk Version 5.0 supported ULC 1.5. VisualAge Smalltalk Version 5.5 supported ULC 3.0. Here is a list of what was new:

### UI Engine

- FormModels now preload all data for their clients reducing the number of round trips required to populate the view.
- The UI Engine now supports Swing 1.1.1 and JRE 1.2.2.
- Most widgets now support a BlockInputTrigger action. This trigger displays the busy cursor and blocks further user input till an answer has been received from the ULC server.
- The Table widget supports edit navigation using the keyboard.
- The Date validator allows multiple date formats (eg: dd:MM;dd:MM:yy;dd:MM:yyyy) which allows partial inputs of dates to be correctly interpreted.
- A new ULCTreeTable widget has been added as well as a corresponding ULCTreeTableModel.
- A new ULCContainer widget has been added.
- Labels now support multi line text.
- The ULCShell now supports a destroy on close flag that causes the shell to be released from the UI when it is closed.
- FormModels now preload all data for their clients reducing the number of round trips required to populate the view.
- Concatenated enablers are supported allowing simple boolean logic (AND, OR, NOT, XOR) to be used directly in the UI to enable or disable a widget. Wherever an enabler was supported a concatenated enabler can be used in its place and each concatenated enabler can be linked to other concatenated enablers and multiple widgets as input sources simultaneously.

- Mandatory input support allowing a widget to be highlighted whenever input is mandatory. The Mandatory input enabler also functions as an AND Enabler allowing for example a button to be enabled once all the source widgets have input.
- Table and Column supports a sortRequest API.
- The ULCIcon now supports a isCachedOnUI property and can now be cached in the UI persistently across sessions.

**Note:** The caching support is currently not available when the UI is running as an Applet.

- RowModel and TreeRowModel proxy have been added allowing any Form widget to access its data from a row in a TableModel or a TreeModel.
- Menus can now be linked to a Form model allowing the menu to be displayed when the form model has a specific value. This allows dynamic context menus to be supported.
- Batching of requests to the UI is supported using the startBatchingRequests/stopBatchingRequests API.
- Table, List and ComboBox support an ItemList API that allows multiple widgets to share the same TableModel but have different sort orders.
- The Column allows a header renderer to be specified.
- A HelpBrowserContext proxy has been added to provide simple F1/Help key support for all widgets.
- TableModel and TreeModel support a preloadAttributes API that forces the specified attributes to be uploaded to the UI whenever the row is uploaded for the first time.
- All widgets that support tooltips now support multi-line tooltips.
- All widgets that support mnemonics now support automatic mnemonic parsing from the label string (eg: ~File indicates the mnemonic F).
- Label supports the horizontalTextPosition attribute.
- The ComboBox is now closed whenever a mouse click is made outside the ComboBox popup area.
- The Shell supports a defaultButton API that specifies a Button that is triggered whenever the user hits enter within a Shell.
- The Date validator rolls partially specified years into the 20th century or 21st century based on a 70 year window using standard Y2K rules. (eg: 00= 2000, 98=1998)
- The CheckBoxMenuItem and the RadioButtonMenuItem can now be linked to a ButtonGroup.
- The Page supports a setIcon API that allows the notebook page icon to be changed dynamically. Table and List now display an empty string on receiving a null value instead of "null".
- Table Columns now support validation at the keystroke level similar to fields as well as maintaining the last input for an editable column.
- The StringValidator allows -1 as a string length indicating unlimited length.
- The TreeModel supports inserting children before or after a specific node.
- The TreeModel supports changing the icon or label of a node dynamically.

- The ULCTransport.properties file has been moved into the ulc.jar file as well as the uiengine.jar. Customers that implement their own transports can override the default file by placing their modified version of the ULCTransport.properties file in the root of their classpath. If such a file is found it will be used in place of the default file provided with the ULC release.

#### **ULC Smalltalk API**

- The NLS support for ULC has been enhanced to show the difference between local code page on the server and application strings (Unicode).
- A new UlcDataModel has been added that when used in conjunction with a ULCTreeItemList can be shared among Trees, Tables and List widgets.
- The IBM ULC VisualAge Server ULC feature is a prerequisite. (IBM ULC R1.5).
- The ULC Smalltalk API supports the ULC Filter API as well which allows the ULC Java UI Engine based record/playback tools to be used to test ULC Smalltalk applications as well.
- Support for ENVY/Stats has been added.
- Pagebooks signal the optional pageChanged event.

#### **Web Integration**

The UI Engine code is no longer packaged along with the HTTP Server. The environment variable ULCUIHOME must point to the root directory of the UI Engine.

---

## **What's new in this guide**

Based on the changes to the ULC feature, the Ultra Light Client Guide and Reference has also changed. Here is a summary of the changes to this guide:

- Updated the setup instructions for clarity and accuracy with the new version of ULC.
- Added documentation for the new widgets.
- Added documentation for the new enablers
- Added documentation for the TreeTable, Data Model and the Row Model.

---

## **Part 1. User's Guide**





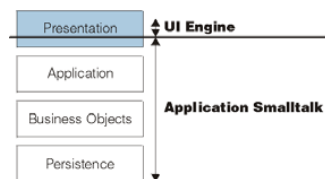
---

## Chapter 1. Overview of ULC

**Ultra Light Client (ULC)** supports the deployment of applications with very lightweight clients. It enables centralization of system maintenance and administration, decreasing the cost of desktop management.

ULC is designed to run applications on a centrally controlled application server. Only the presentation portion of an application is run on the end-user's desktop. Presentation is implemented as a universal user interface engine (**UI Engine**). Because no application-specific code is run on the client, there is no need for application-specific administration of the desktop machine.

An application communicates with the UI Engine through a high-level user interface protocol. This protocol is designed to scale down to low-bandwidth **thin-pipe** communication links. ULC shields you from this protocol; all distribution is handled behind the scenes.

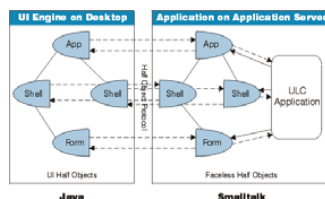


You develop ULC applications using conventional programming languages like Java and Smalltalk. Use of heterogeneous technologies (HTML, plug-ins, or JavaScript) is not required. This improves both robustness and ease of development. ULC uses communication infrastructures from Server Smalltalk: TCP or IIOP. For visually constructing ULC user interfaces, you use the Composition Editor.

---

### Half objects

Building ULC applications requires a special set of objects or widgets. These widgets have an API and functional behavior that is similar to that of "normal" widgets sets, but they completely lack a user interface. These widgets communicate with a corresponding "real" widget in the UI Engine by a socket-based mechanism. They act as a proxy to the real widget. Because every widget is split into an API and a user interface element, we call the widgets on both sides **half objects**. The application half is termed **faceless**.



Applications communicate with the UI Engine through the Half Object Protocol, which consists of requests, events, and callbacks. The application sends requests to the UI Engine. User interaction typically results in low-level events (for example, a mouse click) that are handled by the UI Engine first and then converted into a semantic event (for example, an *execute* action) that is passed back to the application. These events are typically used to synchronize the other half of the

object and then to trigger some application-specific action. If the UI Engine needs some data from the application's half object, it sends a callback. Callbacks are identical to normal requests, but their direction is reversed.

Half objects form a hierarchy. At the root is an Application object that provides methods for manipulating the global state of the application (for example, stopping) and maintaining a list of windows, or *shells*. A shell represents a top-level window with a content area and, possibly, a menu bar. The content area is a tree of composite widgets. Composite widgets form the inner nodes of the tree and implement the layout. Simple widgets are the leaves of the tree. The following illustration shows a part of the tree from the sample Dossier application:



Maintaining state across this half object split is a tricky business. You do not want to render the application unusable just because there was a communication problem or your client machine crashed. To prevent this, the UI Engine is conceptually stateless; that is, all state is kept in the application. Of course, some state is held in the UI Engine as well (for instance, the widget hierarchy), but only as a type of cache. It is always possible to clear that cache (for instance, by stopping and restarting the UI Engine) and reconnect to the application from the UI Engine.

This functionality has a major impact on communication between half objects. Methods of the faceless half objects typically modify some state on their half and then try to synchronize their half with the UI half. If the UI half is not available (for example, because the socket timed out or the UI Engine is down), synchronization cannot occur, but the faceless half remains in a consistent state. When the UI half becomes available once again, the faceless half sets the UI half to the correct state.

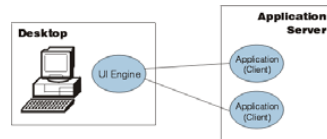
ULC is designed for thin-pipe connections, so network latency and network bandwidth influence some design decisions. Communication between half objects must be minimized, and requests should be batched together as a single message to avoid a sluggish user interface. ULC minimizes communication overhead by transmitting only presentation data that is visible (for example, just the visible 10 rows in a table with 10000 rows) or likely to become visible soon (for example, the next 10 rows in that table).

To address high-latency environments, communication between the application and the UI Engine is mostly asynchronous. For example, if the UI Engine has to draw a table, it requests the data for the visible part of the table from the application. The UI Engine does not wait for the requested data; it draws a placeholder instead. As a result, the UI Engine remains responsive through the wait. Depending on network latency and application responsiveness, the requested data asynchronously arrives later and replaces the placeholder data.

## How the UI Engine works

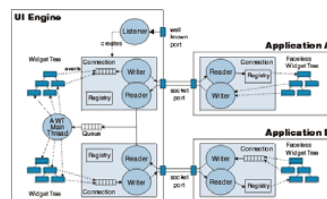
The UI Engine is implemented in Java; you can easily extend its behavior through subclassing. Adding new widgets is possible if you follow the guidelines described in “Chapter 7. Implementing ULC objects” on page 55. In the remainder of this section, we discuss the two ways the UI Engine can work in the deployment of your application.

### Test mode



In this case, the UI Engine is running as a multithreaded server process listening on a well known port; different applications connect to that server.

A listener thread accepts connections from applications and creates connection objects for each application client. A thread in the connection object receives requests from applications and posts them into an Abstract Windowing Toolkit (AWT) event queue. Events and callbacks returning from the UI Engine to the application are put into a write queue of the associated connection object and processed inside a separate writer thread (left side of the following illustration):

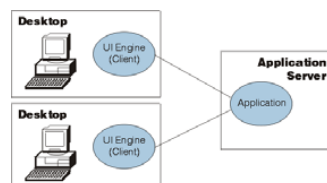


The AWT main thread reads and processes events from the AWT event queue. These requests either create objects that are registered for later reference in the connection's registry or call methods on already existing objects.

All manipulation of the widget tree is serialized, so explicit synchronization is not necessary. Because callbacks from the UI to the application never wait for results synchronously, blocking cannot occur in the main thread. Both design decisions simplify the architecture considerably and improve its robustness.

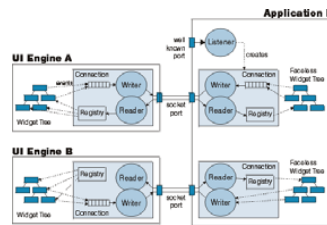
The architecture of the application side of ULC is similar to that of the UI Engine side. Every application has a connection object that handles asynchronous communication by means of two threads and maintains the faceless halves of objects in a registry (right side of the previous illustration).

### Production mode



In this case, the UI Engine and application switch roles: the UI Engine becomes the client and connects to the application, which runs as a server. In this case, the listener thread runs in the application server and accepts connections from UI

Engine components. For every connection, a new connection object is created that establishes an independent context for running the application.



This architecture is very similar to the one depicted previously. In fact, the connection objects are identical in both cases. The most significant difference is that in this case, the application code runs in parallel within a single address space. As a result, you must protect your data from synchronization problems arising from concurrent access. For more information, see “ULC and Server Smalltalk” on page 26.

---

## Chapter 2. Setting up ULC

This section provides an overview of ULC setup. For complete installation instructions, see the release notes shipped with the installation components listed below. You must install the VisualAge Smalltalk Server Workbench feature before setting up ULC.

The first step to installing and setting up ULC is determining what type of desktop setup you need. You can either install ULC as a development environment, as a standalone production environment, or as a Web production environment. The type of setup you need will determine which ULC components you need to install. For help in making setup decisions, see “What you need for ULC desktop workstations”.

ULC consists of the following components:

### **VisualAge features installable in the development environment**

These are subfeatures of the Server Workbench. You must load the following features:

- VA: ULC — Development
- VA: ULC — Examples
- ST: ULC — Server

To install these features, open the System Transcript window, select **Tools → Load/Unload Features**.

### **The ULC runtime support packages**

This includes the UI Engine. It is bundled in a self-extracting archive files (OS/2) or self-installer files (Windows), as follows:

- ULCOUie.exe, the UI Engine package for OS/2
- ULCWUie.exe, the UI Engine package for Windows

For installation guidelines, see the release notes for these files.

### **Java components and tools available from the Web**

For a list of Java resources that are appropriate for your chosen desktop environments, see “Setting up Java support” on page 9.

---

## What you need for ULC desktop workstations

ULC 3.1 requires one of the following platforms:

- Windows NT
- Windows 2000
- Windows 95 or 98
- OS/2 WARP 4.0

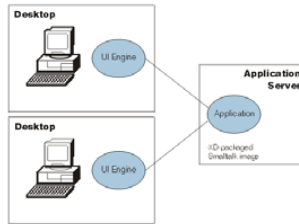
ULC 3.1 also requires the Java run-time environment (JRE) Version 1.2.2.

Setup of ULC components depends on how you intend end-users to run your ULC applications. In all of the scenarios that follow, the application server contains XD-packaged applications. For more information about these alternatives, see the following sections:

- “How the UI Engine works” on page 5

- “Using the UI Engine” on page 13

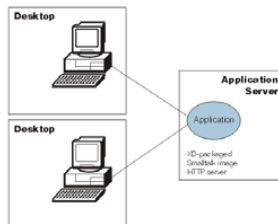
### Setup for running standalone applications in production mode



For running standalone applications, each desktop workstation must have the following installed:

- Java Runtime Environment (JRE) 1.2.2
- The UI Engine package

### Setup for running ULC applications over the Web in production mode



For running ULC applications over the Web, each desktop workstation must have the following installed:

- A Web browser enabled for the following:
  - JRE 1.2.2 (for Java 2 support) or later. Java 2 is recommended and the Swing class libraries are bundled inside. Download and install the Sun Java Runtime Environment (JRE) 1.2.2 from <http://java.sun.com/products/jdk/1.2/jre/download-windows.html>

**Note:** Ensure that the International version of the JRE is downloaded as all ULC samples require the internationalization support.

You also need an HTTP server running on the application server, Try the HTTP server shipped in the ULC samples package.

### Setup for developing applications

Each development workstation must have the following installed:

- JRE 1.2.2.
- The UI Engine package, installed in the ULC directory of your VisualAge Smalltalk installation. If this package is not installed in the correct location, the samples will not run.
- Java Development Kit (JDK) (for translation support only)

For development and early testing, all components can be set up on the same machine. You can use ULC Monitor or a similar tool to check the amount of data moving between client and server components.

When you are ready to move components to separate machines, you can use the Application Controller sample or a similar tool to remotely start and stop application servers from your test desktops.

Running applications in production mode from the development image requires some Smalltalk code and system setup. For more information on Smalltalk setup, see “Setting up a ULC development image to run in production mode” on page 51.

---

## Setting up Java support

For detailed installation instructions, see the release notes for the individual packages.

### Setting up support for running standalone applications

1. Install the JRE.
2. Set the ULCJREROOT environment variable for the location in which you installed the JRE:

```
SET ULCJREROOT=D:\JRE
```

This variable must be set. If the variable is not set the assumption is that a JRE installation is available on the client machine and the system path and classpath variables must point to this installation.

3. For Java 2, the ULCSWINGHOME environment variable is ignored.

### Setting up support for running applications over the Web

1. Install a Java-enabled browser.
2. Install the JRE.
3. Set the ULCJREROOT environment variable for the location in which you installed the JRE:

```
SET ULCJREROOT=D:\JRE
```

This variable must be set. If the variable is not set the assumption is that a JRE installation is available on the client machine and the system path and classpath variables must point to this installation.

4. For Java 2, the ULCSWINGHOME environment variable is ignored.

**Important:** If you installed Java Plug-In and notice unstable runtime behavior, turn off the JIT compiler from the Java Plug-In Control Panel.

### Setting up support for developing applications

1. Install the JRE.
2. Set the ULCJREROOT environment variable for the location in which you installed the JRE:

```
SET ULCJREROOT=D:\JRE
```

This variable must be set. If the variable is not set the assumption is that a JRE installation is available on the client machine and the system path and classpath variables must point to this installation.

3. For Java 2, the ULCSWINGHOME environment variable is ignored.
4. If you plan to internationalize your applications, install the JDK.

---

## Setting up the sample ULC HTTP server

The ULC HTTP server reads HTML pages and replaces specific tokens (instances of \$WEBHOST) in these pages to enable you to run ULC application servers using a Web browser.

Set the ULCUIHOME environment variable for the location in which you installed the UI Engine:

```
SET ULCUIHOME=D:\VAST\ULC\UIEngine
```

Ensure that the file ULCSamples\HttpServer\lib\HttpServer.properties points to the correct document root directory:

```
HttpServer.documentroot=d:/ulcrt/ULCSamples/HttpServer/lib
```

To see examples of the specially formatted web pages, unpack the file ULCSamples\HttpServer\lib\httpserver.jar and view the following files:  
/com/ibm/ulc/httpServer/resources/\*.uhtml

To start the HttpServer on the default port run the batch file:

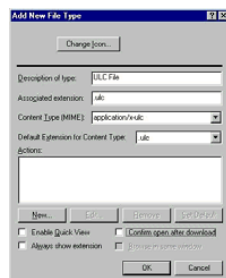
```
ULCSamples/HttpServer/bin/httpserver.bat
```

**Note:** When testing the Http server you must select either the ULC Java or the ULC Smalltalk link from the ULC Http Home page depending on the ULC environment being used.

## Configuring Internet Explorer for ULC

If using Internet Explorer 3.0 and 4.0, you must define a ULC file type in the Windows operating system. This configuration step is best done before the ULC HTTP server home page is loaded in the browser.

1. Start the Windows Explorer.
2. From the View menu, choose **Options**; a dialog appears. In this dialog, select the File Types page.
3. Click the **New Type** button; a dialog appears. Enter values into the dialog as shown here:



4. Associate this new type to an application by adding an **open** action. Click the **New** button under the **Actions** list; a dialog appears. Enter values into the dialog as shown here (the application can be chosen by clicking the **Browse** button):



5. Close all dialog windows by clicking **OK**.

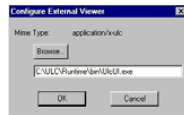


You are now ready to use the ULC HTTP server page for starting and controlling sample applications. You can also add links to start your own applications.

## Configuring Netscape Navigator for ULC

Netscape Navigator (3.0 and 4.0) and Communicator enable you to pick helper applications based on the MIME type of the file they receive.

1. Start Navigator and load the HTTP server home page (the server must be running; see “Using the sample ULC HTTP server” on page 15).
2. On the home page under **Running as Helper**, click the *Java Hello World* link; a dialog appears.
3. Click the **Pick App** button; a dialog appears. Pick the UlcUI.exe application in the ULC Runtime directory by using the **Browse** button as shown here:



4. Click **OK**; the association between the application/x-ulk MIME type and the UlcUI.exe application is registered with the browser (and the UI Server Console appears). You can check the association or change it later by going to the Applications category in Navigator's preference settings.

You are now ready to use the ULC HTTP Server page for starting and controlling sample applications. You can also add links to start your own applications.



---

## Chapter 3. Running ULC components

For convenience, ULC uses standard URL format to specify address, port, and application name in one argument. The following table summarizes how to start ULC components for *myApp*, an XD-packaged Smalltalk application:

	Test Mode	Production Mode
UI Engine (desktop)	ulcui.exe -server <port number>	ulcui.exe -url ulc://<host> :<port> /myApp
Application	myapp.exe -url ulc://<host> :<port>	myapp.exe -server <port number>

In the development image, sample applications can be started from the System Transcript window. First, start the UI Engine. Then, from the menu bar, select **ULC**.

Running your ULC-based applications from the development image requires handwritten code and system setup. For more information, see “Setting up a ULC development image to run in production mode” on page 51.

**Known problem:** Because of a Swing bug, the first window of an application can initially appear behind other windows.

---

### Using the UI Engine

You can use the UI Engine in any of the following ways, depending on the needs of your application’s end-users:

- Run the UI Engine as a standalone Java application
- Download and run the UI Engine as a Java applet using a Web browser
- Run the UI Engine as a helper application using a Web browser

#### Running the UI Engine as a standalone application

You can run the UI Engine as a standalone Java application with or without a Web browser. The UI Engine must be installed either on the desktop machine or on an accessible network drive.

To run the UI Engine without a Web browser, just start it from a command prompt set in the UIEngine\bin directory. For details, see “UI Engine command options” on page 14.

You can also launch and monitor applications through regular Web browsers. The entire process of launching a ULC application on a server machine, starting the UI Engine on the user’s machine, and controlling the started processes is accessible through standard Web-based mechanisms. The ULC HTTP server handles these tasks.

#### Running the UI Engine as an applet

The UI Engine can be run as a Java applet that connects back to an application server through a Web browser. The currently supported Web browsers for running the ULC UI Engine as an Applet are Netscape Communicator 4.x and Microsoft Internet Explorer 4 or 5. The browser must also have the Java Plug-in 1.2.2 installed.

You can verify that the browser opens the applet successfully by opening the Java Console. If your browser does not have the required Java support, the console will show a corresponding exception.

### Loading an HTML page containing the applet

A few sample HTML pages are included in the `httpserver.jar` file in the `ULCSamples\HttpServer\lib\` directory of the ULC samples package. (You must extract the samples from the `\com\ibm\ulc\httpServer\resources` subdirectory of the `httpserver.jar` file.) When one of these pages is loaded into a Web browser, it starts the applet directly. No Web server is needed. They enable you to start Application Controller as an applet in various forms: out of place, embedded, and embedded in a separate browser window.

When run as an applet, the UI Engine is less than 250KB, resulting in reasonable download times even over low-bandwidth connections. To avoid repeated connection overhead, the code is packaged in a JAR file (Java ARchive format).

## Running UI Engine as a helper application

You can run the UI Engine as a helper application for a configured `ulc` MIME type. To add the UI Engine as a helper application, follow the instructions listed for your browser:

- “Configuring Internet Explorer for ULC” on page 10
- “Configuring Netscape Navigator for ULC” on page 11

After you configure the UI Engine as a helper in Windows, clicking on any file with the extension `.ulc` causes the UI Engine to start and attempt to connect to the application referred to within the `.ulc` file. For example:

1. Create a file named `dossier.ulc`.
2. Within this file, enter the parameters you want to pass to the UI Engine, for example, `-m -url ulc://localhost:4444/Dossier`
3. Save this file.

If you now start the Dossier application as a server on port 4444 and then click on the `dossier.ulc` file, a connection is established to the Dossier server.

If the `dossier.ulc` file is referred to in a hyperlink on a Web page, clicking on the link starts a connection between the UI Engine and the ULC application server. In addition, if you use the ULC HTTP server, you can replace the host name with `$WEBHOST`, and the ULC HTTP server replaces the token with the name of the host on which the HTTP server is running.

## UI Engine command options

The `UlcUI` command starts the user interface engine of ULC with the JIT compiler enabled by default. To run with the JIT compiler disabled, use the `UlcUINoJit` command.

```
UlcUI
  [-m]
  [-url URL | -server port_number]
  [-locale ISOLanguage_ISOCountry]
  [-look Look&Feel_class]
```

The `UlcUI` program checks if the UI Engine is running and starts it if necessary. The engine can be started with different widget sets or with ULC Monitor, as appropriate.

The ULCUI program expects the directory and file structure in which it is embedded. Always start the program from its location in that tree (UIEngine\bin directory).

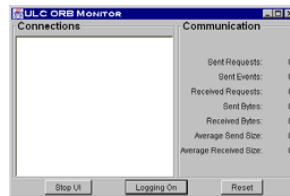
### Options

#### **-url** *URL*

Starts the UI Engine in production mode; that is, the UI Engine tries to connect to an application server with the given URL. The URL should conform to the following syntax:

`ulc://hostName[:portNumber]/[appName]`

#### **-m** Starts ULC Monitor, a simple tool for monitoring communication between the UI Engine and connected applications.



#### **-autoReconnect**

The UI started in client mode will retry to connect to the server for each connection that went down. This will normally start a new context except for ULC Smalltalk applications with enabled Client Reconnect feature. See “ULC Client Reconnect Feature” on page 61 for more details.

#### **-doublebuffering**

Toggles double buffering. By default, it is on.

#### **-locale** Overrides the default locale of the UI Engine. The format is *Language\_Country*, where language and country are the two-character ISO abbreviations, for example, `-locale en_US` for English language and the U.S.

#### **-look** *Look&Feel\_class*

Specifies the class name for the desired look and feel. Possible values are:

- `com.sun.java.swing.plaf.windows.WindowsLookAndFeel` (default)
- `com.sun.java.swing.plaf.jlfx.JLFXLookAndFeel`
- `com.sun.java.swing.plaf.motif.MotifLookAndFeel`
- `com.sun.java.swing.plaf.metal.MetalLookAndFeel`

#### **-server** *port\_number*

Starts the UI Engine in test mode.

---

## Using the sample ULC HTTP server

The sample ULC HTTP server can generate custom Web pages that enable a Web browser to start the UI Engine as an applet and then connect to a running ULC application. The HTTP server comes with all of the directory structure, Java files, and HTML pages required to run a set of sample ULC applications as applets. The setup also enables you to easily modify the sample Web pages to add your own applications and extensions.

This component is part of the ULC samples package shipped with VisualAge. To use it, follow these steps:

1. **Make sure that installation of the ULC HTTP server has been completed.** See “Setting up the sample ULC HTTP server” on page 10.

2. **Start a Web browser.** The HTTP server currently supports Navigator 3.0 and above, and Internet Explorer 3.0 and above.
3. **Start the HTTP server.** By default, the HTTP server listens on port 80 for HTTP requests. The server is generally run on the application server machine. You can start it from a DOS prompt or by double-clicking `ULCSamples\HttpServer\bin\httpserver.bat` from a file browser.
4. **Retrieve the ULC HTTP server home page.** You do this by following a URL to the HTTP server.
  - If you started the server locally, this URL is `http://localhost`.
  - If the server was started on a server machine (named *serverhost*), the address is `http://serverhost`.

**Note:** The sample ULC HTTP server is shipped for both VisualAge Smalltalk and VisualAge for Java. You must select the ULC Smalltalk link from the ULC HTTP home page.

5. **Connect to an application by doing one of the following:**
  - Connect to a running Application Controller, which can start and stop ULC application servers and request the UI Engine to make connections to them. In this case, the UI Engine runs as an applet (see “Running the UI Engine as an applet” on page 13).  
Select the **Connect** button on the Application Controller.
  - Connect using a custom Web page, using the UI Engine as a helper application. In this case, you must start each application server manually before selecting the corresponding link.  
Click the appropriate application icon on the custom Web page.

**Note:** If the samples do not run as an applet, open the Java Console window of your browser and check for security exceptions. This is commonly caused when the DNS name of your machine does not match the name used to make the connection to the HTTP server.

Using the ULC HTTP server, you can create pages to connect to application servers running on the same machine as the Web server without having to explicitly code the host name into each web page. The HTTP server treats HTML pages with the extension *.shhtml* in a special way: if the HTTP server is asked to serve one of these pages up, it searches within the page for all occurrences of the `$WEBHOST` token and replaces it with the host name of the machine on which it is running.

For example, suppose the Web page to connect as an applet to a running ULC Application Controller contains the following:

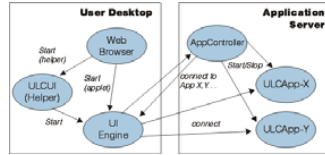
```
<APPLET archive="ulcui.jar"
         code="UIApplet.UIApplet.class"
         width="300" height="70">
<PARAM name="url" value="ulc://$WEBHOST:2222/AppController">
<PARAM name="title" value="ULC AppController Applet">
</APPLET>
```

When the file is served up by the ULC HTTP server, the Web browser starts the UI Engine as an applet and attempts to connect to the Application Controller running on the same machine as the Web server at port 2222.

For examples of the specially formatted web pages, unpack  
`<ulcrt>ULCSamples/HttpServer/lib/httpserver.jar` and look at  
`<ulcrt>ULCSamples/HttpServer/lib/com/ibm/ulc/httpServer/resources/*.uhtml`.

## Using Application Controller

Application Controller enables you to start and stop ULC application servers, something useful during development. Once these application servers have been started, Application Controller enables you to establish connections between them and the UI Engine.



Use Application Controller as follows:

1. If you have not done so already, unpack the samples in the `ulc\ULCSamples` directory.
2. **Start Application Controller in production mode (as server).** On a server machine (named *serverhost*), bring up a command prompt and switch to the `ULCSamples` directory. Start Application Controller on the selected port (2222) by entering the following:

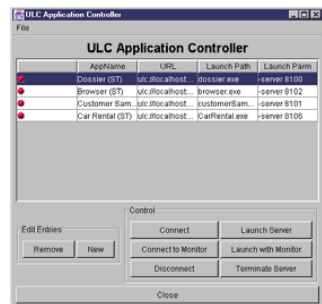
```
AppController2222
```

**Note:** 2222 is the default port for Application Controller, and the Java Plug-In pages expect Application Controller to be running on this port.

3. **Start the UI Engine in production mode (as client).** On a desktop workstation, bring up a command prompt and switch to the `UIEngine\bin` directory. Start the UI Engine and make it connect to Application Controller by entering the following:

```
run -url ulc://serverhost:2222/appcontroller
```

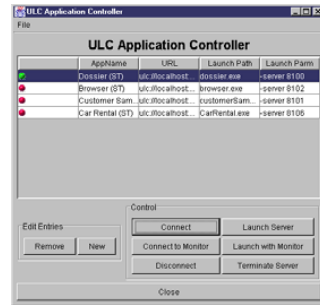
Application Controller opens as follows:



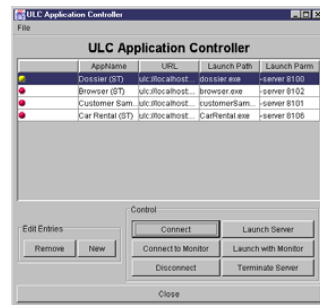
From here, you can do the following:

- To start an application server, select the application from the table and then select **Launch Server**.
- To open a connection to a running application server, select the application from the table and then select **Connect**.

When a connection to an application server can be established, Application Controller indicates this by turning the red-light icon to a green light, as follows:



Connections can also be pending, that is, started in the UI Engine but not yet established because the server is not responding. Pending connections are indicated by a yellow light, as follows:



- To close a connection, select the application from the table and then select **Disconnect**.
- To stop an application server, select the application from the table and then select **Terminate Server**.



---

## Chapter 4. About building ULC applications

This section describes the different types of ULC widgets and addresses application development topics common to both handwritten and visually constructed ULC applications, as follows:

- “ULC and Server Smalltalk” on page 26
- “Enabling national language support in ULC applications” on page 30

For more information about individual widgets, see the “Part 2. Programmer’s Reference” on page 85 or browse class information online.

For examples of visual composition using ULC widgets, start with “Building the To-Do List with ULC” on page 36.

---

### How ULC compares with common widget protocol

The most significant difference between ULC and CW is in its handling of system events. You register interest in a ULC event by sending a message to the widget that would signal the event. CW callbacks are not used.

For example, to register for the *action* event of a *UlcButton* instance, send the message *#ulcWhenActionSend: aDirectedMessage* to the button. At run time, *aDirectedMessage* is sent whenever the *action* event occurs. If the event has additional objects associated with it, *nil* parms of *aDirectedMessage* are replaced with those associated objects, in sequence, when the event occurs.

All event registration methods implemented in a given widget are categorized under *ULC-API-Events*.

#### Other differences from CW

- **Widget layout.** Manually laying out a ULC window is a lot simpler than constructing the same window under Common Widgets (see “Using layout widgets” on page 23).
- **Setting ULC widget attributes.** The two set methods available for all widget attributes differ in how they update the UI Engine. The standard setter (that is, *#attribute: anObject*) only sets the attribute and does not trigger an update of the UI. Use this standard setter for initialization only. The ULC specific setter (that is, *#setAttribute: anObject*) does update the UI. It is not possible to set, as a bundle, attributes of a widget already created on the UI and send the changes collectively to the UI. With the *UlcProxy>>#bundleRequestsWhile:* API, it is now possible to set multiple attributes of a widget already created on the UI and send the changes collectively to the UI.

---

### ULC class overview

ULC classes fall into the following five categories:

1. **Resources** are not user interface elements themselves but are used to configure these elements and therefore should live in the UI Engine as well as the application. Examples are fonts, bitmaps, images, and cursors.
2. **Widgets** are all kinds of user interface elements ranging from simple ones (like buttons, labels, editable fields, menus, and menu items) to more complex ones (like one-dimensional scrolling lists and two-dimensional tables).

3. **Layout** widgets are composites that implement a specific layout policy for their children.
4. **Shells** are the top-level widgets forming the root of every widget tree. A shell is typically represented as a modal or non-modal window and optionally has a menu bar. The shell controls the collaboration of these elements. Examples are standard shells, dialogs, and alerts.
5. **Models** are classes that can be used as data structures for model-enabled widgets. Model classes correspond to the models of the MVC paradigm; widgets represent the view and controller components.

---

## The simple widgets

Simple widgets include the following:

- Resources: *UlcIcon*, *UlcFont*
- Widgets: Label, Button, Entry Field, and so forth
- Menus: Menu, Menubar, Menu Item, Menu Separator

To reduce the number of round-trips between the UI Engine and the application, ULC provides built-in mechanisms for the following behavior:

- **Enabling and disabling.** Special conditions on some widgets can be used to enable or disable other widgets without any communication between the UI Engine and application. Examples are empty/non-empty fields or empty/non-empty selections in list and table widgets.
- **Validation.** Predefined validator and formatter objects can be attached to some widgets in order to perform validation like range checking and syntax checking without any communication between the UI Engine and the application. This is especially useful in form-model based widgets, where changes are not transmitted immediately but batched.
- **Management of event signaling.** Signals are sent from the UI to the application server only for those events for which interest has been registered.

---

## ULC layout

Because the implementation of the UI Engine varies with operating environment, specifying widget layout such that the result is visually appealing can be a challenge. Explicit placement of widgets does not work very well in this situation because widget sizes are not known in advance. Moreover, the precise horizontal and vertical alignment of widgets and the specification of resize behavior is a tedious task even with the help of the Composition Editor. This led to the integration of layout management in ULC, based on a hierarchical and high-level layout description rather than on the explicit placement of widgets, which handles resize behavior automatically.

The ULC layout widgets employ three basic layout types:

- **Border** draws a borderline or leaves a margin around a single widget. This layout is used by GroupBox, Shell, and Page.
- **Box** aligns widgets in a row/column fashion. This layout is used by Box, Horizontal Box, and Vertical Box.
- **Pile** stacks widgets on top of each other. Only the topmost widget is visible. This layout is used by Notebook and Pagebook.

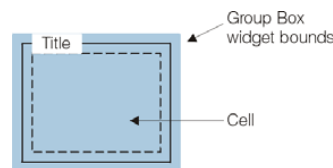
All widgets that use the first two layout types incorporate the notions of **cell** and **cell alignment**. A cell is a space which can be empty or can accommodate a single widget. The size of a cell is always equal to or larger than the minimum size of the

enclosed widget (if any). If the cell is larger than the minimum size, the cell alignment specifies what to do with the extra space. Possible options are to expand the widget until it fills the cell completely or to align the widget within its cell. The following illustration shows all possible cell alignments:

Alignment	Left	Center	Right	Expand
Top				
Center				
Bottom				
Expand				

### **Border layout**

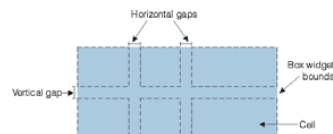
The simplest example for a cell is the GroupBox widget. GroupBox has a single cell and draws a border line and title around it. The size of the cell is determined based on the natural size of the enclosed widget. If GroupBox is made larger, the alignment is used to determine how to align the content widget.



Shell and Page also use Border layout to create a margin around their content areas.

### **Box layout**

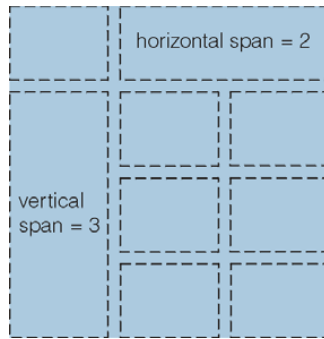
Box lays out cells in a two-dimensional grid separated by horizontal and vertical gaps. The width of a column is determined by the maximum width of all widgets in the corresponding column. The height of a row is determined by the maximum height of all widgets in that row.



If the box is made larger than its minimal size, any extra space is distributed evenly among expandable rows or columns. A row or column is expandable if at least one cell attribute in that row (or column) is set to expand. If expandable rows or columns do not exist, the box is centered within its cell bounds.

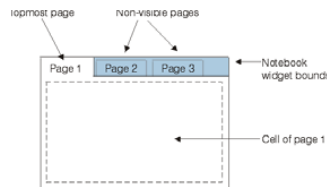
Box cells can span multiple columns or rows. In this case, the cell alignment applies just as for the spanning cell. If the spanning specification results in

overlapping cells, the (visual) result is undefined.



### Pile layout

Notebook piles tabbed Page widgets on top of each other and shows only the topmost page. By selecting a tab, users can switch between pages interactively. The size of Notebook is determined by the maximum size of all pages.



Like Notebook, Pagebook piles Page widgets, but Pagebook pages have no tabs. Page switching is completely under program control. Pagebook is used to implement dynamic layout, that is, to switch between different widget subtrees programmatically.

## ULC layout design tips

ULC layout is based on a few fundamental concepts. Combining layout types recursively enables not only sophisticated layouts but also automatic resize behavior. Many ways to solve a given layout problem exist, and it is not always obvious which approach is the most appropriate. The following paragraphs outline an overall strategy for designing and implementing a layout using ULC. In addition, we provide guidelines for cases with alternative solutions.

**Use nested boxes.** You cannot implement layouts by explicit positioning and resizing of widgets. Instead, design and build a tree of one- or two-dimensional boxes. We believe that box layout is easiest to use; therefore, layouts based on Box are much easier to understand and maintain across different desktop environments.

**Plan and design the structure of your layout in advance.** Even when using the Composition Editor, layout design is not just painting dialog controls on a drawing surface. For almost every layout, a resizing strategy has to be defined: Which elements should grow if the containing window is enlarged? What happens in a localized version of the layout when a supported language has different string length requirements? Another requirement is aesthetics: How do we keep controls nicely grouped and aligned and spaced not too far apart, even if everything resizes drastically?



**Start from top to bottom.** Identify the top-level elements or groups of elements. Decide whether to use a horizontal, vertical or two-dimensional box layout. Sometimes, the best solution is not as simple as it might seem. For example, is the top-level layout of the previous illustration a Vertical Box with two rows, each containing a Horizontal Box (following, right), or is it a 2x2 Box (following, left)?



Sometimes an answer can be found by asking whether elements should be kept aligned across horizontal rows, for example, whether the label *Dossier* should line up with the find entry field. If yes, a 2x2 Box has to be used. If not (which appears to be more realistic in this case), it is better to nest independent horizontal boxes within one Vertical Box.

**Avoid deep nesting.** Always try to keep your layout structures simple. The most important way to achieve this goal is by avoiding too-deeply nested structures. Using two-dimensional boxes instead of nesting boxes simplifies the layout. This goal sometimes conflicts with making the layout visually appealing, because the two-dimensional box forces elements into a rigid grid, aligning elements that need not or should not align. Next, we describe how spanning makes the use of two-dimensional boxes much more flexible.

**Use spanning.** With spanning, you can merge adjacent cells to form larger cells. This is useful if you want to align components with different size requirements within an overall grid. Without spanning, the width or height of rows and columns is determined by the widest or tallest element. With spanning, exceptionally large elements can grow into neighboring cells without making their containing rows or columns too wide or tall.

One way to implement the following example is to nest a 1x3 Box inside a 3x2 Box. With spanning, it is possible to employ a 3x4 box and avoid nesting altogether.

Spanning is a very powerful mechanism. You can use a grid layout, which improves the overall aesthetics because elements are aligned and their spacing is more uniform. On the other hand, you are not forced to make everything the same size, because you can span elements across multiple grid cells. Spanning makes it easy to have elements with different sizes always use multiples of some base cell size, which is visually more appealing than having elements with various unrelated sizes.

**Build the nested grid first; fill in the details later.** Because constructing Box hierarchies is simpler than rearranging them, build and test the complete box layout before filling in and adjusting the settings of all the non-layout widgets (labels, buttons, and so on).

## Using layout widgets

ULC provides the following layout widgets:

- Box
- Filler
- GroupBox
- Notebook
- Pagebook

## Example

We used layout widgets to create the following:

Name	Rupal	Mahmud
Address	Postfach 31	8045 Zurich
Country	Switzerland	

```
(UlcBox rows: 3 columns: 4)
"First row: Name-label + FirstName + LastName"
add: (UlcLabel new label: 'Name'; yourself);
add: (UlcField new columns: 10; ulcName: 'firstName'; yourself);
add: (UlcField new columns: 10; ulcName: 'lastName'; yourself);
add: UlcFiller new;

"Second row: Address-label + Street + ZipCode + City"
add: (UlcLabel new label: 'Address'; yourself);
add: (UlcField new columns: 4; ulcName: 'street'; yourself);
add: (UlcField new columns: 4; ulcName: 'zipCode'; yourself);
add: (UlcField new columns: 6; ulcName: 'city'; yourself);

"Third row: Country-Label + Country"
add: (UlcLabel new label: 'Country'; yourself);
add: (UlcField new columns: 10; ulcName: 'country'; yourself);
skip: 2;
add: UlcFiller new;
yourself
```

---

## Shells

Shells are the second-highest level widgets, representing the root of every widget tree. The topmost node of any widget tree is an instance of *UlcApplication*. A shell can be a modal or nonmodal window and can have a menu bar.

---

## About model-based widgets

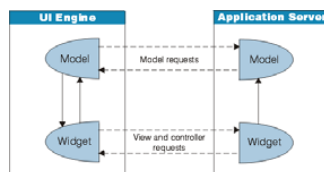
Most ULC widgets support two types of model-widget relationship.

In the first type (subclasses of *UlcFormComponent*), every widget has a built-in model and provides an API for accessing and changing the model:



For example, a field has a text model and methods *setText:* and *getText*. In addition, the widget provides methods for specifying the visual appearance of the user interface element, for example, what font to use or how many characters to display.

The second type uses an external model that is a separate object; this model can be shared between different widgets:



With this type of model-widget relationship, the application creates and configures both model and widget and then updates only the model. To retrieve changed data, the application asks the model for it, not the widget.

An example of this correspondence is the table widget and table model pair. The table model implements a two-dimensional data structure with rows and columns. The UI half of the object contains a cache so that data access requests of the table widget can be fulfilled immediately. If the cache does not contain valid data, the table model returns placeholders and asynchronously requests new data from the application half object. When this data arrives, the dependent widgets are notified and updated.

Synchronization of the two halves of the table model is configurable. Its notification-policy property gives you fine-grained control over when updates are sent from the UI to the application.

The form model is another example of the model-based API. A form model represents a set of named and typed attributes, similar to a heterogeneous dictionary. Most simple ULC widgets can be initialized with a form model and attribute name. Fields can be used on string attributes, check-boxes on boolean attributes, and groups of radio buttons on enumeration-type attributes. These widgets ignore their built-in model but track changes and allow updates of the named attributes of the specified form model.

As with the table model, the form model's notification-policy property gives you fine-grained control over when updates are sent from the UI to the application.

---

## Using models with model-based widgets

ULC provides five kinds of models:

- Form Model
- Table Model
- Tree Model
- Row Model
- Data Model

The dynamic environment of Smalltalk enables the generic implementation of all three of these models. *UlcModel* serves as data accessor for the UI. All widgets that can take their values from *UlcModel* have a string attribute called *(form)AttributeName*. This string is the attribute name of the domain model instance behind *UlcModel*. For *UlcModel* to function, all domain models must implement set and get methods conforming to standard Smalltalk conventions. When the UI requests a certain value from *UlcModel*, *formAttributeName* is passed in. *UlcModel* sends this string to its domain model as a message and answers whatever the domain model answers.

The implementation of *UlcModel* in Smalltalk saves the developer from having to implement the mapping of *formAttributeName* to domain attributes. To use *UlcModel*, do the following:

- Create one instance of *UlcModel* per domain object
- Set the correct attribute names as properties in the widgets that are supposed to display these attributes
- Associate the widgets with the *UlcModel* instance

It is always possible to follow the Java approach and create the necessary adapters as subclasses of the appropriate *UlcModel* subclass. The explicit message-sends in the adapter would eliminate potential problems that might occur at packaging time. All get and set methods used in *UlcModel* adapters must be either explicitly



included in the packaging instructions or referenced (that is, sent) in code that gets packaged (see “About model-based widgets” on page 24 for more details).

If a widget uses a form model, it will not receive widget-specific events. This is because the form model handles all changes to its widgets on the UI side to minimize traffic. Widgets that must trigger special events on the application-server side should not be linked to a form model.

Form Models now preload all data for their clients reducing the number of round trips required to populate the view.

For usage examples, see “Using ULC nonvisual parts” on page 40.

---

## ULC and Server Smalltalk

In ULC-packaged images, code runs in background processes that are owned and managed by Server Smalltalk (SST). Messages, both in- and outbound, are processed in *UlcRequestProcessor* objects that are owned by an instance of either *UlcApplication* or *UlcContext*. Messages processed by *UlcContext* handle initialization and context-specific settings like look-and-feel or local information; they are internal to ULC. Application-relevant code always runs in a process associated with an instance of *UlcApplication*. For each instance of *UlcApplication*, no more than one request is being processed at any given moment.

To identify the object for which the current Smalltalk process is running, send the *#ulcActiveProcessOwner* message. Similarly, the messages *#ulcActiveContext* and *#ulcActiveApplication* answer the appropriate object or nil. You can assume the following based on the responses to these messages:

- Processes that have an active *UlcContext* object but no *UlcApplication* are processing context-specific requests. In general, they are not executing business logic.
- Processes that have an active *UlcApplication* object also have an active *UlcContext* object.
- Processes that answer nil when sent the *#ulcActiveProcessOwner* message were created outside the ULC system (by business logic or by SST).

These relationships allow for custom exception handling for each *UlcContext* object. For more information, see “Customizing exception handling by context” on page 74.

### Concurrency issues

When writing server applications, you must address concurrency issues. When dealing with these issues, you must often identify the object space from which an object should be retrieved, based on the active connection or user. ULC allows for but does not directly support the creation of object spaces. In most cases, though, it is enough for the application to be able to identify the context under which it is running. As described previously, the *#ulcActiveContext* message will provide this information for all processes created or scheduled by ULC. It is up to you to make sure that a *UlcContext* is set for any processes created by business logic.

To set the owner for an active process, send the *Object>>#ulcSetUlcProcessOwnerTo: aUlcProxy while: aBlock* message. *aUlcProxy* must be either a *UlcContext* or *UlcApplication* object. *aBlock* contains the business logic that will be run in the



process to be owned by aUlcProxy. In the case where code in business logic has registered for context-related events in *UlcContext*, aUlcProxy must be set to the active *UlcApplication* instance.

Within one *UlcApplication* instance, only one request is active at any given time. The process that actually dispatches the request can be different each time, because ULC maintains a pool of processes to dispatch the requests.

### What does this mean for your application?

You must be conscious of this process model if *either* of the following is true:

- You are writing a server application, and data is shared among multiple views within the same instance of *UlcApplication* or *UlcContext*.
- Data is accessed by using background processes or threads, because in this case, replies are usually delivered asynchronously.

This process model is of no consequence if *all* of the following is true:

- You are not writing a server application.
- All data is shared across the image, in which case data access must be protected by critical blocks.
- All data is held by the *UlcView* class.

---

## Opening new ULC views

The following paragraphs refer to issues concerning SST process management. You can ignore this section if either of the following is true:

- The code that opens the view resides in a subclass of *UlcAppBldrPart*. This is usually the case if you are constructing the ULC application visually.
- The view will be opened as the result of UI input (for example, a button being clicked). In that case, ULC sets the currently active *UlcApplication* instance to be the view's process owner.

However, if the code that opens the view is part of the application's business logic, you have to be more careful. For more information about SST process management, read "ULC and Server Smalltalk" on page 26.

### Registering the start-up view of your application

You must register the first view of your application with the ULC system so that the correct class is given control as soon as the ULC environment is properly initialized. To do this, send the following message:

```
#registerApplicationNamed: <string_parameter> withStartupClass: MyStartupView
```

*string\_parameter* is a string that can be used as a command-line parameter. *MyStartupView* is the name of the class from which the application view will be instantiated. This new instance of *MyStartupView* is opened in a Smalltalk process whose owner is a fully initialized instance of *UlcApplication*.

### Opening additional views in running applications

After the startup view is open, you can open any additional views with this message:

```
MyOtherViewClass new openWidget
```

MyOtherViewClass is the name of the class that you want instantiated. The new view is opened in the instance of *UlcApplication* that is currently active. If you are not sure whether the active Smalltalk process is owned by your *UlcApplication* instance, send the *#evaluate: aBlock* message to the *UlcProcessOwnerToken* of your *UlcApplication* instance and open the new view inside aBlock.

### Opening a child view

To open a new view as a child of a view that is already open, send this message:  
`MyOtherViewClass new openWidgetAsChildOn: aViewAlreadyOpen`

Again, if you are not sure if the Smalltalk process is owned by your application, proceed as described previously.

---

## Handling Strings in ULC applications

For ULC applications, all strings are passed from and to the server as Unicode (UTF-8) strings. However, Smalltalk uses the local code page for processing strings. Before release 3.0, ULC processes all String data with the assumption that it is Unicode data. On ASCII platforms such as Windows (R), the English characters ('A-Z' among others) have the same codepoints both in ASCII and in Unicode. Therefore, most applications can send data in the local code page even though the ULC client processes that data as if it were Unicode data. For platforms that are not ASCII based, this does not work.

Before release 3.0, for platforms that are not ASCII based, hard coded English strings in the server could cause problems because the string data needs to be translated to Unicode before being transmitted to the client. Also, the strings coming back to the client need to be translated back to the platform codepage. This could cause characters to be lost because the client could be in a different locale than the server and the translation of certain characters cannot be made (This could occur even with ASCII platforms where the locale of the server and client did not match).

In release 3.0 ULC provides two new classes for processing Unicode data, *UlcUnicodeSBString* and *UlcUnicodeDBString*. All Strings and Labels generated by ULC and all string data stored in MPR files are converted to these new data types (see "Converting Strings to Unicode" on page 29). The application programmer needs to be aware that the UIEngine sends any string data back as Unicode strings. This can cause some problems with hardcoded strings because String does not equal an Unicode string and therefore will not compare as equal with data received from the client. (When writing your application, it is recommended that you use MPR files instead of hardcoded strings). There are two solutions to this problem:

- Send the *#ulcAsUnicodeString* method to an instance of a *#ulcAsUnicodeString* in order to convert it to Unicode. If you are receiving data from some other application or data source you may need to convert these strings to Unicode using the *#ulcAsUnicodeString* method.
- Convert the Unicode strings to your local code page by sending the *#ulcAsLocalString* method to an instance of a *UlcUnicodeSBString*

For ULC applications written before release 3.0, see "Running applications in Compatibility Mode" on page 29 and "Migrating to Unicode" on page 29.

## Running applications in Compatibility Mode

In order to support ULC Smalltalk applications written before Release 3.0, the ULC application can run in Compatibility mode. The application runs in compatibility mode when it pre-requisites the application *UlcNlsCompatibilityApp*. This mode, the Pre-R3.0 NLS Compatibility Mode, allows existing applications to work without any change. When using this mode, any strings coming out of the new MPR files are converted to Strings and any string data returning from the client is placed into String. For information on migrating the MPR files, see “Migrating MPR files”.

**Note:** When packaging applications, you must not mix different compatibility requirements. All applications in the packaged image must be in compatibility mode (or all applications in the packaged image must not be in compatibility mode).

## Migrating to Unicode

In this release, the MPR files are generated using Unicode strings (*UlcUnicodeSBStrings*). However, in previous releases, the MPR files were generated in Unicode of type *String*. Unless you are running in compatibility mode, you must migrate your application as follows:

1. Migrate your MPR files. See “Migrating MPR files”.
2. Migrate your application by one of the following ways:
  - Convert the hardcoded strings to Unicode strings. For information on converting strings to Unicode, see “Converting Strings to Unicode”.
  - Add code to convert the data received from Unicode strings into local code page strings.

**Note:** This alternative may cause you to lose data if your client and server are not using the same locale.

### Migrating MPR files

Migrate your MPR files by performing the following steps:

1. Select **ULC>Tools>NLS Tools>Migrate pre-R3.0 Unicode MPRs to new Unicode MPR format**.
2. Select which MPRs to migrate.

### Converting Strings to Unicode

If you are writing a new or migrating a previous application, use the following classes to ensure that the string is in Unicode format:

- *UlcUnicodeSBString*
- *UlcUnicodeDBString*

Use the following methods with the above Unicode classes:

*bindWith*

Replaces strings within the Unicode string. If the arguments are Strings, the Strings are converted to Unicode before being bound into the Unicode string.

*bindWith: with:*

Replaces strings within the Unicode string. If the arguments are Strings, the Strings are converted to Unicode before being bound into the Unicode string.

*ulcAsLocalString*

Converts the Unicode string to a String in the local codepage

*ulcAsUnicodeString*

Tests if the string is a Unicode string. If the string is not a Unicode string, converts the string to a Unicode string.

To convert Strings and DBStrings to Unicode, use the *#ulcAsUnicodeString* method.

---

## Enabling national language support in ULC applications

A well behaved server application must provide NLS support for every client individually. Because every client potentially has its own language, the default Smalltalk NLS mechanism has been extended for ULC.

ULC clients can operate on any platform that is supported by Java, the language in which the UI Engine component is implemented. Locale information can vary from platform to platform. VisualAge already provides a rich set of NLS tools; ULC leverages these by putting them to use in slightly different ways.

ULC clients deliver country and language information in ISO two-character abbreviated format. This results in a Locale object that represents the locale to be created within a ULC context associated with a given UI Engine client.

ULC uses the MPR file format provided by VisualAge Smalltalk as an external source for language-specific strings. All end-user tools (such as those listed in the System Transcript window's **Tools** menu) for managing the language resources can be used in ULC. (For details, see the *VisualAge User's Guide*.)

For support of multiple national environments, you have a choice as to how to create the MPR files:

- You can create native-encoded MPR files individually from native workstation environments. This parallels the standard national language support provided in VisualAge.
- You can create Unicode-encoded MPR files for all environments from a single workstation that represents your default language environment. This parallels the standard national language support provided by Java. This approach enables you to avoid multiple workstation setups for the purpose of translation support. However, you must download and install the JDK on this single workstation.

You must use Unicode encoding for those languages that are enabled but not shipped in VisualAge national language versions (for example, Russian). It is also a good idea to use Unicode for double-byte languages.

ULC implements a straight mapping of classes to MPR files. This approach is a simplification from VisualAge in that ULC does not use named NLS groups. Rather, ULC assumes that the resources of all classes defined in a manager application share one MPR file. This file is defined by the manager application class in the method *#abtExternalizedStringBuildingInfo*.

*UlcSystem* is configured by ULC applications with their language-to-file-name substitution information in an application class method (see "Implementing NLS support for ULC applications" on page 31).

Visually constructed views retrieve their view constants on an instance basis by accessing their contexts for the NLS group concerned. *UlcProxy* implements the *#getMRI:* method, which in turn sends the *UlcContext>>#getMRI:class:* message. This scheme allows for multiple contexts, each with its own language, while still using standard VisualAge NLS mechanisms. Applications wishing to make use of

this feature can do so by accessing the NLS group through the process owner. For more information about SST process management, see “ULC and Server Smalltalk” on page 26.

## Differences between ULC and standard VisualAge NLS mechanisms

- ULC does not support the unique combination of language and country (that is, English/US and English/UK). NLS language in ULC is country-unaware. If your application needs this kind of support, you can subclass *UlcAbstractLocalizer* for implementation. By default, ULC uses the *UlcDefaultLocalizer* class for mapping local information to MPR file names.
- If used by classes defined by different manager applications, indexed messages are duplicated and given different file names.
- If business objects need access to externalized strings, they must retrieve them from their context. To get the correct locale-specific resources, send the following messages:
  - *UlcContext*>>#*abtSeparatedConstantsFor: aClass* (for *UlcView* class-specific GUI strings)
  - *UlcContext*>>#*getMRI: anInteger for: aClass* (for an indexed message)

## Implementing NLS support for ULC applications

If you have chosen to create Unicode-encoded files, you must install the JDK first. You will use the JDK's **native2ascii** utility program to convert native encoding to Unicode. For more information about this utility program, see <http://www.javasoft.com/products/jdk/1.2/docs/tooldocs/win32/native2ascii.html>.

Next, follow these steps in VisualAge:

1. Implement the *#abtExternalizedStringBuildingInfo* method in the startup class for your ULC-based application, as follows:

```
SomeClass class>>#abtExternalizedStringBuildingInfo
    ^Array
      with: 'ULC?R40'  "MPR filename without extension; use yours here"
      with: false      "NOT platform dependent"
      with: true.       "DO recurse subApplications"
```

The question mark (?) in the file name is a placeholder for the locale strings mapped in *#initializeUlcSystemNLSMappings* later in this process.

2. From the System Transcript window, generate a TRA file for the default language environment. Select **Tools**→**NLS Tools**→**Generate TRA**.
3. Generate a MPR file for the default language environment. Select **Tools**→**NLS Tools**→**Generate Default Language MPR**.
4. Replicate and translate a TRA file for each additional language you want the application to support.
5. From a command prompt, convert the TRA files to Unicode with the **native2ascii** utility program found in the JDK.

**Important:** Skip this step if you do not wish to use Unicode or if the default language environment is Latin-1 (code page 850 or ISO 8559–1).

For example, the following syntax converts a Japanese input file to Unicode:

```
native2ascii -encoding Cp942 input.tra output.tra
```

6. From the System Transcript window, generate MPR files for environments other than the default:

- If you converted the TRA files to Unicode, select **Tools>NLS Tools>Generate language MPR from unicode TRA**.
  - Otherwise, from each native workstation, select **Tools>NLS Tools>Generate language MPR from current language TRA**.
7. Make sure to place the MPR files in a directory contained in your NLS path, or in the same directory as the image.
  8. Write the initialization script for locale mappings, as follows:

```
SomeClass class>>#initializeUlcSystemNLSMappings
```

```
UlcSystem default
nlsMapDefault: 'en' to: UlcNlsFileEnglish;
nlsMappings: (
  LookupTable new
    at: 'en' put: UlcNlsFileEnglish;
    at: 'de' put: UlcNlsFileGerman;
    yourself)
```

In this method, the ISO language code `de` is mapped to the string specified in `UlcNlsFileGerman`. This string is substituted into the name of the MPR file established for a German locale in the `#abtExternalizedStringBuildingInfo` method.

9. Call this method in *loaded*, a method in the startup class:

```
loaded
```

```
UlcSystem isRuntime ifTrue: [self initializeUlcSystemNLSMappings].
UlcSystem registerApplicationNamed: 'MyUlcApp' withStartupClass: SomeClass
```

For more information on registration during startup, see “Registering the ULC visual application class” on page 49.

*UlcAlert* is the only class shipped with ULC that uses string resources according to this scheme. The abstract MPR file name is `ulcwi?40`. Its language-to-file-name substitution mappings follow the standard values described in the *VisualAge User's Guide*. To implement a different mapping, adjust the definitions for the widget strings to match those of the application.

The language mappings shipped with ULC correspond to those languages for which a national language version of VisualAge is available, with the addition of German. For any other languages, you must include TRA and MPR files. The names and scripts for their generation can be found in the *UlcWidgetApp* class categorized under *ULC-Internal-NLS Support*.



---

## Chapter 5. Building ULC applications visually

Building an application with ULC parts follows the same process as building any VisualAge application. However, important differences exist, as follows:

**Connections.** Unlike standard VisualAge parts where the user interface is expected to run on the same machine as the application, ULC parts are designed so that user interface and application can be running on different machines connected through a network. ULC parts are therefore optimized to reduce the amount of network traffic. This approach requires discipline on the part of developers when making connections between widgets and parts to ensure that only necessary updates are sent across the network.

ULC parts do not signal events for state changes (no *self* event). As a result, attribute-to-attribute connections have limited utility; they run only once, sent as initialization messages in the `#ulcBuildInternals` method.

Use event connections to explicitly set the contents of widgets. Each ULC part has a well defined set of events that can be used to make event-to-action or event-to-script connections.

**Layout.** ULC parts use a different layout scheme than standard VisualAge parts do. You arrange ULC parts using a grid-based layout scheme. This scheme lays out widgets in a portable way independent of actual widget size. For more information about layout, see “ULC layout” on page 20.

**Converters.** ULC supports local validation and formatting of user input in the UI Engine. The ULC Field part supports the following ULC converters:

- String formatter
- Range validator
- Percent validator
- Date validator
- Regular expression validator

ULC converters substitute for the standard VisualAge converters, which would require round-trips between the UI Engine and the application server. Using ULC converters to validate and format data in the UI Engine avoids those extra round-trips. You can extend this set of ULC converters using a process similar to that described in “Chapter 7. Implementing ULC objects” on page 55.

Standard VisualAge converters should be used only when local validation in the UI Engine is not possible. You cannot do this visually; you must write a method to pass the object to be converted to the converter and then set the contents of the ULC widget part with the converted object.

**Enablers.** It is common practice in GUI development to enable or disable a widget based on the state of another part. ULC parts support enablers, whereby traffic between the UI Engine and the application server is not needed to enable or disable a widget.

An enabler informs its registered widgets that it has changed its state, and effects the change of the enable/disable state of these widgets. These widgets can serve as enablers: `UlcField`, `UlcFormModel`, `UlcTableModel`, `UlcList`, `UlcTable`, `UlcTree`,

UlcToggleButtons, etc.. For example, you can use the UlcTable as the enabler for a UlcButton. The UlcTable object will tell the button to change its state to enable whenever there is a selected item in the table.

Enablers can be concatenated with the help of these classes: *UlcAndEnabler*, *UlcNotEnabler*, *UlcOrEnabler*, and *UlcXorEnabler*.

As of ULC Release 3.1, all ULC enabler parts (i.e. subclasses of *UlcEnablerAbstract*) can not only be used to enable/disable a widget but can also toggle any boolean property of a widget by specifying the appropriate setter method name.

Enablers now offer the setter property. The setter property holds the name (string of a Boolean setter method. If the setter property is set, the enabler uses the Java Reflection API to dynamically invokes the specified setter method with the current state as a parameter. (The enabler does not send `setEnabled(state)` to its targets.)

**Note:** You normally only specify the setter property on terminal enablers

---

## Visual composition pitfalls in ULC

When working with ULC parts in the Composition Editor, pay close attention to these issues:

- Event names in ULC are pool references. Keep this in mind if you refer to events in handwritten code.
- ULC parts do not signal a *self* event.
- When signaling events in handwritten code, send the `#ulcSignalEvent:*` message, **not** `#signalEvent:*`. Registering for events requires the message `#ulcWhen*Send:aDirectedMessage`.
- Entry Field parts do not signal change events (*UlcEventValueChanged*) when they are connected to Form Model parts.

---

## Using ULC visual parts

When working with the ULC visual parts, you will work with three different kinds of parts:

- **Box parts** are used to define the overall layout. They define a grid of cells.
- **Cell parts** contain a single widget part and define how the widget is positioned inside its bounds. Cells keep track of the alignment of their widgets and not the widgets themselves. Cell parts are automatically created and deleted as needed. You interact with them only to change their settings.
- **Widget parts** represent buttons, entry fields, and so on.

## Defining layout

When you create a ULC container part like Shell, Page, or GroupBox, it contains a single cell. To define layout, insert a Box part, which defines the grid. Pick one of the following:

- **Box:** the general grid layout. Initially, a box part has two rows and two columns
- **Horizontal Box:** a convenience part initially set with a single row of two cells
- **Vertical Box:** a convenience part initially set with a single column of two cells

The layout of your part is then defined by either nesting box parts or by spanning the cells of a box part. See “ULC layout” on page 20 for a description of how to use boxes to define layouts. In a nested box and cells structure, you need to be



able to distinguish boxes from cells. To make this distinction more explicit, the bounds of a box are indicated by a solid blue line, and the bounds of a cell are indicated by a dashed line.

## Adding widgets

Add widgets by dropping them into cells. A cell can contain a single part. Once dropped into a cell, the part is immediately resized according to the cell's alignment.

To reserve some fixed amount of white space in a layout drop a Filler part into a cell. Filler has properties to define the minimum amount of white space it should occupy.

## Setting layout properties

A Box part has the following properties:

- *columns*: the number of columns
- *rows*: the number of rows
- *horizontalGap*: the horizontal gap between the cells
- *verticalGap*: the vertical gap between the cells
- *margin*: the margin around the cells.

A cell part has two properties to define the alignment of its contained part:

- *horizontalAlignment*: the horizontal alignment of the contained part: Expand, Center, Left, Right.
- *verticalAlignment*: the vertical alignment of the contained part: Expand, Top, Bottom, Center.

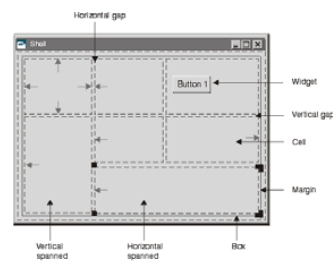
You can change the alignment property of a cell in the following ways:

- From the cell's settings
- From the **Align** submenu of the cell's pop-up menu. To change the alignment of multiple cells, select them and use the **Align** submenu.

Cells contained in a Box part have additional spanning properties, which you can use to define cells which span over other cells.

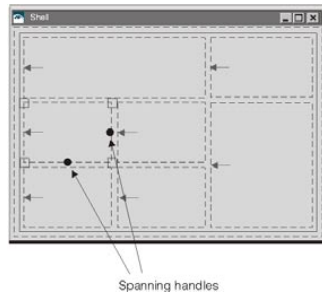
- *horizontalSpan*: specifies the number of cells this cell spans horizontally.
- *verticalSpan*: specifies the number of cells this cell spans vertically.

The following figure shows how the parts and properties are presented in the Composition Editor:



The figure above shows a box which defines as three-by-three grid. There is both a cell which spans vertically and a cell which spans horizontally.

The arrows in an empty cell indicate how it will align its contents. An arrow is shown at every attachment point. For example, the top left cell has the alignment set to expand in both dimensions. If there is no attachment point, the cell centers the part in the corresponding dimension. In addition to changing the spanning with the spanning properties, you can change it by direct manipulation with the blue **grid handles**. The grid handles are shown in the figure below. You can drag the grid handles either horizontally or vertically to span a cell.



## Changing the ULC layout grid

You can change the dimensions of the initially defined grid in one of the following ways:

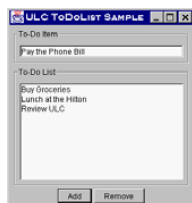
1. Change the number of rows and columns in the settings view of the Box part.
2. Add an individual row or column relative to a cell. You first select the cell part. The cell's pop-up menu provides a **Row** and **Column** submenu to add a row or column before or after the selected cell.
3. Delete a row or column. Select the cell in the row or column that you want to delete. Then select **Delete** from the **Row** or **Column** submenu.

---

## Building the To-Do List with ULC

This section guides you through building a sample ULC application. It is assumed that you are familiar with Smalltalk and the Composition Editor. For a quick overview of the Composition Editor, read *VisualAge Smalltalk: Getting Started*.

You will create a To-Do List application as shown below:



## Creating a ULC application

1. Start VisualAge.
2. From the VisualAge Organizer menu bar, select **Applications** and then **New**. A dialog appears.
3. Enter `MyUlcToDoListApp` and click **OK**.

## Creating a new ULC visual part

1. From the Organizer **Parts** menu, select **New**. A dialog appears.
2. In the **Part class** field, enter `MyUlcToDoListView`.

3. From the **Part type** list, select **ULC Visual Part**.
4. Ensure that the **Open now** check box is selected.
5. Click on **OK**.

The Composition Editor opens your newly created ULC visual part. It already contains a Shell part that contains a single cell. Change the text in the title bar of the Shell part to ULC ToDoList Sample by directly editing the text.



## Setting the layout

ULC visual parts use box layout. (For more information about layout, see “ULC layout” on page 20.) The To-Do list sample can be visualized as having three basic groups:



- To Do Item
- To Do List
- Buttons

We therefore choose a grid of a single column with three rows.

### Adding a Box part

1. From  the Ulc Canvas category, select  the Box part in the right-hand column and drop it onto the Shell part. By default, the Box has two rows and two columns, each containing a cell. The Box part can be distinguished from the cell by the Box’s continuous blue outline; the cell has a dotted black outline.
2. Edit the following Box settings:
  - Set *columns* to 1.
  - Set *rows* to 3.
3. Click **OK** to apply your changes.

### Adding GroupBox parts



1. Select  the GroupBox part and drop it into the first row. By default, a cell’s content is horizontally left-aligned and vertically centered.
2. Edit GroupBox settings as follows:
  - Set *horizontalAlignment* to **Expand**.
  - Set *label* to To Do Item
3. Click **OK** to apply your changes. The GroupBox part immediately expands to take up all space within this cell.
4. Select  another GroupBox part and drop it into the second row.
5. Edit GroupBox settings as follows:
  - Set *horizontalAlignment* to **Expand**.
  - Set *label* to To Do List
  - Set *verticalAlignment* to **Expand**.
6. Click **OK** to apply your changes. The GroupBox part immediately expands to take up all space within this cell.

## Adding the remaining parts

Next, add UI components to the layout you have just set up:

- An Entry Field part in the **To-Do Item** group box
- A List part in the **To-Do List** group box
- **Add** and **Remove** buttons

## Adding the Entry Field part

From  the Ulc Data Entry category, select  the Entry Field part and drop it into the **To-Do Item** group box.





## Adding the List part

1. From  the Ulc Lists category, select  the List part and drop it into the **To-Do List** group box.

Double-click on the List part to bring up its settings view. Look at the *attributeName* setting. This setting is used to retrieve a list-item object's string representation from the underlying collection part for display in the list. By default, this value is yourself. Because we are dealing only with strings, we can leave the default value as is.

## Adding the buttons

At this point, we need to add two buttons to the remaining empty cell. Because a cell can contain only a single widget, add a Box part to the cell and then add the button parts to the box, as follows:

1. Change the alignment of the cell to be horizontally and vertically centered.
2. From  the Ulc Canvas category, select  the Horizontal Box part, which creates a box layout with two columns by default. Drop it in the empty cell.
3. From  the Ulc Buttons category, select  the Button part; drop one onto each half of the horizontal box.
4. Edit settings for the Button parts as follows:
  - Set the left button's label to Add
  - Set the right button's label to Remove

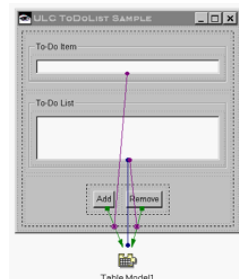
## Adding the Table Model part

To keep a list of to-do items, we need a nonvisual collection part. For ULC applications, use the Table Model part.

From  the Ulc Models category, select  a Table Model part and drop it on the free-form surface.

## Connecting the parts

When you have finished, your sample will look something like this:



## Making the attribute-to-attribute connection

To display the contents of the collection in the list, make an attribute-to-attribute connection between the Table Model part and the List part. The purpose of this connection is to initialize the List part. Whenever an item is added or removed

from the collection, the items displayed in the list are automatically updated. Select the *model* attribute of the List part and linking it to the *self* attribute of the Table Model part.

1. To start the connection, display the pop-up menu for the Table Model part and select **Connect**, then *self*. The mouse pointer changes to indicate that you are in the process of making a connection. If you accidentally started the wrong connection, use the Esc key to cancel it.
2. To complete the connection, click mouse button 1 on the List part. From the pop-up menu that appears, select *model*. A connection line is now displayed from the Table Model part to the List part.


**Important:** Use ULC attribute-to-attribute connections only to initialize part attributes. To keep ULC part values in sync, use event-to-action connections.

### Making the event-to-action connections

The To-Do List application is supposed to add the text entered in the entry field to the list when the **Add** button is clicked and to remove the item selected in the list when the **Remove** button is clicked. To make this happen, you must make event-to-action connections between the Button parts and the Table Model part. You make these connections to the Table Model part instead of to the List part because the Table Model part is the one that maintains both the items entered and the order in which they were added.

1. Connect the *action* event of the Add button part to the *addRow:* action of the Table Model part. A dashed line appears, which means that more information is necessary. In this case, the parameter for the *addRow:* action is missing.
2. To supply the parameter, connect the *value* attribute of the Entry Field part to the *aRow* attribute of the previous connection.
3. Connect the *action* event of the Remove button part to the *removeRow:* action of the Table Model part. A dashed line appears, which means that more information is necessary. In this case, the parameter for the *removeRow:* action is missing.
4. To supply the parameter, connect the *selectedItem* attribute of the List part to the *aRow* attribute of the previous connection.

## Testing the application

Select  the Test tool from the tool bar. All views opened with the Test tool use a named context. This makes debugging easier. The context can be inspected and reset from the **ULC** menu in the System Transcript window.

## Adding support for enablers

For the final polish, you can add enablers to your sample.

A common behavior in GUI applications is to enable or disable a button based on the state of another widget. ULC parts support this with an *enabler* attribute. To enable the **Add** button only when the entry field is not empty, add an attribute-to-attribute connection between the *enabler* attribute of the Add button part and the *self* attribute of the Entry Field part.

Similarly, to enable the **Remove** button only when an item has been selected from the list, add an attribute-to-attribute connection between the *enabler* attribute of the Remove button part and the *self* attribute of the List part.

Now test your application again to verify that the buttons are enabled and disabled properly.

Congratulations! Your To-Do List application is finished. Before versioning your application, read “Packaging ULC-based applications in XD” on page 49.

---

## Enabling reuse with ULC composite parts

Composite ULC parts are visual parts that can be stored in the library for later use in other windows. They are much like standard ULC visual parts except that their topmost widget is not a shell, but a cell-like widget that looks and feels much like a *FormView* from the standard Composition Editor. Building a composite part really means building a reusable ULC widget that can be used in any other ULC visual part (both in other composites and shells).

To create a ULC composite part from the VisualAge Organizer’s New Part window, simply choose **ULC Composite Visual Part** from the **Part type** list. You will see that the superclass of the part switches to *UlcCompositeView*. From there, building a ULC composite part is no different than building a ULC visual part. All features and parts that can be used in shell-based parts can also be used in composite parts. Any restrictions that apply to shells also apply to composite parts.

For an example, see the name example shown in “Working with Form Model parts” on page 41.

---

## Using ULC nonvisual parts

This section explains how to work with the nonvisual ULC parts, as follows:

- Variable
- Form Model
- Table Model
- Tree Model
- Data Model
- Row Model

### Working with Variable parts

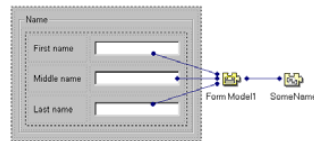
In ULC, Variable parts have work to do that standard VisualAge Variable parts do not. ULC Variable parts must be able to communicate with both ULC and standard Smalltalk classes, for example, pre-existing business domain objects. This task is complicated by the need for compile-time resolution of run-time UI/application connections under ULC. The *UlcVariable* class (which appears on the palette as the Variable part) does this largely without your having to be concerned with what is happening.

1. To use a Variable part in ULC, drop it on the free-form surface the way you would a standard VisualAge Variable part.
2. Set the type of the Variable from the part’s pop-up menu. From that point on, the Variable part will accept as valid values only those objects that inherit from the specified type.
  - To access the wrapped type from code, use the *#object / #object:* protocol.
  - To access the wrapped type visually, connect to *ulcVariableValue* or *setUlcVariableValue*. In addition, any features that are part of the type’s public interface will also appear on the connection menu for the variable.

The type of a ULC variable cannot be changed at run time. However, the contents can be changed at any time as long as the new value is of the correct type. `nil` is also a valid object for variables of any type.

## Working with Form Model parts

The Form Model part enables changes made in the UI to be deferred and sent to the application only when explicitly flushed. Form Model parts are typically used with standard business domain objects. The following example of a composite part shows the essentials:



The connections between the Entry Field parts and the Form Model part are all the same: between the *formModel* attribute of each field and the *self* attribute of the Form Model.

What makes the correct information appear in each field is its *formAttributeName* setting.

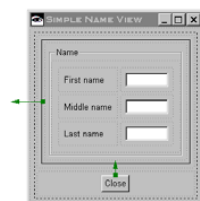
- In the **First name** field, this is set to `firstName`. This corresponds to a *firstName* attribute for the *Name* class.
- In the **Middle name** field, this is set to `middleName`. This corresponds to a *middleName* attribute for the *Name* class.
- In the **Last name** field, this is set to `lastName`. This corresponds to a *lastName* attribute for the *Name* class.

The connection to the *SomeName* part associates the Form Model with a "real" instance of *Name*: between the *model* attribute of the Form Model and the *self* attribute of the *SomeName* part.

Note the use of *valueHolder* rather than *self*. *Name* is a Smalltalk class that follows standard VisualAge protocol. When *SomeName* was dropped, ULC automatically wrapped it in a non-ULC-object wrapper of type *UlcNonUlcObject*. This class is related to the ULC variable discussed previously. The square brackets surrounding *SomeName* indicate that it is neither a true ULC part nor an instance of the *Name* class itself.

This name view is actually a ULC composite part. To (re)use it in other ULC visual parts, two features of the Form Model part must be promoted: the *saveInput* action and the *inputSaved* event.

Now suppose the name view is added to a ULC visual part that looks something like this:

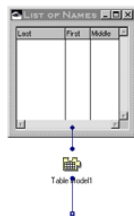


The name composite has been dropped into the upper cell of a Vertical Box part. A **Close** button occupies the lower cell. For this trivial example, we use an alert to indicate that the data has been sent. Here is how the connections go:

1. The *action* event of the **Close** button is connected to the promoted *saveInput* action of the name composite. This triggers the explicit flush mentioned previously.
2. The promoted *inputSaved* event of the name composite is connected to a script called *signalAlert*, which creates and shows a confirmation that the event occurred. In a more robust application, the *inputSaved* event would trigger some other UI action, for example, the synchronization of another view.

## Working with Table Model parts

The Table Model part stands in for collections of objects. Like the Form Model part, the Table Model part defers changes in the UI until explicitly sent to the application server. In “Building the To-Do List with ULC” on page 36, it is used to hold strings for a List part. In the example below, it is used to hold *Name* instances for a Table part:



In this tiny example, a get selector lazy-loads the instance variable *names* (an OrderedCollection) with a few *Name* objects. The attribute-from-script connection between the *rows* attribute of the Table Model part and the *#names* selector sets the Table Model part.

To associate the Table Model part with the Table part, connect the *model* attribute of the Table part and the *self* attribute of the Table Model part.

The Table part contains three Column parts. What makes the correct information appear in each column is its *attributeName* setting.

- In the **Last** column, this is set to *lastName*. This corresponds to a *lastName* attribute for the *Name* class.
- In the **First** column, this is set to *firstName*. This corresponds to a *firstName* attribute for the *Name* class.
- In the **Middle** column, this is set to *middleName*. This corresponds to a *middleName* attribute for the *Name* class.

The finished example looks like this:

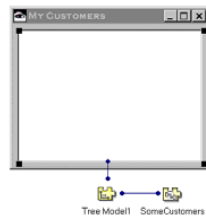
Last	First	Middle
Carpenter	Carolyn	Jayne
Carpenter	Susan	
Carpenter	Oail	

## Working with Tree Model parts

The Tree Model part stands in for an object hierarchy. Like the Form Model part, the Tree Model part defers changes in the UI until explicitly sent to the application server. In the example below, Tree Model is used to arrange a list of *Customer*



objects in a tree view:



The root node of the tree is a *CustomerList* object that contains a collection of *Customer* instances. Each *Customer* instance contains a *Name* instance and a collection of *Address* instances (as *homeAddress* and *workAddress*). Lazy loading at each step into the tree provides a basic demonstration of how this works.

To use the Tree Model part, follow this process:

- Connect up the parts on the Composition Editor.
- Set properties for the Tree Model part.
- Implement methods to support traversal and display in the domain classes.

### Connecting up the parts

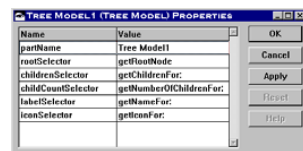
As shown in the previous illustration, the customer-list view is a ULC visual part to which Tree, Tree Model, and CustomerList parts have been added. *CustomerList* (shown here as *SomeCustomers*) is a standard Smalltalk class.

To associate the "real" object with the Tree Model part, connect the *self* attribute of the *SomeCustomers* part with the *model* attribute of the Tree Model part.

To associate the Tree Model part with the tree view, connect the *self* attribute of the Tree Model part with the *treeModel* attribute of the Tree part.

### Setting the Tree Model part

In setting the Tree Model part, you specify the names of the accessors that specify the methods used to traverse and display the tree. An example follows:



These accessors are implemented in *CustomerList* as follows:

```
getRootNode
"answers root"

^self

getChildrenFor: aNode
"answers how to get branch of tree"

^aNode children

getNameFor: aNode
"answers how to get displayable node name"
```

```

^ aNode nodeLabel

getNumberOfChildrenFor: aNode
"answers how to get size of tree branch"

^ aNode numberOfChildren

getIconFor: aNode
"answers how to get icon for displaying node"

^ aNode icon

```

### **Implementing methods to support the tree view**

Now implement the messages sent to aNode. The Tree Model part cannot handle a heterogeneous set of accessors, so the technique for making this work is to implement these methods polymorphically for *CustomerList*, *Customer*, and *Address*. Because there is only one root node, the *getRootNode* method is not implemented elsewhere in the tree.

The remaining implementations for *CustomerList* follow:

```

children
"polymorphic method to access child nodes.
    customers is an instance variable of type OrderedCollection"

^ self customers

nodeLabel
"polymorphic method to provide the name of the node "

^ 'Customers'

numberOfChildren
"polymorphic method to provide the size of tree"

^ self customers size

icon
"polymorphic method to provide icon for tree view"

^UicSharedResource named: 'treeIcon'
    ifAbsent: [UicIcon fromFile: 'bitmaps/red-ball.gif']

```

Implementations for *Customer* follow:

```

children
"polymorphic method to access child nodes.
    addresses is an instance variable of type OrderedCollection"

^ self addresses

nodeLabel
"polymorphic method to provide the name of the node.
    nameString, a string accessor for the embedded Name instance,
    is exposed to the public interface of Customer"

^ self nameString

numberOfChildren
"polymorphic method to provide the size of tree.
    In this example, only two addresses are supported"

^ 2

icon

```

```

"polymorphic method to provide icon for tree view"

^UlcSharedResource named: 'customerIcon'
  ifAbsent: [UlcIcon fromFile: 'bitmaps/green-ball.gif']

Implementations for Address follow:

children
"polymorphic method to access child nodes.
  nil means the end of the tree"

^nil

nodeLabel
"polymorphic method to provide the name of the node.
  addressString is a string accessor for Address instances"

^self addressString

numberOfChildren
"polymorphic method to provide the size of tree"

^0

icon
"polymorphic method to provide icon for tree view"

^UlcSharedResource named: 'addressIcon'
  ifAbsent: [UlcIcon fromFile: 'bitmaps/blue-ball.gif']

```

When complete, the example looks something like this:



## Working with Data Models

### Introduction

ULC has always strived to minimize traffic between the client and the server. For data this means that only data is loaded into the UI that is actually needed. The UI caches all data until it is invalidated from the server. Prior to Release 3.0 ULC has concerned itself mainly with minimizing the traffic related to data contents. Release 3.0 goes one step further and addresses traffic related to presentation.

### Pre R3.0 Releases

Table- and TreeModel were designed to work closely with their widget. They encapsulate both the local ULC cache of application provided data and their structure on the UI. The model defines the order in which the rows are displayed, or the parent-child relationship in the tree's case.

Accessing instance variables of the models' objects is implemented by the model classes. To handle special cases new subclasses had to be created. If data of a given model needs to be displayed in different ways this implementation is too limiting. Example:

1. The same list of employees is shown in one list sorted by names, in another sorted by domicile.
2. Employees of an organisation need to be displayed in a tree showing the hierarchy of the organisation, but also in alphabetical order
3. Subsets of a given set of application objects need to be displayed in various widgets, or changing subsets of the same data in the same widget.

## ULC Release 3.0

**UlcDataModel / ULCLocalTableModel:** Release 3.0 introduces a new model that can serve both hierarchical and indexed itemLists. In Smalltalk this class is called UlcDataModel.

Any presentation instructions (such as ordering of rows, parent-child relationships, root objects for trees etc) must be defined in the itemlists served by an instance of a dataModel. It is possible to directly connect a dataModel to an index-based widget, in which case it works like a TableModel.

**ItemLists:** ItemLists define the structure of the data provided by its model. There are two groups of itemLists:

### IndexedItemList

Provides a mapping of the model's data into an ordered collection. This collection is based on sort criteria in the case of the SortedItemList. Developers may implement their own subclasses to implement for example creation of subsets based on various criteria.

**Note:** A TableModel may serve any number of indexed itemLists.

### HierarchicalItemList

Defines the parent-child relationships of the hierarchy's nodes, the hierarchy's root, and implements the accessing of children / parent of a given node. The ULC class UlcTreeItemList uses a HierarchyAdapter for this, making the itemList more reusable. Application developers will either code their own adapter class, or configure instances of the UlcHierarchyAdapter class included in the release.

**RowAdapters:** The object instance variable access implementations of both UlcTableModel and UlcTreeModel have not changed. They answer themselves as their rowAdapters. Subclasses of these classes may override this, however, and answer a specific rowAdapter. Such adapters may be implemented easily as subclasses of UlcRowAdapter if the application objects do not conform to their platform's instance variable accessing conventions. If they do conform, ULC includes a default rowAdapter that can be used for all application objects. NOTE that UlcDataModel always uses a rowAdapter.

**Summary:** Index-based widgets (List, Table, ComboBox)

- widget uses an itemList
- itemList uses a dataModel
- dataModel uses a rowAdapter

Hierarchical widgets (Tree, TreeTable)

- widget uses an itemList
- itemList uses a hierarchy adapter uses a dataModel
- itemList uses a dataModel
- dataModel uses a rowAdapter

## Working with Row Models

The Row Model provides the functionality of a Form Model. The main difference between the two classes is that the Row Model gets its data (attribute values) from a row of a Table Model. A row model must have a table Model defined. There are two ways to point a row model to a specific row.

- The `widget` attribute of the Row Model can be connected to a list widget (one of `UlcList`, `UlcTable`, or `UlcComboBox`). The row model uses the currently selected item of that widget as its row.

**Note:** The ULC proxy is not aware of the row its UI counterpart is using at any time.

- The `setRow:` method of Row Model allows the application to set the row to which the model should point. Since Row Model uses the `saveInput` mechanism of Table Model, all UI widgets using the table Model are automatically notified of changes made using the Row Model.



---

## Chapter 6. Deploying ULC-based applications

On some Windows 95 machines, you can get the message *Out of environment space* when starting a ULC-based application. Ensure that your environment is large enough by editing the Shell entry in the config.sys file as follows:

```
SHELL=someDisk:\somePath\command.com /P /E:2048
```

/E: specifies the maximum size of the environment. Ensure that the path someDisk:\somePath is valid on your machine.

---

### Packaging ULC-based applications in XD

You cannot package ULC-based applications using **Applications**→**Make executable**; you must use the packager in XD. Packaging ULC-based applications involves the following steps:

- Editing dependencies and prerequisites for the ULC visual and composite parts
- Registering the ULC visual application class
- Creating an appropriate passive image in XD
- Creating packaging instructions and outputting the packaged image

For more information about XD packaging, see the *Server Guide*.

To illustrate this process, we return to the *ToDoList* sample, which represents the simplest ULC program possible: a single view with no user-defined server-side processing. Packaging your application may involve additional steps. If necessary before proceeding, review “Building the To-Do List with ULC” on page 36.

### Preparing ULC visual and composite parts

Before versioning and releasing the visual application for packaging, you must edit its list of immediate prerequisites and dependencies:

- CORBA applications must include *UlcCommunicationIiopApp* in their list of prerequisites.
- All ULC applications must include *UlcRunWidgetApp* in their list of prerequisites.

If necessary, use the Application Editions Browser to edit the list. As soon as this is finished, you can version and release *MyUlcToDoListView*. However, do not version the application yet.

### Registering the ULC visual application class

Add the following methods to the class *MyUlcToDoListApp*:

- *appName* (optional if you prefer to hardcode this in the following examples)

```
appName
```

```
^'MyUlcToDoListApp'
```

- *registerInUlcSystem*, which registers the application with the ULC system object

```
registerInUlcSystem
```

```
UlcSystem
```

```
registerApplicationNamed: self appName
```

```
withStartupClass: MyUlcToDoListView
```

- *loaded*, a synchronization method that calls *#registerInUlcSystem*

loaded

```
self.registerInUlcSystem
```

If the application supports multiple languages, this method should send a message to initialize national language support. For details, see “Implementing NLS support for ULC applications” on page 31.

- *deregisterFromUlcSystem*, which removes the application from ULC system control

```
deregisterFromUlcSystem
```

```
UlcSystem deregisterApplicationNamed: self appName
```

- *removing*, which calls *#deregisterFromUlcSystem*

```
removing
```

```
self.deregisterFromUlcSystem
```

When you are finished, version and release the class; then version the application.

## Creating and populating the passive image

From the System Transcript window, follow these steps:

1. From the XD menu, select **New Image**. The Image Properties window appears.
2. Set properties as appropriate for your application. For ULC, you must load the Ultra Light Client (ULC) feature.
3. Select **OK** to create the passive image.
4. After the passive image has been created, load the applications you want to package into the image:
  - From the XD Transcript window, select **Tools→Manage Applications**. The Application Manager window appears.
  - From the menu bar, select **Applications→Load→Available**.
  - Select each application you want loaded (in this case, only *MyUlcToDoListApp*). Then select **OK**.

## Creating the packaging instructions and outputting the image

1. Switch back to the development image to create a packaging application. We create the packaging application in the development image because it is not necessary to put it in the passive image.
2. From the VisualAge Organizer, create a new application. Call it *MyUlcToDoListPackagingApp*.
3. Edit the immediate prerequisites for *MyUlcToDoListPackagingApp*: Select the application, click mouse button 2, and select **Prerequisites**. The Prerequisites window appears.
4. Packaging applications for ULC must list at least one prerequisite: *UlcPackagingBaseApp*. If necessary, use the Application Editions Browser to edit the list. Then select **OK** to close the window.
5. Switch back to the XD image.
6. From the **Tools** menu, select **Browse Packaged Images**. The Create New Instructions tab appears on top.
7. Choose the type of image you want to package. In this case, select **XD Runtime ULC Application**.
8. Add an image abstract and description in the spaces provided. Then select the **Modify Instructions** box at the bottom of the window.



9. Select the appropriate applications for packaging. (For this example, select *MyUlcToDoListApp*.) Then reduce the image.  
If you are prompted for a significant number of *Sst*- or *Ulc*- applications, the SST and ULC features were probably not loaded properly into the passive image. If this happens, return to “Creating and populating the passive image” on page 50 and repeat.
10. Examine and fix any relevant problems.
11. Save the instruction instructions as a class in the new ULC packaging application. For this example, save the instructions as *MyUlcToDoListPackagingInstructions*, subclass of *ULCBaseXdPackagingForUlcApplications*, in application *MyUlcToDoListPackagingApp*.
12. Output the image to *ulctodo.icx*.

To deploy the image, follow these steps:

1. Copy *esvio.exe* to *ulctodo.exe* (give it the same name as the image file).
2. Create an *ulctodo.ini* file, using *ULCSamples\AppController.ini* as a model:
  - Set *nlspath* to point to the MPR files for your application. Save the file.
  - Add Server Runtime license and key information, using the Unlock Product tool. To apply this information specifically to the *ulctodo.ini* file, select **File** → **Select INI**.

---

## Setting up a ULC development image to run in production mode

If you package your application as described in “Packaging ULC-based applications in XD” on page 49, your application will be enabled for production mode. The following procedure enables you to run your application as a server from a development image:

1. Add a startup method to the application, *registerInUlcSystem*, to register the application with the ULC system object. Note that this is a class method.  

```
MyUlcApp class>>registerInUlcSystem

    UlcSystem
      registerApplicationNamed: 'MySample'
      withStartupClass: MyUlcView
```
2. Configure the *UlcSystem* instance for production mode. This can be done from the System Transcript window by selecting **ULC→System→Change Server Port** from the menu bar.
3. From the System Transcript window, start the server process that will wait for new connections by selecting **ULC→System→Start Server Mode** from the menu bar.  
Your image is now ready to accept new connections from multiple UI Engine components.
4. To connect to this image from a command prompt, use the following:  

```
UIEngine\bin\ulcUI.exe -url ulc://localhost:4444/MySample
```

If connecting to the application from another machine, replace *localhost* with the IP address of the machine running the application. As appropriate, replace 4444 with the server port assigned earlier.

If your application does not run for any reason, ensure that the Debug option is enabled on the **ULC→Debug** menu. Retry the previous sequence and watch the System Transcript window for error messages.

5. You can connect multiple UI Engine components to the same Smalltalk image by repeating the previous step from multiple machines or from multiple command prompts on the same machine. After your tests are complete, you can put your image back in the default mode from the System Transcript window by selecting **ULC→System→Stop Server Mode** from the menu bar.

---

## About running ULC applications from a command prompt

ULC-packaged images support the following command-line parameters. These parameters are not case-sensitive:

**-appName** *String*

Packaged Smalltalk images may contain more than one named application. If this parameter is not set, all applications are accessible. Specifying one or more application names restricts access to the listed applications.

**-corba** Sets the default communication protocol of the image to CORBA (IIOP). This parameter works only for images that include the IIOP support (*UlcCommunicationIiopApp*). If that application is not present when this parameter is passed, the image will not work.

**-debug**

This parameter starts the debug mode of ULC. If started, all debug aspects are output.

A useful complementary VisualAge parameter is *-lFileName*, which redirects output to TTY (the default output for ULC debugging) to the file specified. *-lCON* outputs to the console.

**-server** *XXX*

The number specified by *XXX* defines the port number on which the application server waits for connections.

**-url** *String*

The format of *String* is *ULC://hostname:xxxx/ApplicationName*.

**-userParameter** *String*

*String* can be any valid single command-line parameter (as supported by the platform). *UlcSystem* provides access to this parameter via its *#userParameter* API. This string is not interpreted by ULC. It lets business applications define their own startup parameters.

**-crStrategy** *className*

The name of the ULC reconnect token strategy (default: nil). This turns Client Reconnect feature on if the class exists and inherits from *UlcReconnectTokenStrategy*.

**-crTimeout** *seconds*

Specifies how long a context should be kept alive (default: 300).

**Note:** You can not turn Client Reconnect off by specifying 0 or less. The value must be bigger than 0, otherwise the default is taken. This parameter is only used if the Client Reconnect token strategy (*-crStrategy*) is valid.

### Examples

`MyImage.exe -server 4444`

The application server waits for connections on port number 4444. This configuration allows access to all applications included in the image. To have a UI Engine connect to this server, the command line would read:

```
UlcUI.exe -url ulc://localhost:4444/myBeautifulApp
```

```
MyImage.exe -server 4444 -appName MyBeautifulApp
```

The application server waits for connections on port number 4444 and restricts access to the application named *MyBeautifulApp*. This is useful only if the server image includes another application (for example, *MyUglyApp*).

The UI Engine command lines might read:

```
UlcUI.exe -url ulc://localhost:4444/myBeautifulApp  
result: the MyBeautifulApp starts up
```

```
UlcUI.exe -url ulc://localhost:4445/myBeautifulApp  
result: nothing happens, because the port is wrong
```

```
UlcUI.exe -url ulc://localhost:4444/myUglyApp  
result: nothing happens, because access to MyUglyApp has been refused
```



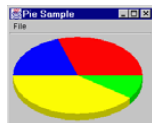
---

## Chapter 7. Implementing ULC objects

The UI Engine can be extended to support new kinds of ULC objects like widgets and data types (that is, type converters and formatters). This section demonstrates the required steps for implementing a ULC widget. As discussed in “Chapter 1. Overview of ULC” on page 3, a ULC widget consists of two half objects:

- The faceless half defines the ULC API for application developers.
- The UI half communicates with a real widget and adapts it so that it can be used by the UI Engine.

In addition to the half objects, there is the real widget, which often already exists and needs to be adapted to be used in ULC. The development of a ULC extension is demonstrated based on the *PieChart* widget, which is shown in the following illustration:



Implementing a new ULC widget requires the following steps:

1. If the widget does not exist, implement the real widget. In this example, the widget is called *PieChart*.
2. Implement the UI half (*UIPieChart*).
3. Implement the faceless half (*ULCPieChart*).

The real widget and the UI half must be implemented in Java. The faceless half can be implemented in either Java and Smalltalk, depending on the needs of your business.

**Note:** All the strings that are used as tags/keys in the ULC protocol between UI and faceless half must contain 8-bit characters (16-bit Unicode is only supported for data and not for the internal ULC protocol).

---

### Implementing the widget

The pie chart widget was implemented from scratch. *PieChart* supports setting the values to be shown, their names, and colors. Clicking on a pie segment sends an action with the name of the clicked segment as an argument.

For simplicity's sake, *PieChart* is not implemented as a model-based widget. Implementing a model-based widget would require implementing a corresponding half object that accesses and tracks changes of a model object.

---

### Implementing the UI half

The UI half objects of an extension to the base ULC widgets can be placed in one or more separate packages. It is important to follow the rules described below (i.e. provide a `typeString` method that returns the fully qualified class name). Do not modify the existing ULC Java packages! In this example the package is called `com.ibm.ulc.examples.pieExtension`. For more information on using Java, see the javadoc.

**Note:** The extensions must be placed in the *classpath* of the UI Engine. The easiest way of doing this is to export the extension to the *UIEngine/lib* directory.

To instantiate an object, the UI Engine uses the following name-qualification rules:

- If the type string supplied by the ULC half object contains a period (*.*), the name is assumed to be a fully qualified class name, and the UI tries to instantiate the class that matches the supplied name.
- If the type string does not contain a period, the *IClassLookUp* object of the UI searches its known class mappings for a match. If a match is found, the matched class is instantiated.
- If the *IClassLookUp* object cannot find a match, the default UI package prefix *com.ibm.ulc.ui.UI* is prepended to the type string, and the resulting class name is used to attempt the instantiation.

Each UI half object must initialize its state from the faceless half. In addition, *PieChart* offers the following API:

- Handle a request to set the data (values, labels, colors)
- Send an event to the faceless half when a pie segment is clicked

*UIPieChart* descends from *com.ibm.ulc.UIComponent*. *UIPieChart* adapts *PieChart* so that it can be used by the UI Engine. As part of doing so, it keeps a reference to the *PieChart* widget. Here is an excerpt from the class definition of *UIPieChart*:

```
public class UIPieChart extends UIComponent implements ActionListener {  
    private PieChart fPieChart= null;  
}
```

To restore its state from the faceless half object, *UIPieChart* overrides *restoreState()*:

```
public void restoreState(ORBConnection conn, Anything args) {  
    super.restoreState(conn, args);  
    fPieChart= new PieChart(args.get("w", 200), args.get("h", 150));  
    fPieChart.setData(args.get("data"));  
    fPieChart.addActionListener(this);  
}
```

A call to the inherited *restoreState()* method restores the inherited state. The *PieChart* widget is created and initialized with the restored state. The *ORBConnection* argument is not used in this example and is just passed on to the base class.

Communication between the UI Engine and faceless half objects is based on data objects called *Anythings*. An *Anything* is a dynamic data structure that can be transferred between processes. The *restoreState()* method receives an *Anything* as its argument and uses *Anything* accessor methods to retrieve the individual arguments. *Anythings* can contain either a simple data type or arrays and dictionaries of *Anythings*. Proper retrieval of the arguments requires that both faceless and UI half objects handle *Anythings* in a analogous way. In this case, the *Anything* is a dictionary, and *restoreState()* retrieves the arguments by name. For example, *args.get("w")* retrieves the width argument of the pie chart. In the same way, the *setData()* method restores the pie chart's data from the *Anything* and passes it on to the *PieChart* widget.

So that *UIPieChart* can notify the faceless half when a pie segment is clicked, *UIPieChart* implements the *ActionListener* interface. *UIPieChart* also registers itself as an action listener of *PieChart*.

Requests sent from the faceless half are dispatched to *handleRequest()*. This method receives the name of the request with its arguments packaged as an Anything. UIPieChart implements only a single request (named *setData()*) to set the pie chart's data:

```
public void handleRequest(ORBConnection conn, String request, Anything args){
    if (request.equals("setData")) {
        setData(args);
        return;
    }
    super.handleRequest(conn, request, args);
}
```

The *handleRequest()* method uses *setData()* to extract the data from the Anything and install it in *PieChart*. Calling the inherited *handleRequest()* method enables the base classes to handle its request.

As part of implementing the ActionListener interface, UIPieChart implements the *actionPerformed()* method, which calls *sendEventULC()* to send the event:

```
public void actionPerformed(ActionEvent e) {
    sendEventULC("action", "cmd", new Anything(e.getActionCommand()));
}
```

*sendEventULC()* takes the name of the event, its type name, and an argument. The event's argument is wrapped into an Anything. In this case, it is a simple string that corresponds to the label of the clicked pie segment.

---

## Implementing the faceless half

This section describes how to implement the faceless half of the pie chart. For this example, implementations exist in both Java and Smalltalk.

### Implementing the faceless half in Java

The Java half of the pie chart is defined in the *ULCPieChart* class. In contrast to *UIPieChart*, you can define *ULCPieChart* in a package of your choice. The ULC implementation uses the convention of prefixing faceless class names with the prefix *ULC*. When the faceless widget creates its UI counterpart, it passes the type name of the widget to be created to the UI Engine. By default, the type name corresponds to the class name without the *ULC* prefix. If desired, this default can be changed by overriding the *typeString()* method defined in the *UlcProxy* class.

Faceless half objects must retain the state of their corresponding UI halves. To do so, *ULCPieChart* stores the values, labels, and colors of the pie chart widget in its instance variables:

```
public class ULCPieChart extends ULCComponent {
    protected double[] fValues;
    protected String[] fColors;
    protected String[] fLabels;
    int fWidth;
    int fHeight;
}
```

*ULCPieChart* identifies its UI half by implementing the *typeString()* method, as follows:

```
public String typeString() {
    return "com.ibm.ulc.examples.pieExtension.UIPieChart";
}
```

ULCPieChart has to implement the UI Engine requests symmetrically. First, it must transfer the widget state to the UI Engine; this is done by overriding the *saveState()* method. *saveState()* packages the arguments kept in its instance variables into an Anything:

```
public void saveState(Anything a) {
    super.saveState(a);
    a.put("w", fWidth);
    a.put("h", fHeight);
    Anything data= new Anything();
    fillData(data);
    a.put("data", data);
}
```

The *setData()* method is implemented in ULCPieChart as follows:

```
public void setData(String[] labels, double[] values, String[] colors) {
    fValues= new double[values.length];
    fColors= new String[colors.length];
    fLabels= new String[labels.length];
    System.arraycopy(labels, 0, fLabels, 0, labels.length);
    System.arraycopy(values, 0, fValues, 0, values.length);
    System.arraycopy(colors, 0, fColors, 0, colors.length);
    Anything data= new Anything();

    fillData(data);
    sendUI("setData", data);
}
```

A request method needs only to package its arguments into an Anything. In this case, the arrays with the values, labels, and colors are wrapped into the Anything structure which is expected by the UI half. Then, the request is sent to the UI with *sendUI()*, which transmits the request name and the Anything argument to the UI Engine.

The last step is to handle action events from UIPieChart; to do this, you override *handleRequest()*. This method receives the name of the request with an Anything that stores the request data:

```
public void handleRequest(ORBConnection conn, String request, Anything args) {
    if (request.equals("event")) { // (1)
        String type= args.get("type", "???");
        if (type.equals("action")) // (2)
            distributeToListeners(new ULCActionEvent(this, args.get("cmd", "???"))); // (3)
        return;
    }
    super.handleRequest(conn, request, args); // (4)
}
```

These steps are implemented in *handleRequest()*:

1. Test whether the request is an event.
2. If so, check the type of event to find out if it is an action event.
3. If so, create an instance of ULCActionEvent and use the *distributeToListeners()* method to notify registered listeners.
4. Otherwise, pass the request to the base class.

## Implementing the faceless half in Smalltalk

The Smalltalk implementation of ULCPieChart is conceptually identical to the Java one. The *saveState:* method packs the widget data into the Smalltalk version of Anythings.



```

saveState: aStcAnything

super saveState: aStcAnything.
aStcAnything
  at: 'w' put: self width;
  at: 'h' put: self height;
  at: 'data' put: (self
    fillData: self values
    colors: self colors
    labels: self labels);
yourself

```

The Smalltalk implementation of the *setData:* method follows:

```

setData: aValueCollection colors: aColorsCollection labels: aLabelsCollection
| data |

data := self
  fillData: aValueCollection
  colors: aColorsCollection
  labels: aLabelsCollection.
self sendToUI: 'setData' with: data

```

In Smalltalk, the type name used to request the creation of UI half objects is defined by overriding the *typeString* method:

```

typeString
  ^'com.ibm.ulc.examples.pieExtension.PieChart'

```



---

## Chapter 8. Advanced Techniques

---

### ULC Client Reconnect Feature

#### Introduction

This feature enables ULC applications started in server mode to keep the contexts (i.e. state) if a connection fails. This allows the UI Engine started in client mode to reconnect to those contexts which would normally have been terminated.

Normally, an application server running an ULC application in server mode terminates the context if the corresponding connection gets lost. For the user this means that his state is lost and that he must restart the application. With the Client User Reconnect feature, the user will be able to reconnect to his applications.

When a connection fails an error message will be displayed to the user. This happens if the programmer set it up to do so - browse `#getConnDownErrorMsg` for more detail). If there are no other ULC applications with working connections in the UI Engine, the UI Engine will terminate. If the user desires to reconnect he simply restarts the UI Engine. If auto-reconnect is enabled then the UI Engine will not terminate but will try to reconnect to the application server(s).

When the UI Engine reconnects to the "hanging" context the normal `#saveState` sequence will be run.

#### Keep-Alive Timeout

To prevent the server from being overloaded by hanging contexts, each context (i.e. state) will only be kept alive for a given timeout. After that timeout the user will not be able to reconnect. Instead, he will get a new context. However, it is possible to set this timeout to a very high number.

See "Enable Client Reconnect in Runtime Time" on page 63 or "Enable Client Reconnect in Development Time" on page 64 to learn how to set the keep-alive timeout in a runtime or development time environment.

#### ULC Reconnect Token Strategy

The ULC client reconnect feature assigns a token to each context. This token is used during reconnect to find the contexts. Normally these are the contexts which belong to the user who just tries to reconnect. Therefore this token is normally a unique identifier for a user (i.e. user ID). However the token can also contain other information such as an application name.

Every client reconnect token strategy must be a subclass of the abstract class *UlcReconnectTokenStrategy* and is responsible for creating and handling the token of a context. It is up to the (infrastructure) programmer to implement such a strategy. A company would normally use one single strategy for all ULC applications. Specific strategies can be implemented for administrative and maintenance purposes (e.g. a strategy that allows a super user to reconnect to any hanging context).

The following strategies are included in this release for demonstration and debugging purposes only. They should not be used in production code.

### **UlcUserPasswordTokenStrategy**

This is an example of the common use case where the context token is a unique user ID. This strategy asks the user for name and password. Because user interaction is involved, this strategy also has a UI proxy which handles the user input and encrypts it for minimal security using a challenge string.

**Note:** Even though user name and password are encrypted using a challenge string sent to the UI, this is not very safe since byte codes can easily be re-engineered. Therefore you should only use this strategy if the connection itself is secure (e.g. SSL).

### **UlcShowAllTokenStrategy**

This strategy is an example that shows how every user can be allowed to reconnect to any hanging context.

## **How to Write a ULC Reconnect Token Strategy**

### **Writing Strategies Where the Token Identifies a User**

This section discusses the normal case where the token is used to find out which contexts belong to a specific user. To prevent to user from getting the same application more than once, the framework internally adds the application name to the token. This is needed because if a connection to a (physical) server fails for some time the UI might try to reconnect to all its contexts (i.e. applications).

Since the reconnect feature allows the UI Engine to reconnect to an existing context with potentially secret business data it is important to have a secure ULC reconnect token strategy. Therefore the strategies contained in the release should only be used as examples. If using a symmetric encryption algorithm (as within XXX) keep in mind that it will not be very hard for an enemy to reverse engineer the Java part (i.e. UI proxy of the strategy). If the company's underlying infrastructure (middleware) identifies and knows the user of the connection then the easiest and most secure implementation would be to identify the user on the server side without a UI round trip to ask the user for name and password.

To write the new strategy a new subclass of `UlcReconnectTokenStrategy` must be made and the `getToken` private method must be implemented: This method must return a `String` which is used when a user wants to reconnect to disconnected contexts. Normally this token identifies a user. If the server can identify the user then it can return its ID/token directly; otherwise a request must be sent to the UI to query the user for name and password.

```
getToken
| challenge |

challenge := self nextRandomNumber.
^self
  decrypt: (self context getUIReconnectTokenWithStrategy: self withChallenge: challenge printString
    usingChallenge: challenge
```

The above example makes a call to the UI to ask for user name and password. This is always done using the `UlcContext>>#getUIReconnectTokenWithStrategy:withChallenge:` method. Get the token used for client reconnection from the UI using the specified strategy and the optional challenge (which can be used for encryption). This method must only be used by `UlcReconnectTokenStrategy` subclasses. This request is executed synchronously.

If user interaction is involved (eg entering name and password) then the *typeString* private method must be implemented too: Return the String that should be used as my identifier in all requests to the UI. Of course this method need only be implemented if the strategy sends requests to the UI.

```
typeString  
    'com.ibm.ulc.ui.UserPasswordTokenStrategy'
```

**Note:** The string must be the class name of the fully qualified UI proxy class.

## Writing the UI Proxy of a Strategy

There are four things to keep in mind:

1. If your middleware on the server offers to retrieve the user of the current connection then use this information and don't go to the UI at all.
2. The UI proxy for the strategy must implement the interface *IReconnectTokenStrategy*. The only method that comes with this interface is *getToken(String)* which is responsible to return a token for the given (optional) challenge String. This method must query the user for name and password (or something else to get a token) and then (if desired) encrypt the token eventually using the challenge string.
3. The UI proxy must inherit from *UIProxy*.
4. Make sure that *MyTokenStrategy>>#typeString* returns the fully qualified class name of the corresponding UI proxy class.

## Writing other Types of Strategies

This section describes the methods involved if strategies different from the ones above need to be implemented.

### **isContextToken:sameAsUserToken:**

This method must return a Boolean which is used when the two tokens are compared. A subclass may override this method to implement a strategy based on user/group permissions (e.g. a special user might see all hanging contexts). Together with *getToken* (see above) it is the most common method used to implement the ULC reconnect token strategy.

The next two methods must be kept in sync because they are used to get the token enriched with the application name. Normally the framework attaches the application name to the token that *getToken* returns. If you want to change the scope of the reconnect dialog, e.g. showing all hanging applications (in that server process/image) for a user without looking at the application name, you can do it by overwriting these two methods:

**Note:** Normally these two methods should not be changed.

### **computeTokenFrom: aTokenString for: anAppName**

This method extracts the original token (*getToken*) using the given *aTokenString* (normally taken from another context) and then enriches it with *anAppName*.

### **getTokenFor:anAppName**

Returns a token which is used when a user wants to reconnect to disconnected contexts. Normally this token identifies a user and the *anAppName* to which the user wants to connect initially.

## Enable Client Reconnect in Runtime Time

All the parameters discussed above can be set when a ULC application is started:

### **-crStrategy className**

Specify the name of the ULC reconnect token strategy (default: nil). This activates the feature if the class exists and inherits from *UlcReconnectTokenStrategy*.

### **-crTimeout seconds**

Specify how long a context should be kept alive (default: 300).

**Note:** You can not turn client reconnect off by specifying 0 or less. The value must be bigger than 0, otherwise the default is taken. This parameter is only used if the client reconnect token strategy (-crStrategy) is valid.

## **Enable Client Reconnect in Development Time**

All the parameters discussed above can be set via the System Transcript->ULC->Tools->Reconnecting menu items:

### **Active**

Shows the state of the client reconnect feature. To be active the following conditions must hold:

- *UlcReconnectApp* is loaded
- A valid reconnect token strategy is specified
- The ULC system is in server mode

This menu item is always disabled.

### **Disable**

Disable the client reconnect feature (sets the strategy to nil).

### **Token strategy...**

Set the client reconnect token strategy. A check mark appears if a valid strategy is defined. The old value is kept if an illegal value is entered.

### **Keep-alive timeout**

Specify the keep-alive timeout in seconds. The old value is kept if an illegal value is entered.

## **Packaging**

To add/remove the client reconnect feature to/from a packaged image you can do one of the following. (It is recommended to use the second way.)

- Add/remove *UlcReconnectApp* in the Packager Control Panel (Modify Instructions step)
- Add/remove *UlcReconnectApp* from the packaging instructions (#applicationNamesToPackage and #actualIncludedSubApplicationNames).

## **Known Problems**

The *AppController* sample shows wrong states for the connections after a reconnect, because the *AppController* does not refresh itself. This is not related to the Client Reconnect feature itself.

---

## **Server Support**

### **Introduction**

When started, the ULC Smalltalk Monitor keeps track of messages and bytes sent and received by each *UlcContext* individually. In addition a timestamp of the last

message sent and received is kept. Information is gathered and totalled over the lifetime of `UlcContext`, or from the last reset message a monitor has received. At development time, this tool can aid in debugging. At runtime, the monitor can assist a system administrator to keep track of active users of a given server. The ULC product includes a default implementation of a monitor user interface. Developers may however implement their own custom interface to the monitoring.

## Monitoring at Development Time

Start monitoring from the **Transcript->ULC->Tools->Monitoring** menu and open the monitor view to observe the traffic. One of the start and stop options is available all the time. Monitoring is dynamically startable and stoppable at development time. This is not so at runtime. The Monitor user interface may be opened any time, even if monitoring has not been started.

## Including the Monitor in Packaged Images

If the monitoring tool should be included with your packaged application, make sure you include `UlcMonitoringViewApp` and not just the `UlcMonitoringApp`. The Monitor View has been kept separate from the actual monitoring so as to enable developers to build their own user interfaces for the monitor if needed.

### Starting the Monitoring

If the `UlcMonitoringApp` has been included with the packaged image, monitoring is started if the command line starting the Smalltalk server includes the parameter `-m`. The decision to start monitoring must be made when the server is started. Thereafter, no change is possible.

### Opening the Monitor View

When included in packaged images, the monitor's user interface application registers itself as an application named `ULCMONITOR`. To start the monitor user interface is like starting any other application and its command line would look like this:

```
ulcui -url ulc://localhost:1111/UlcMonitor
```

**Note:** Here `localhost` and `1111` are assumed to be the host and port at which your ULC application has been started

If the server is not started with the `-m` parameter, the monitor's view will still open, but will never show any information.

## How to write a Custom UI for the Monitor

`UlcSystem` includes new API to subscribe to the addition and removal of `UlcContexts`:

### `ulcWhenContextAddedSend: aDirectedMessage`

Whenever a new context has been created, `UlcSystem` will signal this event. The event passes the newly created context as its single parameter.

### `ulcWhenContextRemovedSend: aDirectedMessage`

This event will be signalled by a `UlcContext` just before it is being destroyed. At this point, the context is still fully functional, which will not be the case thereafter. The UI implementation for the monitor has to take care not to send messages to a `UlcContext` after having received this message.

Every new `UlcContext` will be assigned a new instance of `UlcMonitor`. Use the API on the `UlcMonitor` to access information about the state of the context and messages sent and received by the context.

**Note:** The monitor does not get reset even when the context is destroyed but retains all information collected in the context's lifetime until it is manually reset or is garbage collected. Custom user interfaces may use the `UlcMonitor` API, which is also Composition Editor enabled.

---

## Support for ENVY/Stats

### Overview

ULC Release 3.0 enables the ENVY/Stats profiler for ULC. ENVY/Stats is of course a development tool and should not be packaged with applications. ENVY/Stats is a global tool that profiles all active processes in the image. This means that if you want to profile the processing of a specific `UIRequest`, you are also profiling the SST server process, plus all other running processes. When profiling we recommend that only one `UlcContext` be active during the profiling. If the request is profiled. When a new context is being created in `UlcSystem`, the system will set that context as the active one in its profiler. This means that only the context that was created last can be profiled. ULC profiling starts with the arrival of a request from the UI (`UlcContext>>#sendUIRequest:`) and ends when that request has been fully processed. Only one `UlcContext` will be considered for profiling, and for this context only one request will be profiled at any one time.

### Using the ULC Profiler

Send the `#start` message to the default `UlcProfiler`. Every `UIRequest` arriving from the `UIEngine` will be profiled unless one is already being profiled. Profiling stops when you send the `#stop` message to the default `UlcProfiler`. No new `UIRequests` will be profiled after this. The current profiling will run its full course however.

**Note:** The `#enable` message, the `#disable` message, and the `isEnabled` and `isEnabledAndActive` messages have been deprecated.

Send the `#enable` message to the default instance of `UlcProfiler` ( `UlcProfiler` default). If this message is not sent to the profiler, subsequent start/stop commands have no effect. When enabled, the profiler can be started and stopped repeatedly.

Send the `#disable` message to the default profiler to prevent any further profiling. The disabled profiler will hold on to its active context. It is therefore possible to enable it again later and it will work on the same `UlcContext`.

All commands are available from the **Transcript->ULC->Tools->Profiling** menu.

**Note:** The configuration map `Ulc ST ENVY/Stats Support` requires that the map `ENVY/Stats ES-All` is loaded in the image already.

---

## Test Workbench

### Standardized Test Format

Having a standard format for all or most of our ULC tests makes it possible to provide a flexible environment for running those tests. In addition, the format also allows us to maintain a growing suite of tests. Test cases should be simple and easy to use. The test controls should be grouped in a small units (see below) in one notebook, or they be implemented as separate test cases. All tests are expected to be subclasses of `UlxAbstractTestComposite` and generally conform to this layout:



### Test Area

The widgets being tested are grouped in the left part of the split pane.

### Control Area

The right half of split pane contains the controls to manipulate the tested widgets and instructions as to how to use the controls. If there are too many controls to fit in one border, the control area may be implemented using a notebook. In any case, below every group of control widgets is the instructions panel explaining how the controls must be used, in what order (if any), etc.

### Test API

In addition to the interactive test mode there is API in place that allows tests to be run in unattended mode. Subclasses of *UlxAbstractTestComposite* implement the (automatically generated) *#runIt* method:

```
testStarted("The name of the Test");
    boolean result = callingTheTestRoutine();
    testCompleted(result);
    return result;
```

Classes that have meaningful unattended tests to run should call the test methods after the *#testStarted*, but before the *#testCompleted* method. Those two methods will take (eventually) log the test(s), and update the Test Suite window's visual feedback widgets. The string passed to *#testStarted* names the test, and is added to the log entry. The boolean parameter of *#testCompleted* indicates the success of the test, with nil/null standing for no test run. Classes that cannot run unattended tests, should nevertheless log the fact that they do not have tests by passing "N/A" to the *#testStarted* method, and null to the *#testCompleted* method, while doing nothing between the two calls.

### Test Data

Whenever applicable, test cases should create their test data based on the data scope defined by the test suite in which the tests are running. To facilitate this, the test project includes the class *UlxTestData* which implements methods to get the standard sets of data. An instance of this class is created by every test suite and should be used by tests (by using their inherited API *#getTestData*) to obtain the data needed in the test. Class *UlxTestData* will be extended by OTI to provide most if not all data needed for all supported UlcProxies.

- DATA\_NORMAL (the default)
- DATA\_HUGE (very large data)
- DATA\_NONE (no data)

## Workbench

The workbench is the main interface of the project. It can be opened in:

### Smalltalk

1. Load the application *UlxWorkbenchApp*
2. Open the workbench from the Transcript menu **ULC->Tools->Test Workbench**

The window opening is the ULC Test Workbench. It contains a notebook with three pages.

- Create Test Suite
- Create Test

- All Tests

## Create Test Suite

The page lets you dynamically define test suites based on information held by each test. Currently these selection modes are supported:

- By class and API, where class is the *UlcProxy* class tested, and API is the tested API of that class.
- By Problem Report (PR) where tests created to verify a given PR can be selected

Selecting items from the Test Selection area of the page dynamically updates the test list to the right. If nothing is selected in the selection area, all tests are deemed selected. Individual tests may be removed from the selected tests list. Once the selection has been completed, open the test suite window with the selected tests by pressing the Launch with ... button in the lower right corner of the window. Below the button the data scope to be used in the test suite can be defined. Test suites may be named in the field below the launch button. Currently this name is only used in the label of the test suite window.

## Create Test

The page is an interface to create a new test implementing all static / class methods needed by the test workbench.

- Compulsary Input:

### Test Class

The class name of the test composite to be created.

### Package / Application

The package / application in which Test Class will be defined

### Description

A short description of the test. This text is displayed in the summary field of the Test Suite editor. It is not intended to include test instructions.

### One of the following

#### Classes / API

##### Classes

A list of *UlcProxy* classes that are being tested in the test. In most cases, the list will include only one class name

**API** If a subset of the classes' API is being tested, the selectors should be listed. If the class as a whole is tested, enter ALL into the list of API.

##### Category

This is currently a free text list. Future releases will provide an extendable list of standard categories to choose from.

**PRs** If one or more problem reports are being verified in the test case, the PR's ID should be entered here. The description field could include the problem report's description.

- Optional Input:

### Author

Free text

### **Tab String**

This string is used as the label of the test's page in the test suite window. If this entry is left empty, the test's class name will be displayed instead.

Press the Save Input Button to (re)-generate the test class. The generated class is immediately included in the Test Suite Editor

### **Edit Test**

Upon creation of a new test case, or if you select to edit an existing test, the VisualAge composition editor (CE) is automatically opened on the test case. Initial layout of newly created tests is already the standard test composite.

### **Test Overview**

All tests currently loaded in the image are listed in a sortable table widget. Individual tests may be run or edited (in a composition editor) from the table's popupmenu.



---

## Chapter 9. Troubleshooting ULC applications

ULC includes a number of development support tools that are accessible from the **ULC** menu of the System Transcript window. The exact composition of the **ULC** menu depends on the ULC applications loaded in the image. If *UlcDevelopmentSupport* is not loaded, the **ULC** menu does not appear at all.

**Important:** These tools provide low-level access to ULC internal code. These are manipulated at the user's peril. In particular, this concerns Smalltalk processes and any *Sst-* prefixed classes. For general information about SST process management, see "ULC and Server Smalltalk" on page 26.

---

### Named ULC contexts

Every instance of *UlcContext* defines the space in which a ULC application resides. Two significant development contexts are summarized here:

- **Builder Context**, in which all views run when tested from within the Composition Editor or VisualAge Organizer. This context becomes available when the *UlcEditWidgetApp* application is loaded in the image.
- **Example Context**, which serves as context for all examples shipped with the release. This context becomes available when the *UlcExamples* application is loaded in the image.

Both named contexts offer three menu choices (disabled when their target context is not active):

- **inspect** opens an Inspector window on the context. In this window, the context can be walked.
- **reset** terminates the context. This option brings down the UI and closes the context with all its connections. After the reset, all objects of the context should be garbage-collected. If the context was the last one registered, *UlcSystem* performs a runtime exit.
- **inspect selected widgets** opens an Inspector window on a collection of all instances of selected ULC widget classes currently available in the context.

---

### Cleaning up the ULC system

When the ULC system stops working, stops with a Debugger window, or just will not stop, you can reset the system. From the System Transcript window, select **ULC→Debug→Reset ULC System**. In response to this selection, all active ULC contexts stop, and all open communications are ended. ULC Monitor should no longer list any open connections.

**ULC→Debug→Inspect all ULC Instances** provides verification of a cleaned-up image if it lists only these instances:

- *UlcSystem* (one instance; the default system is always initialized)
- *UlcIconImageDescriptor* (used in the VisualAge parts palette, the number depending on the number of ULC parts loaded)
- *UlcUndefinedProcessOwnerToken* (one internal instance defined in a pool)

Any application-provided objects and all instances of *UlcAbstractView* subclasses still visible after the reset constitute a potential problem (runtime memory leak in

ULC servers). The causes should be investigated and eliminated. Causes for hanging instances are known to include references to the object from some global object; start looking for those first.

---

## Using the Debugger window with ULC

ULC communications employ Smalltalk processes extensively. Terminating one of these ULC processes from the Debugger window can stop all communications for the context using that process. Usually it suffices to reset the builder context, but sometimes the UI must be stopped altogether.

- To ensure a clean image before resuming tests, select **ULC→Debug→Reset Ulc System** from the System Transcript window.
- To verify that all ULC objects have been garbage-collected, select **Inspect all Ulc Instances** from the same menu.

Be careful when setting breakpoints in ULC code, because they can render a process unusable thereafter. When this happens, make sure that the image is in a consistent state before you continue testing.

---

## Configuring #debugPrintString

You can configure the way ULC-controlled objects are printed when inspected in Inspector and Debugger windows. To do this, toggle options under **ULC→Debug→#debugPrintString** from the System Transcript window as follows:

- **Standard format** uses the default image implementation. This option is used only by subclasses of *UlcObject*. Select **Use standard #debugPrintStream** only.
- **Short format** uses the *#ulcPrintObject* method implemented in *Object*. All objects pointed to by the one inspected are printed using the same format. Select **Use short #debugPrintStream** only.
- **Long format** prints the entire object tree, which can be extremely large. Deselect both options.

---

## Tracing inside the Smalltalk image

ULC includes a trace facility that prints messages to a configurable stream interface based on a system of aspects.

- To toggle the tracing option from the System Transcript window, select **ULC→Debug→Debug Mode**.
- To configure the aspects actually output, select **ULC→Debug→Filter Debug Aspects**. This enables a trace that displays only the information requested. For more information about settings, see “Default ULC debugging aspects” on page 73.

To direct output from the debugger from the System Transcript window, select **ULC→Debug→Set Debugger To** as follows:

- **Transcript**.
- **TranscriptTTY**. At run time, this is the only option that is usable.
- **WriteStream**. This is a generic Smalltalk *WriteStream*; it is also the fastest and safest of the three options.

If output is directed to *WriteStream*, you can display the contents of the stream in the System Transcript window.

- To display the contents, select **ULC→Debug→View Debug WriteStream**.

- To reset the contents, select **ULC→Debug→Set Debugger To→WriteStream**.

ULC provides for user-defined aspects to be handled in the same routine. This feature can be used anywhere in your application code. Send the `#ulcDebug:print:` message to the `UlcSystem` default, with the first parameter being the aspect and the second one a block that evaluates to a string. If the aspect passed is selected, the result of the print block is appended to the current debug stream. The output is structured as follows:

```
ULC
<_ASPECT_String_Name_> <_Sending object printObject_String_> -> <_value of printBlock_>
```

The sending object is printed using `UlcBaseApp->Object>>#printObject`. This method in turn sends the `#ulcPrintObjectOn: aStream` message to the receiver. By default, that method prints the class name of the receiver, followed by the state of the receiver in brackets. The method printing the state (`#ulcPrintStateOn: aStream`) is typically reimplemented by classes and can be used by application classes as well.

If the aspect is set to nil, the `#ulcDebug:print` method of `UlcSystem` prints the block regardless of which aspects have been selected. This enables on-the-fly tracing during development.

This feature is not intended for packaged images, because it is not suppressible. In this case, you can extend the list of ULC aspects as follows:

- From code (typically the `#loaded` method).
- From the System Transcript window, select **ULC→Debug** and then either **Add Aspect** or **Remove Aspect**. Aspects added from the System Transcript window are not persistent and are available only in the image where they were added.

## Default ULC debugging aspects

- **All Aspects** traces all aspects (including application-defined ones).
- **Communication** traces low-level communication.
- **Save FormModel input** traces (with timestamp) before or after saving the input from a form model.
- **LifeCycle** traces life cycles of objects (creation, registration, destruction).
- **ModalWait** traces the start and end of a modal wait for an answer.
- **Obsolete** traces all senders of obsolete methods.
- **Receive** traces all incoming message (from UI).
- **Send** traces all outgoing messages (to UI).
- **ShouldNotHappen** traces events that are generally not critical, but should not happen by design.
- **SignalEvent** traces ULC events that are about to be signaled.
- **Synchronization** is used only by `UlcApplication>>synchronize`.
- **Views** traces view-related events. Default is opening and destroying a view.

## Defining application-specific debugging aspects

Although the application owns the aspects, the values of these aspects are defined by `UlcDebugger` and can be different in every image. You set these as follows:

1. Define aspect names in the `_PRAGMA_` method of your application.

```
_PRAGMA_UlcExampleDebugAspectConstants
"%PRAGMA DECLARE
(name: UlcExampleDebugAspectConstants isPool: true)
(pool: UlcExampleDebugAspectConstants declarations: (
(name: UlcCustomDebugAspectOne isConstant: false )
(name: UlcCustomDebugAspectTwo isConstant: false )
))
"
```

Always set the *isConstant* flag to false. If you do not, the pool entry is marked read-only, and the value defined by *UlcDebugger* cannot be assigned (step 2).

2. Initialize the aspects defined in step 1.

```
SomeApplication class>>#loaded
  UlcExampleDebugAspectConstants::UlcCustomDebugAspectOne :=
    UlcDebugger nextAspectIdentifier.
  UlcExampleDebugAspectConstants::UlcCustomDebugAspectTwo :=
    UlcDebugger nextAspectIdentifier.
  UlcDebugger
    addAspect: UlcExampleDebugAspectConstants::UlcCustomDebugAspectOne
      text: 'name of the first aspect as shown in the selection promptter'.
  UlcDebugger
    addAspect: UlcExampleDebugAspectConstants::UlcCustomDebugAspectTwo
      text: 'name of the second aspect as shown in the selection promptter'.
```

3. Use the aspects in application code.

```
SomeObject>>#someMethod
...
  UlcSystem default
    ulcDebug: UlcCustomDebugAspectTwo
    print: ['some text shown in the trace'].
...
```

4. Remove the aspects previously defined.

```
SomeApplication class>>#removing
  UlcDebugger
    removeAspect: UlcExampleDebugAspectConstants::UlcCustomDebugAspectOne
    removeAspect: UlcExampleDebugAspectConstants::UlcCustomDebugAspectTwo
```

---

## Customizing exception handling by context

You can customize exception handling by ULC context in the startup class for any ULC-packaged application. This exception handling can completely replace the default, partly replace the default, or merely perform application-specific actions before turning over control to default exception handling. For more information about contexts before you get started, see “ULC and Server Smalltalk” on page 26.

### Default exception handling in ULC

ULC handles instances of three exception classes (and their subclasses). Behavior varies with environment.

#### ExError

During development, the following occurs:

1. ULC opens an Inspector window on the context in which the error occurred. You must manually terminate the context because it will no longer be able to process user input.
2. ULC opens a Debugger window on the process in which the error occurred.

At run time, the following occurs:

1. ULC logs the error with the debugger.
2. ULC attempts to display an error message in the UI to inform the user that the connection is being closed because of an error.
3. ULC terminates the active context.

#### ExHalt



During development, ULC opens a Debugger window on the process in which the exception occurred.

At run time, ULC ignores this event and resumes the process.

### ExUserBreak

During development, ULC opens a Debugger window on the process in which the exception occurred.

At run time, ULC ignores this event and resumes the process.

## Implementing custom exception handling

To customize exception handling, implement a class method called `#exceptionHandlerForContext: aUlcContext` in the ULC startup class for your application. This class is typically a subclass of `UlcAppBldrView`. At run time, the active context sends this message to the startup class before an instance of the class is created.

```
MyAppBldrViewStartupClass class>>#exceptionHandlerFor: aUlcContext
    "answer the custom exception handler for this application"

    ^MyCustomExceptionHandlerClass new context: aUlcContext; yourself
```

If the receiver of the message answers nil, default exception handling is installed on the current process. Any other object answered is expected to have implemented the `#catchExceptionsWhile: aBlock` method. The `aBlock` parameter represents the exception-handling code to be run.

```
MyCustomExceptionHandlerClassOne>>#catchExceptionsWhile: aBlock
    "Install the custom exception handlers on aBlock.
    Terminate the receiver's context in all cases"

    ^aBlock
    when: ExError
    do: [:signal|self closeApplicationLog.
        "performing some application specific tasks"
        UlcDebugger errorMessage: self someApplicationSpecificErrorMessage.
        [self context terminate] forkNamed: 'terminating context after error'
    ]
```

If default handling suffices for now, you can replace the last two lines of the handler block with code that invokes default handling, as follows:

```
MyCustomExceptionHandlerClassTwo>>#catchExceptionsWhile: aBlock
    "Install the custom exception handlers on aBlock.
    Terminate the receiver's context in all cases"

    ^aBlock
    when: ExError
    do: [:signal|self closeApplicationLog.
        "performing some application specific tasks"
        signal signal "invoking default handling"
    ]
```

To change the exception handler dynamically, send `#setExceptionHandler: anObject` to a context or its `UlcProcessOwnerToken`. `anObject` must have implemented the `#catchExceptionsWhile: aBlock` method. `anObject` can also be nil, which causes exception handling to revert to the default.

In any case, the new exception handler takes effect only on processes spawned after it was set. Any handler already installed on a process is not affected by the change.

To change the error message displayed at run time, edit the message called *UlcAlertTerminateContext*. You can find the definition for this message in the files *UlcWidxxx.tra* and *UlcWidxxx.mpr*.

## Customization considerations

A process in which an *ExError* exception occurs is stopped. The context and UI associated with this process is no longer functional. Custom handler code must relieve this situation by doing one of the following:

- Terminate the context by sending *#terminate* to the object answered by *#ulcActiveContext*. Use a forked process to do this, because the exception-causing process might no longer be functional.
- Resume the process by sending *#resumeWith: anObject* to the Signal object. Do this only when the original error is well understood and known to be resumable.
- Invoke default exception handling by sending *#signal* to the Signal object.

If the handled exception disrupted the connection between application server and UI, attempts to display messages on the UI will probably not succeed.

---

## Chapter 10. Frequently asked questions

---

### General Questions

#### How do I run the UI Engine as an applet?

When you want to run the ULC UI Engine as an applet you need a browser with JDK 1.2 support. The currently supported Web browsers for running the ULC UI Engine as an Applet are Netscape Communicator 4.x and Microsoft Internet Explorer 4 or 5. The browser must also have the Java Plug-in 1.2.2 installed.

You can verify that the browser opens the applet successfully by opening the Java Console. If your browser doesn't have the required JDK support the console will show a corresponding exception. If you have no such browser available you can still run the UI Engine using the Java Runtime Environment (JRE) and in this case no HTML browser is needed.

#### How do I increase the Environment under Win95?

On some Win95 machines you may get the message Out of environment space when starting a ULC application. The fix is to ensure that your environment is large enough. Under Win95 the environment is set by adding/editing the entry Shell in the file c:\Config.sys. Add/Edit the entry to be:

```
SHELL=C:\WIN95\COMMAND.COM /P /E:2048
```

Here the number following /E: specifies the maximum size of the environment. Ensure that the path c:\win95\command.com is a valid path on your machine.

#### How do I run ULC on a standalone machine?

- The easiest way is to go into the DNS page of the TCP/IP bindings (eg. **Control Panel ->Network ->"TCP/IP->Dial-up Adaptor**) and set your host name to localhost. You will want to reset it when you connect back to a network. A very useful shareware utility to switch configurations on Windows based machines is NetSwitcher.
- Another way that has been successful is to do the following:
  1. Install ULC
  2. Install TCPIP on your machine.
    - Win95
      - a. Go to Network in your Control Panel,
      - b. Choose Configuration and click Add.
      - c. Now choose Protocol and select Microsoft as the Manufacturer.
      - d. In the right hand pane choose TCP/IP.
    - WinNT
      - Go to Network in your Control Panel,
      - Choose Protocols and click Add.
      - Choose TCP/IP.
  3. After the above steps reboot your machine.
  4. Use localhost or 127.0.0.1 as the hostName when forming a URL.

**Note:** On machines that are sometimes connected to the network and other times not, the following have been found to help with varied degrees of success on different machines.

- Leave the network card plugged in even if not connected to the network.
- Enable DNS lookup but leave the list of DNS servers empty.

We will update this FAQ as soon as we get a more definite solution to this problem.

## How can I request a remote ULC server to exit?

After ULC R1.3 onwards all ULC server applications interpret a connection request with the application name EXIT to mean that the server itself should exit. Hooks are provided in Smalltalk for the application to override the default behavior which is to do a System.exit immediately. It is up to the application developer to decide what actions should be performed when this EXIT request is received.

For example if you have started a ULC application on your local machine as a server on port 4444. Starting the UI Engine with the parameter `-url ulc://localhost:4444/EXIT` will cause the ULC application server to exit immediately.

To change the default behavior, in Smalltalk the developer must register for the event using the `ulcWhenAboutToTerminateSend:` API in `UlcSystem`.

## How much code is downloaded when connecting to a ULC application?

The typical mode for running ULC applications is where the UI Engine is run as a client on the users machine and the ULC application is running on an application server. In this mode the primary concern for the user will be how much code needs to be downloaded to his machine before a connection to an ULC application is made. Assuming that the user has installed JRE 1.2.2 locally on his machine (this is a one time installation process) then the only code that needs to be downloaded is the UI Engine. The UI Engine is packaged as a single .jar file and is approximately 440KB in size which allows even slow modem connections to perform satisfactorily.

## What kind of applications is ULC designed for?

The strength of applications built using ULC technology are that they can be distributed easily, are automatically web enabled and have very low bandwidth requirements. On every release of ULC we are adding new widgets and capabilities but the primary focus of ULC has been to support the requirements of commercial business applications.

## How can I convert my standalone application to a ULC server application?

The problems of converting a standalone application to a ULC server application are mostly the same faced by any server developer. The key areas that need to be looked at are:

- As far as possible remove references to global/class variables etc. If you have to use globals ensure that the access to them is restricted to a well defined path. You will also need to protect the access to these kinds of variables to ensure that they are thread safe.
- Be aware of the thread un-safe areas of your environment.

- In most cases an Application Context object will need to be introduced to represent the active connection/user that is associated with each running instance of the application. Each running ULC application needs to be able to identify its Application Context. One way to achieve this is to maintain a table keyed on the ULContext instance.
- Be aware that the user interface of your application can now be running remotely. Use the ULC provided enablers/Remote Models/Notification policies etc. to reduce traffic between the application and UI Engine.

## What happens when a connection fails?

The behaviour of the ULC application when a connection fails depends on whether the application is in client or server mode.

### Server mode

If there is no mechanism in place to identify the remote user when the connection is restored and to associate it with a previously running instance of the application, the application just terminates. However, the new Client Reconnect feature enables ULC applications started in server mode to keep the contexts (i.e. state) if a connection fails. This allows the UI Engine (started in client mode) to reconnect to those contexts (which would normally have been terminated). See “ULC Client Reconnect Feature” on page 61 for more details.

### Client mode

In client mode the behaviour is as follows:

#### Smalltalk

In Smalltalk the value of *ULCSystem.connectionRetries* controls the application reconnect behavior. If *connectionRetries=0* then the application terminates immediately. If *connectionRetries=-1* then the connection will be retried forever. If *connectionRetries>0* then the connection will be retried the specified number of times.

## Why do I get an error “ULC Transport Unable to load defaults”?

When attempting to connect a ULC application to a UI Engine on some platforms if the DNS entry of the target machine does not match its configured IP address an error is reported on the Console stating ULC Transport Unable to load defaults. The fix for this problem is to either correct the DNS entry or add an entry in the HOSTS file with the current IP address of the target machine.

## Why am I not able to connect a sample ULC application to a UIEngine on the same machine?

On some platforms the local loopback interface is not enabled by default in the TCP/IP settings. From a command prompt type `ping 127.0.0.1` as well as `ping localhost`. If a reply is not received your machine does not have the local loopback interface enabled.

### Windows

Under Windows 95/98/NT this interface is enabled by default no further configuration is needed.

**OS2** Under OS2 start the `tcpcfg` program and select the loopback interface and choose the Enable Interface option.

### Other platforms

Contact your network administrator for help

## In ULC lists (Table, TreeTable), how is addPreloadColumns different from addPreloadAttributes?

### `addPreloadColumn(attributeName)`

The `addPreloadColumn(attributeName)` or `preload(String attributeName)` api on the *ULCAbstractTableModel* is used for ensuring that all the data for the column associated with the given `attributeName`, is uploaded to the UI at startup. This feature is useful for widgets such as the `ComboBox`, which (typically) have a small number of rows. Since this api precludes any lazy-loading of data for the given column, it is not advisable to use this api on table or tree widgets.

### `addPreloadAttribute(attributeName)`

This api on *ULCAbstractModel* ensures that the attributes for the given name are uploaded each time that row (oid) is uploaded. The oid's themselves are still lazily loaded, but since the attribute always accompanies the oid, one round-trip is saved. On the down side, setting this preload attribute means higher traffic between the ULC and the UI, since this attribute will get updated even if it has not changed.

## What can I do to tune the performance of my ULC List widgets (Tree, Table, TreeTable) ?

- Using *addPreloadAttributes* on the *treeColumn*. If you know for sure, that your tree column is always visible, you can optimize the loading of this column by setting it's attribute to be preloaded.
- `notifyAdd` vis-a-vis `notifyChange`. If you are inserting a large number of rows dynamically into a *treeTable* or a *tree*, it is more efficient to use the `notifyChange`

api rather than the  
`notifyAdd`

api on the *ULCAbstractTableModel*. The down side is that a reload results in the current selection getting lost.

## My long running ULC Java server seems to be leaking memory. What can I do?

When a ULC Java application terminates the `free` method is called on all ULC Java Proxy parts. In some cases classes that are built using the ULC Java Builder have references to other parts that prevent them from being garbage collected. You can optionally implement the method `free` on all subclasses of *ULCVajContext*, *ULCShell* and *ULCVajComposite* to reset any instance variables and other data structures.

**Note:** Ensure that if you do implement the *free* method that you call *super.free()* within your implementation. In addition the *ULCShell* implements the API *setDestroyOnClose(boolean)* set this value to true on those *ULCShells* that can safely be destroyed when they are closed.

---

## Smalltalk Questions

### Why does the Test button in the Composition Editor stop working?

If during the test of a sample ULC part, a Debugger window is opened, the image can be left in an inconsistent state. After closing the debugger, go to the System Transcript window. From the ULC menu, select **Debug** and then **Reset Ulc System**. This forces the image back to a consistent state.

### How do I inspect the objects in a ULC application?

The set of objects in a ULC application is defined in that application's context. When testing a ULC part from the Composition Editor, you can inspect the active context from the System Transcript window. From the menu bar, select **ULC→Debug→Inspect Builder Context**. The context of the ULC examples is also accessible by selecting **ULC** and then **Debug**.

As soon as you are inspecting the context, you can send messages like *findAllWidgetsOfType: 'Table'* or *findAllWidgetsNamed: 'MyWidgetName'*. These methods are implemented for debugging only and are not part of the ULC API.

### Why do I get a Debugger window when saving the public interface of an object?

When defining event names for *UlcCompositeView* or *UlcAppBldrView* subclasses, the names are converted from symbols to atoms. If an atom contains a colon, the compiler expects it to be a selector, which the atom is not. To avoid this problem, do not use colons in event names.

### Why does the development image not respond when I start a ULC application?

This can happen when debugging output is directed to Transcript. Fix this as follows:

1. Issue a user-break.
2. Redirect output to WriteStream (see "Tracing inside the Smalltalk image" on page 72).
3. Reset the ULC system (see "Cleaning up the ULC system" on page 71).
4. Restart the application.

### How can I add a new menu to an existing menu or menubar?

There are two ways to create new menus: select the menu icon from the ULC palette and:

- drop it onto an existing menu or menubar. This creates a new entry in the drop target, and new menu on the canvas, with a link in place that connects the two or
- drop it onto the canvas. This creates a new menu without any connections. Then connect the menu attribute of the holding menu to self of the submenu.

Existing menus cannot be connected using drag & drop.

**Note:** a connection between a menu and its submenu must always begin with the *menu->menu* attribute, and end with self of the submenu. Creating the connection in reverse order results in an event-to-action connection due to the internal workings of VisualAge. Such a connection is not possible,



because *UlcWidgets* do not support the event *#self*. ULC code generation does not generate for such impossible event-to-action connection.

## How do you inspect the Objects being used within a Smalltalk ULC Application?

The complete set of objects involved with one particular ULCApplication is always defined in that application's ULCContext. When testing a ULC Part from the Composition Editor, the context of all ULC parts being tested at any given moment can be inspected from the *Transcript->ULC->Debug->Inspect Builder Context* menu item: The context of the ULC examples is also accessible from the *Transcript->ULC->Debug-* menu.

## Why do I get a debugger when saving the public interface of an object?

When defining event names for *UlcCompositeView*, or *UlcAppBldrView* subclasses, the names are converted from symbols to atoms internally. If an atom contains a colon, the compiler expects it to be a selector, which the atom is not. The result is a debugger. Therefore, do not use colons in Public Interface Event names.

## How can I find a widget that has disappeared?

When manipulating the sizes of widgets, some widgets may disappear from the canvas. To resurface them, open the parts list (popup menu -> View parts list) of a widget containing the disappeared one. Open the settings view of the missing view from the parts list and adjust the size information. If the change is sufficient, the widget will reappear, otherwise repeat the process.

## How can I execute application specific image startup tasks?

Subclass the ULC class *UlcCustomStartup*, and override the *#startup* class method. This method is invoked once per image startup, after the *UlcSystem* has initialized, but before any communication is started.

## How can I open new ULC views?

### The first view (at startup)

The class of the application's first view (let's call it *MyStartupView*) is registered with the class *UlcSystem* (not the instance!) using the *#registerApplicationNamed:withStartupClass:* API, where the first parameter is a String that may be used as command-line parameter, the second one is *MyStartupView*. This ensures that the registered class is given control as soon as the ULC environment is properly initialized. The new instance of *MyStartupView* is opened in a (Smalltalk) process, whose owner is a fully initialized *UlcApplication*.

### Additional views in running applications

Once the initial view is open, any additional view may be opened with this statement:

```
MyOtherViewClass new openWidget
```

The new widget will be opened in the currently active *UlcApplication*, which is well defined if the message was sent from a process started by ULC. This means that opening views from an existing view will result in the new views being created in the *UlcApplication/UlcContext* of the view from which they are opened. Under most circumstances the issue of what *UlcApplication* will hold a new view is of no concern for the application



developer. If you are unsure whether the Smalltalk process is owned by your UlcApplication, send the *#evaluate: aBlock* message to the UlcApplication's processOwnerToken and open the new view inside *aBlock*.

#### Open a child view

If you want to open a new view as child of one already open use:

```
MyOtherViewClass new openWidget
```

Again, if unsure whether the Smalltalk process is owned by your application, proceed as described above.

**Note:** If a view is opened based on UI input (i.e. a Button has been pushed, etc.), the process ownership is set to your UlcApplication by ULC. You do not need to set ownership explicitly. As a general rule of thumb, if the code opening a new view resides in a subclass of *UlcAppBlDrPart*, chances are that you do not need to concern yourself with process ownership. If the code is in general non-ULC application code, you will need to be more careful.

### Packaging when using non-ULC parts in a variable

The default ULC packaging specifications do not include any VisualAge Common Builder extensions. When packaging a ULC application that uses non-ULC parts in a variable you must explicitly include VisualAge Common Builder - Run Headless

### Why do I see two connections when starting a Smalltalk example (exe)?

When you start the UI Engine in server mode (via run.bat) and the Smalltalk sample in client mode (by starting one of the exe files) then you will see two connections in the monitor. This is because the ULC ST Server Monitor is also packaged with the ICX. You can avoid this by creating a shortcut (or batch file) that calls the exe file with the parameter *-appName X* where X is the name of the application.

For example

```
Dossier.exe -appName
```

Dossier would start the dossier sample in client mode without also starting the ULC ST Server Monitor and therefore you will only see one connection in the Monitor.

### Can I have equal objects in a list?

No! TableModels (and ItemLists) must not contain *\*\*\*equal\*\*\** (=) objects. If you use Strings as your list objects, then adding *xxxtwice* would bring the list in a inconsistent state. If you want to have lists that display equal (=) objects then you have to build a wrapper around them.



---

## **Part 2. Programmer's Reference**



---

## Chapter 11. Resource classes

### UlcFont

Specifies the font to be used in ULC widgets. Plain, bold and italic are the font styles currently available in ULC. They are defined in the pool dictionary *UlcWidgetConstants* (*UlcFontBoldStyleBit*, *UlcFontItalicStyleBit*).

To set the style of a font, use the following:

```
aUlcFont fontStyle: UlcWidgetConstants::UlcFontBoldStyleBit
```

To set the font to normal, use the following:

```
aUlcFont fontStyle: 0
```

To reset the font dynamically, you must use the *#setFont:* method.

### UlcIcon

Specifies a GIF image to be used in buttons, labels, and menu items. The following code creates one with the name *abc.gif*:

```
UlcIcon fromFile: 'abc.gif'  
UlcIcon fromFile: 'bitmaps\abc.gif'
```

Loading an icon resource from file uses *CfsStream*, so the file name passed when creating a new instance of *UlcIcon* can be a simple filename as well as a partially or fully qualified path name. In the ULC examples, all resources can be found in the bitmaps subdirectory of the program folder.

ULC also includes an optional caching facility on the User Interface for Icons and icons can now be cached in the UI persistently across sessions.

**Note:** The caching support is currently not available when the UI is running as an Applet.

See *UlcIcon>>#cacheOnUI:*.

### UlcRGBColor

Specifies a color using standard RGB notation. Once created, color instances are immutable.

To create an instance, use the following:

```
|someColor|  
someColor := (UlcRGBColor new) initializeRed:244 green:244 blue:244.
```



---

## Chapter 12. General classes

### UlcProxy

To set multiple attributes of a widget already created on the UI and send the changes collectively to the UI, use the following:

```
UlcProxy>>#bundleRequestsWhile:
```

In applications where shells stay alive for a long time and inside the shells resources are dynamically changed (e.g. with a pagebook or notebook, or by dynamically adding / removing widgets), the *destroyOnClose* flag will not help very much. In order to give finer grained control to the programmer we have added a release API in ULC R3.0a which allows the developer to explicitly free any ULC object on demand by sending release to this object. Sending release frees the resources of the receiver on the ULC side and sends a request to the UI side to free the resources of the corresponding UI object.

**Note:** When releasing a ULC object the developer must make sure that this object and all of its children are no longer used anywhere. For example if you release a ULC model that is still used by other ULC objects then your application will be in an inconsistent state and if a widget tries to use the model on the UI side it will not work correctly and may throw exceptions or even crash. In general the use of the release API should be restricted to only cases where it is absolutely necessary since releasing widgets/models requires the developer to be aware of all places within the application where the widget/model is being used and this can in some cases be error prone.





---

## Chapter 13. Alert

### Category

 Ulc Models

### Palette icon

 Alert

### Class name

*UlcAlert*

### UlcAlert

A general-purpose message box. Although Alert is a shell and appears on the parts palette, its properties cannot be reset after the instance has been created, so this class is better used in code.

*UlcAlert* is typically opened as a child of *UlcShell*, as follows:

```
(UlcAlert confirm: 'Do you really want to quit?'  
  title: 'Please Confirm'  
  parent: aShell)  
ifTrue: [aShell parent terminate]
```

*UlcAlert* handles itself with respect to its parent, so there is no need to add alerts to *UlcApplication*.



---

## Chapter 14. Box parts

Box is a container for other widgets. Several Box-based parts exist on the palette, as follows:

- Box provides a two-dimensional grid of rows and columns.
- Horizontal Box provides a single row.
- Vertical Box provides a single column.
- Box for Radio Group provides a Box with several RadioButton parts. The *radioGroup* attribute of the Box part is preset to true.

### Category

 Ulc Canvas

### Palette icon

 Box

 Horizontal Box

 Vertical Box

 Box for Radio Group

### Class name

*UlcBox, UlcHBox, UlcVBox*

---

## Box attributes

Most of the following attributes are valid for two-dimensional boxes only.

### backgroundColor

The background color of the part.

### columns

The number of columns in the underlying grid.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

**font** The typeface used for text displayed on this part.

### foregroundColor

The color of any text shown in the part, if applicable.

### horizontalGap

The space between adjacent columns (in pixels).

### margin

The space around the perimeter of the part (in pixels).

### radioGroup

Determines behavior of RadioButton or CheckBox parts dropped in the box.

- If *radioGroup* is set to true, only one radio button or check box can be selected at any given time. If the box contains both radio buttons and check boxes, one of either can be selected.
- If *radioGroup* is set to false, multiple radio buttons or check boxes can be selected at any given time. This is the default value.

**rows** The number of rows in the underlying grid.

**self** The class instance itself. This is the only connectable attribute for Horizontal Box and Vertical Box.

**toolTipText**

The text displayed in hover help for the part.

**verticalGap**

The space between adjacent rows (in pixels).

---

## Box events

---

## Box general advice

### Code examples

The dimension of a new box is defined by sending the message *#rows:columns:* or *#rows:columns:radioGroup:* to the class:

```
UlcBox rows: 3 columns: 5 radioGroup: false
```

All boxes define the gap between contained cells, which is set by sending *#setVerticalGap:* or *#setHorizontalGap:* to the box. At initialization, use the methods *#verticalGap:* or *#horizontalGap:*, as follows:

```
UlcHBox new horizontalGap: 5.
```

```
UlcVBox new verticalGap: 5.
```

```
UlcBox new
  horizontalGap: 5;
  verticalGap: 5;
  yourself
```

Box also enables you to expand or align widgets within their cells. The *UlcBox* API includes the following convenience methods:

- *UlcBox>>#alignHorizontalCenter*
- *UlcBox>>#alignHorizontalExpand*
- *UlcBox>>#alignHorizontalLeft*
- *UlcBox>>#alignHorizontalRight*
- *UlcBox>>#alignVerticalBottom*
- *UlcBox>>#alignVerticalCenter*
- *UlcBox>>#alignVerticalExpand*
- *UlcBox>>#alignVerticalTop*

For example, the following code aligns widgets in *UlcBox* to the top left corner of their cells:

```
(UlcBox rows: 3 columns: 5 radioGroup: false)
  alignVerticalTop;
  alignHorizontalLeft;
  yourself
```

Widgets are added to a box sequentially and fill the box row by row (left to right, top to bottom). For more sophisticated layouts, you can specify that a widget uses more than one cell. This is known as *spanning*. In the following code, the `familyName` field is spanned over two cells:

```
(UlcBox rows: 3 columns: 5 radioGroup: false)
  add: (UlcField new ulcName: 'familyName'; columns: 10; yourself)
    horizontalSpan: 2
    verticalSpan: nil
```

For more details on layout and alignment, see “ULC layout” on page 20.

\*



---

## Chapter 15. Browser Context

The Browser Context part represents the default HTML browser in effect for the desktop machine's operating system.

### Category

 Ulc Models

### Palette icon

 Browser Context

### Class name

*UlcBrowserContext*

---

## Browser Context attributes

### browserPath

The directory location of the browser program.

**self**     The class instance itself.

---

## Browser Context events





---

## Chapter 16. Button

### Category

 ULC Buttons

### Palette icon

 Button

### Class name

*UlcButton*

---

## Button attributes

### backgroundColor

The background color of the part.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

### enabled

Indicates whether the part can be selected.

### enabler

The part whose state prompts activation of this part at run time. For more information on connecting enablers, see “Chapter 5. Building ULC applications visually” on page 33.

### foregroundColor

The color of any text shown in the part, if applicable.

**icon** The name of the GIF file displayed in the part.

**label** The text displayed on the part.

### mnemonic

The shortcut letter associated with this part.

### popupMenu

An instance of *UlcMenu* that supports this part.

**self** The class instance itself.

### toolTipText

The text displayed in hover help for the part.

---

## Button events

**action** The button has been clicked.

---

## Button general advice

### Code examples

The following code creates a button with the label **Delete**:

```
UlcButton new label: 'Delete'
```

*UlcAbstractButton* and its subclasses use the standard Smalltalk callback mechanism to notify clients when they are pressed. Methods to register callbacks for ULC events are categorized under *ULC-API-Callbacks*.

```
SomeWindow>>#createDeleteButton
```

```
| button |
```

```
^(button := UlcButton new label: 'Delete')
  addActionCallbackFor:self
  selector: #deleteButtonClicked: clientData: callData: clientData: nil;
  yourself
```

```
SomeWindow>>#deleteButtonClicked: aButton clientData: clientData callData: callData
  "handle the delete action triggered by aButton"
  aButton disable.
  .... "delete now"
```

Tooltips, enablers, and mnemonics are features supported by all button-like classes.

### Enablers

A feature required for buttons and menu items is to change their enable or disable state depending on the state of another widget. In ULC, this is done with **enablers**. An enabler informs its registered widgets that it has changed its state, and effects the change of the enable or disable state on those widgets. The following classes can serve as enablers:

- Entry Field
- Form Model
- List
- Table
- Tree

The following classes can serve as concatenated enablers:

- UlcAndEnabler
- UlcNotEnabler
- UlcOrEnabler
- UlcXorEnabler

The following code uses *UlcTable* as the enabler for *UlcButton*:

```
(UlcButton new label: 'Delete';yourself) setEnabler: UlcTable new
```

The *UlcTable* instance tells the button to change its state to *enable* whenever there is a item selected in the table.

### Mnemonics

*UlcButton* also supports mnemonics (a single character). When the user presses that key with the Alt key, the button behaves as if clicked. The following code sets aButton's mnemonic to the A key:

```
aButton setMnemonic: $A
```

The mnemonic character is parsed from the label automatically provided the marker is the tilde (~) character.

---

## Chapter 17. CheckBox

### Category

 ULC Buttons

### Palette icon

 CheckBox

### Class name

*UlcCheckBox*

---

## CheckBox attributes

### backgroundColor

The background color of the part.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

### enabled

Indicates whether the part can be selected.

**font** The typeface used for text displayed on this part.

### foregroundColor

The color of any text shown in the part, if applicable.

### formAttributeName

The name of the attribute from an associated domain object that is used to set the state of this part.

### formModel

An instance of *UlcFormModel* associated with this part. You must also set *formAttributeName*.

**group** The radio group to which this button belongs.

**label** The text displayed on the part.

### mnemonic

The shortcut letter associated with this part.

### popupMenu

An instance of *UlcMenu* that supports this part.

### selected

Indicates whether the part has been toggled on.

**self** The class instance itself.

### toolTipText

The text displayed in hover help for the part.

**value** The string value that toggles this part on when it is connected to a Form Model part.

---

## CheckBox events

### **newlySelected**

The part has just been toggled **on**.

### **selectionChanged**

The part's toggle state has changed.

---

## CheckBox general advice

### Code example

To track the state of *UlcCheckbox*, use the *addStateChangeCallbackFor:selector:clientData:* callback API.

The corresponding widget for use in menus is *UlcCheckBoxMenuItem*. Example code follows for both components:

```
UlcCheckBox new setSelected: true.  
UlcCheckBoxMenuItem new label: 'Show Table'
```

Tooltips, enablers, and mnemonics are features supported by all button-like classes.

### Enablers

A feature required for buttons and menu items is to change their enable or disable state depending on the state of another widget. In ULC, this is done with **enablers**. An enabler informs its registered widgets that it has changed its state, and effects the change of the enable or disable state on those widgets. The following classes can serve as enablers:

- Entry Field
- Form Model
- List
- Table
- Tree

The following classes can serve as concatenated enablers:

- UlcAndEnabler
- UlcNotEnabler
- UlcOrEnabler
- UlcXorEnabler

---

## Chapter 18. CheckBox Menu Item

### Category

 ULC Menus

### Palette icon

 CheckBox Menu Item

### Class name

*UlcCheckBoxMenuItem*

---

## CheckBox Menu Item attributes

### accelerator

The shortcut key combination associated with this part.

### backgroundColor

The background color of the part.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

### enabled

Indicates whether the part can be selected.

### enabler

The part whose state prompts activation of this part at run time. For more information on connecting enablers, see “Chapter 5. Building ULC applications visually” on page 33.

**font** The typeface used for text displayed on this part.

### foregroundColor

The color of any text shown in the part, if applicable.

**icon** The name of the GIF file displayed in the part.

**label** The text displayed on the part.

### mnemonic

The shortcut letter associated with this part.

### selected

Indicates whether the part has been toggled on.

**self** The class instance itself.

### toolTipText

The text displayed in hover help for the part.

---

## CheckBox Menu Item events

**action** The menu item has been selected.

### selectionChanged

The toggle state of the menu item has changed.

---

## **CheckBox Menu Item general advice**

Several classes implement menu support. For more information, see “Menu general advice” on page 133.

---

## Chapter 19. Column

### Category

 Ulc Lists

### Palette icon

 Column

### Class name

*UlcColumn*

---

## Column attributes

### **attributeName**

The name of the attribute from an associated domain class that is used to display instances of the class in this part. For details, see “List general advice” on page 130.

### **converter**

A subclass of *UlcTypeConverter* that validates and formats input in this part. For more information about converters, see “Field general advice” on page 116.

### **editable**

Indicates whether the contents of the part can be edited at run time.

**label** The text displayed on the part.

### **renderer**

The widget used to interpret or render a state within the part.

**self** The class instance itself.

**width** Initial width of the part in pixels. This attribute is explicitly necessary because dimensions of this part must be known before the part is created at run time.

---

## Column events

---

## Column general advice

The Column part is used within the Table part. For a code example, see “Table general advice” on page 164.





---

## Chapter 20. ComboBox

The ComboBox part combines the aided recall of a list with the flexibility of an entry field. For more information about the use of model parts with ComboBox, see general advice for those parts.

### Category

 Ulc Lists

### Palette icon



ComboBox

### Class name

*UlcComboBox*

---

## ComboBox attributes

### attributeName

The name of the attribute from an associated domain class that is used to display instances of the class in this part. For details, see “List general advice” on page 130.

### backgroundColor

The background color of the part.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

### editable

Indicates whether the contents of the part can be edited at run time.

### enabled

Indicates whether the part can be selected.

**font** The typeface used for text displayed on this part.

### formAttributeName

The name of the attribute from an associated domain object that is used to set the state of this part.

### formModel

An instance of *UlcFormModel* associated with this part. You must also set *formAttributeName*.

### foregroundColor

The color of any text shown in the part, if applicable.

**label** Settable but not used in this part.

### popupMenu

An instance of *UlcMenu* that supports this part.

### preloadContents

Indicates whether the UI should initially load list contents into the part. By default, this is set to true.

**renderer**

The widget used to interpret or render a state within the part.

**rows** The collection of domain objects associated with this part, if you choose not to use a Table Model part.

**selectedIndex**

The index of the selected item (as Integer).

**selectedItem**

The highlighted item.

**selectedString**

The highlighted item (as String).

**self** The class instance itself.

**tableModel**

The instance of *UllTableModel* associated with this part, if you use one. You must also set *attributeName*.

**toolTipText**

The text displayed in hover help for the part.

---

## ComboBox events

**itemInserted**

An item has been inserted into the list.

**selectionChanged**

The user has selected a different item.

---

## Chapter 21. Container

The part displays items with icon and label in a container. Each item must conform to the `UlcContainerItem` class. The items are automatically re-arranged (can be turned off) when the container is resized. It is assumed that there are not hundreds of items in the container because there is no model underneath. Only items that are `UlcLabel` or inherit from it can be modified later.

### Category

 `Ulc Lists`

### Palette icon

 `Container`

### Class name

*`UlcContainer`*

---

## Container attributes

### `autoArranging`

Indicates whether the part is automatically arranged.

### `height`

Minimum vertical size (in pixels).

### `horizontalGap`

The space between adjacent columns (in pixels).

**items** Sets the container's items. Each item must conform to `UlcContainerItem`.

### `labelPosition`

Sets the label position relative to the icon.

### `identifierToItemMapping`

Maps the identifier to the item.

### `multiSelectionAllowed`

Indicates if multiselection is allowed.

### `scrollbarDisplayPolicy`

This integer must be one of these 3 constants: (`UlcScrollbarsAlways`, `UlcScrollbarsAsNeeded` (default), `UlcScrollbarsNever`)

### `selectedItems`

The highlighted items (as a collection).

### `verticalGap`

The space between adjacent rows (in pixels).

**width** Minimum horizontal size (in pixels).

---

## Container events

`ulcWhenActionSend`

`ulcWhenDoubleClickedSend`

`ulcWhenSelectionChangedSend`

---

## Container general advice

### Code example

```
fContainer := ULCContainer new
    items: myItemList;
    width: 100;
    height: 200;
    yourself
```

---

## Chapter 22. Data Model

### UlcDataModel

Release 3.0 introduces a new model that can serve both hierarchical and indexed itemLists. In Smalltalk this class is called UlcDataModel.

Any presentation instructions (such as ordering of rows, parent-child relationships, root objects for trees etc) must be defined in the itemlists served by an instance of a dataModel. It is possible to directly connect a dataModel to an index-based widget, in which case it works like a TableModel.



---

## Chapter 23. Enabler

The Enabler part maps data from hierarchical lists onto a domain object.

### Category

 Ulc Models

### Palette icons

 Enablers

### Class name

*UlcEnabler*

### UlcAndEnabler

Allows an enabler to have n inputs and be used as input for other enablers.

Use the *and: aUlcWidget* method to add a widget to the list of children and attempt to send it to the user interface. If any one of the widget's children collection are enabled then the state is enabled.

### UlcMandatoryAndEnabler

Highlights and marks fields that are required to have input.

Use the *mandatoryStyle* and *mandatoryStyle:aWidget* methods to get and set the receiver's mandatory style. Answers 0, if none is defined

Use the *highlightColor* and *highlightColor: anUlcDefaultRGBColor* methods to get and set the receiver's value of highlightColor.

### UlcNotEnabler

Allows an enabler to have n inputs and be used as input for other enablers.

Use the *enabler* and *enabler: aUlcProxy* method to set the receiver's enabler widget. Answer nil, if none is defined.

### UlcOrEnabler

Allows an enabler to have n inputs and be used as input for other enablers.

Use the *or: aUlcWidget* method to add a widget to the list of children and attempt to send it to the user interface. If any one of the widget's children collection are enabled then the state is enabled.

### UlcXOrEnabler

Allows an enabler to have n inputs and be used as input for other enablers.

Use the *xor: aUlcWidget* to add a widget to the list of children and attempt to send it to the UI. If only one of the widgets in the children collection is enabled then the state is enabled.

---

## Enabler general advice

Using enablers is key to a good performing ULC application as enablers reduce traffic.

| As of ULC Release 3.1, all ULC enabler parts (i.e. subclasses of *UlcEnablerAbstract*)  
| can not only be used to enable/disable a widget but can also toggle any boolean  
| property of a widget by specifying the appropriate setter method name.

| Enablers now offer the setter property. The setter property holds the name  
| (string of a Boolean setter method. If the setter property is set, the enabler uses  
| the Java Reflection API to dynamically invokes the specified setter method with  
| the current state as a parameter. (The enabler does not send `setEnabled(state)` to  
| its targets.)

| **Note:** You normally only specify the setter property on terminal enablers



---

## Chapter 24. Field parts

### Category

 Ulc Data Entry

### Palette icon

 Entry Field



Multi-Line Edit

### Class name

*UlcField*, *UlcMultiLineField*. *UlcMultiLineField* is a subclass of *UlcField*.

---

## Field attributes

### backgroundColor

The background color of the part.

### columns

Width of the part (number of characters in the current font).

### converter

A subclass of *UlcTypeConverter* that validates and formats input in this part. For more information about converters, see “Field general advice” on page 116.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

### editable

Indicates whether the contents of the part can be edited at run time.

**font** The typeface used for text displayed on this part.

### foregroundColor

The color of any text shown in the part, if applicable.

### formAttributeName

The name of the attribute from an associated domain object that is used to set the state of this part.

### formModel

An instance of *UlcFormModel* associated with this part. You must also set *formAttributeName*.

### horizontalAlignment

The placement of text within the part. Possible settings are **Center**, **Left**, and **Right**. The default setting for this part is indicated by **<default>**.

**label** The text displayed on the part.

### notificationPolicy

Determines how often the part signals its *valueChanged* event:

- If *notificationPolicy* is set to Focus Change, the part defers the signal until the UI focus switches out of the field (for example, the Tab key is pressed or a button is clicked). This is the default value.
- If *notificationPolicy* is set to Immediate, the part signals each change (that is, each keystroke) as it occurs.

**password**

Indicates whether an entry into the field should be masked, as in a password field. *false* is the default value.

**popupMenu**

An instance of *UlcMenu* that supports this part.

**selectionBackgroundColor**

The background color of text that has been highlighted in the part.

**selectionForegroundColor**

The foreground color of text that has been highlighted in the part.

**self** The class instance itself.

**toolTipText**

The text displayed in hover help for the part.

**value** The current contents of the field.

---

## Field events

**action** The field is in focus and the Enter key has been pressed.

**valueChanged**

The contents of the field have changed. A field does not signal this event if it is connected to a Form Model part.

---

## Field general advice

ULC provides the following converters:

- *UlcDateValidator*, which ensures that the input is a valid date. The input can be a partial date and still be valid. The Date validator allows multiple date formats (eg: dd:MM;dd:MM:yy;dd:MM:yyyy), which allows partial inputs of dates to be correctly interpreted. In addition, the input is no longer filtered. You can enter all characters and the field is validated when leaving the field.
- *UlcRangeValidator*, which ensures that the input is within a given range (between *minValue* and *maxValue*, inclusive).
- *UlcPercentValidator*, which ensures that the input is a percentage value; that is, its value is always between 0 and 100. It displays the value with a percent sign (%) at the end. The following code associates a percent validator with *UlcField*:  

```
aUlcField setConverter: UlcPercentValidator new
```
- *UlcRegularExpressionValidator*, which enables the setting of Perl-like regular expressions for input validation.

### Code examples

The *columns* property indicates the initial number of characters the field should have. At creation time, the box layout mechanism attempts to provide the selected number of characters or more.

```
UlcField new setColumns: 50
```

Changing the text shown in the field is done by calling the `#setValue:` method, for example:

```
aUlcField setValue: 'New text in my field'
```



---

## Chapter 25. Filler

The Filler part displays a blank cell to reserve a fixed amount of white space.

### Category

 Ulc Canvas

### Palette icon

 Filler

### Class name

*UlcFiller*

---

## Filler attributes

### backgroundColor

The background color of the part.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

**height** Minimum vertical size (in pixels).

**self** The class instance itself.

**width** Minimum horizontal size (in pixels).

---

## Filler events

---

## Filler general advice

### Code example

The dimensions (in terms of cells) of a filler can be set with *#height:* and *#width:* messages. In the following code, the empty cell would be on the upper left:

```
(UlcBox rows: 3 columns: 5)
  add: UlcFiller new
```



---

## Chapter 26. Form Model

The Form Model part maps widgets from a set of entry parts onto a domain object. By default, the Form Model is initialized to upload all its known attributes.

### Category

 Ulc Models

### Palette icon

 Form Model

### Class name

*UlcFormModel*

---

## Form Model attributes

**model** The domain object for which Form Model acts as proxy.

### notificationPolicy

Determines how often the part signals its *modelChanged* event:

- If *notificationPolicy* is set to Immediate, the part signals each change as it occurs.
- If *notificationPolicy* is set to On Request, the part defers the signal until your code sends the *#saveInput* message.

**self** The class instance itself.

**veto** Indicates whether changes are to be sent from the UI back to the domain object without polling. If set to true, changes are sent automatically without polling.

---

## Form Model events

### inputCanceled

Data held in the UI has been cleared from the form model.

### inputSaved

Data has been sent from the UI to the domain object.

### modelChanged

The state of the model has changed.

---

## Form Model general advice

| Form Models now preload all data for their clients reducing the number of round  
| trips required to populate the view.

### Code example

In the example below, the *UlcFormModel* is expected to be accessing an (application) object that has an attribute accessible via *#street* and *#street:* from *UlcField*:

```
|formModel box|
```

```
(formModel := UlcFormModel new) model: someObject.
```

```
box := UlcBox new)
  add: (UlcField new columns: 10; formModelAttribute: 'street';
        formModel: formModel; yourself).

UlcShell new add: box
```

The following parts (all of which are subclasses of *UlcFormComponent*) can be associated with Form Model:

- CheckBox
- ComboBox
- Entry Field
- Label
- Pagebook
- RadioButton
- Slider

### **Form Model as enabler**

*UlcFormModel* can act as the enabler for buttons and menu items, as follows:

```
cancelButton setEnabler: formModel
```

*UlcFormModel* sets the widget to its enable state when the user makes the first change to one of the fields of a form.

For an example of using this part in the Composition Editor, see “Working with Form Model parts” on page 41.



---

## Chapter 27. GroupBox

### Category

 Ulc Canvas

### Palette icon

 GroupBox

### Class name

*UlcBorder*

---

## GroupBox attributes

### backgroundColor

The background color of the part.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

### enabled

Indicates whether the part can be selected.

**font** The typeface used for text displayed on this part.

### foregroundColor

The color of any text shown in the part, if applicable.

**label** The text displayed on the part.

### margin

The space around the perimeter of the part (in pixels).

### popupMenu

An instance of *UlcMenu* that supports this part.

**self** The class instance itself.

### toolTipText

The text displayed in hover help for the part.

---

## GroupBox events

---

## GroupBox general advice

### Code example

*UlcBorder* draws an enclosing line (with a title) around other widgets, as follows:

```
(UlcBorder new label: 'Customer')  
  add: (UlcBox rows: 3 columns: 4);  
  yourself
```



---

## Chapter 28. Html Pane

The Html Pane part provides text-only display of a URL. For more comprehensive URL support, use the Browser Context part.

### Category

 Ulc Canvas

### Palette icon

 Html Pane

### Class name

*UlcHtmlPane*

---

## Html Pane attributes

### backgroundColor

The background color of the part.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

### editable

Indicates whether the contents of the part can be edited at run time.

### enabled

Indicates whether the part can be selected.

**font** The typeface used for text displayed on this part.

### foregroundColor

The color of any text shown in the part, if applicable.

### popupMenu

An instance of *UlcMenu* that supports this part.

**self** The class instance itself.

### toolTipText

The text displayed in hover help for the part.

**url** The URL to be displayed in the pane.

**veto** Indicates whether the application should be polled before the pane locates the target of a link selected from the pane:

- If *veto* is set to true, any URL link chosen by the user is sent back to the application for confirmation before the target is loaded into the pane. This is the default setting, but additional work is required to implement this function. See "Html Pane general advice" on page 126 for details.
- If *veto* is set to false, the target URL is located and loaded without polling.

---

## Html Pane events

### **linkActivated**

The user has selected a URL link from the pane.

### **linkActivatedVeto**

The user has selected a URL link from the pane, and you want to poll the application for access before loading the document.

### **linkError**

The user has selected a URL link from the pane, and the URL cannot be located.

---

## Html Pane general advice

### Setting up link polling

The Html Pane part enables you to poll the application for permission to load a URL before the UI actually loads the document. Implementing this function involves the following steps:

- Set the *veto* attribute to true.
- Listen for the *linkActivatedVeto* event.
- When the event is signaled, evaluate whether the document should be loaded. If so, send the *#setPage: aUrlString* message. Otherwise, redirect program flow (for example, put up an alert).

**Note:** The Html Pane part does not display dynamic HTML.

---

## Chapter 29. Label

The Label part displays single or multi-line labels.

### Category

 Ulc Data Entry

### Palette icon

 Label

### Class name

*UlcLabel*

---

## Label attributes

### backgroundColor

The background color of the part.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

### enabled

Indicates whether the part can be selected.

**font** The typeface used for text displayed on this part.

### foregroundColor

The color of any text shown in the part, if applicable.

### formAttributeName

The name of the attribute from an associated domain object that is used to set the state of this part.

### formModel

An instance of *UlcFormModel* associated with this part. You must also set *formAttributeName*.

### horizontalAlignment

The placement of text within the part. Possible settings are **Center**, **Left**, and **Right**. The default setting for this part is indicated by **<default>**.

**icon** The name of the GIF file displayed in the part.

**label** The text displayed on the part.

### popupMenu

An instance of *UlcMenu* that supports this part.

**self** The class instance itself.

### toolTipText

The text displayed in hover help for the part.

### **verticalAlignment**

The placement of text within the part. Possible settings are **Bottom**, **Center**, and **Top**. The default setting for this part is indicated by **<default>**.

---

## **Label events**

---

## **Label general advice**

### **Code example**

*UlcLabel* can show either text or an icon, as follows:

```
UlcLabel new label: 'a label string'.  
aUlcLabel setIcon: (UlcIcon fromFile: 'abc.gif')
```

---

## Chapter 30. List

The List part holds a selectable collection of objects.

### Category

 Ulc Lists

### Palette icon

 List

### Class name

*UlcList*

---

## List attributes

### attributeName

The name of the attribute from an associated domain class that is used to display instances of the class in this part. For details, see “List general advice” on page 130.

### backgroundColor

The background color of the part.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

### enabled

Indicates whether the part can be selected.

### enabler

The part whose state prompts activation of this part at run time. For more information on connecting enablers, see “Chapter 5. Building ULC applications visually” on page 33.

**font** The typeface used for text displayed on this part.

### foregroundColor

The color of any text shown in the part, if applicable.

**height** Initial height of the part in pixels. This attribute is explicitly necessary because dimensions of this part must be known before the part can be created at run time.

### heightInRows

The number of entries that should initially be visible.

**label** Settable but not used in this part.

**model** The instance of *UlcTableModel* associated with this part, if you use one.

### popupMenu

An instance of *UlcMenu* that supports this part.

**rows** The collection of domain objects associated with this part, if you choose not to use a Table Model part.

<b>selectedItem</b>	The highlighted item.
<b>selectedItems</b>	The highlighted items (as a collection).
<b>selectionBackgroundColor</b>	The background color of text that has been highlighted in the part.
<b>selectionForegroundColor</b>	The foreground color of text that has been highlighted in the part.
<b>selectionMode</b>	Determines how many items may be selected from the list at once. Allowable values are Multiple, Single, and Single Interval.
<b>self</b>	The class instance itself.
<b>toolTipText</b>	The text displayed in hover help for the part.
<b>width</b>	Initial width of the part in pixels. This attribute is explicitly necessary because dimensions of this part must be known before the part is created at run time.

---

## List events

<b>doubleClicked</b>	The user has double-clicked an item in the part.
<b>selectedItemChanged</b>	A different item has been highlighted.
<b>selectedItemsChanged</b>	Different items have been highlighted.
<b>selectionChanged</b>	The user has selected a different item.

---

## List general advice

### Code example

*UlcList* can display a single attribute from a collection of objects, using *UlcTableModel* to fetch the data:

```
|tableModel box|

(tableModel := UlcTableModel new) model: someAddress.

(box := UlcBox new)
add: (UlcList new
    tableModel: formModel;
    attributeName: 'street';
    yourself).

UlcShell new add: box
```

In the Composition Editor, you can accomplish this by setting the *attributeName* property to *street* and making the following connections:

- An attribute-from-script connection from a script that answers a collection of addresses to the *rows* attribute of a Table Model part.



- An event-to-action connection from the *rowsChanged* event of the Table Model part to the *setModel* action of the List part, passing in the *self* attribute of the Table Model through a parameter connection. (Remember: An attribute-to-attribute connection is only effective for initialization.)

### **Using List without a table model**

From the Composition Editor, you can preload String values into the list by editing the *rows* property. You can also assemble a collection of items in code and set the *rows* attribute by passing in the collection. However, for most data of any consequence, you are better off using a table model.



---

## Chapter 31. Menu

The Menu part has two possible uses:

- Dropped on a Menubar part, it is integrated visually with the parent shell.
- Dropped on the free-form surface and connected to a UI part (for example, Entry Field), it becomes a pop-up menu.

### Category

 Ulc Menus

### Palette icon

 Menu

### Class name

*UlcMenu*

---

## Menu attributes

### backgroundColor

The background color of the part.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

### enabled

Indicates whether the part can be selected.

**font** The typeface used for text displayed on this part.

### foregroundColor

The color of any text shown in the part, if applicable.

**label** The text displayed on the part.

### mnemonic

The shortcut letter associated with this part.

**self** The class instance itself.

### toolTipText

The text displayed in hover help for the part.

---

## Menu events

---

## Menu general advice

Several classes implement menu support.

To build a menu bar visually, drop a Menubar part on the free-form surface and connect its *self* attribute to the *menuBar* attribute of a Shell part. Then drop Menu parts on the Menubar part. VisualAge adds the appropriate parts and a connection for each Menu part dropped.

To build a pop-up menu visually, drop a Menu part on the free-form surface and connect its *self* attribute to the *popupMenu* attribute of a UI part.

Then drop any of the following parts on each Menu part:

- Menu Item
- CheckBox Menu Item
- Menu Separator

A menu can be made dynamic by linking it to an attribute of a FormModel or RowModel. For examples, see *UlcMenu>>#model;*, *UlcMenu>>#attributeName;*, and *UlcMenu>>#attributeValue;*. The menu's visibility is then dependent on the value of that attribute, making the menu dynamic.

### Mnemonics

Menus and their items support mnemonics. The following code sets the mnemonic of a menu to F:

```
UlcMenu new
  label: 'File';
  setMnemonic: $F;
  yourself
```

### Code example

Creating a menu bar requires the following steps:

1. Create an instance of *UlcMenuBar* and add it to *UlcShell*.
2. Add instances of *UlcMenu* to the menu bar.
3. Add instances of *UlcMenuItem* to each menu.
4. Optionally, separate the menu items with *UlcSeparator*.

These steps are implemented as follows:

```
|shell menuBar fileMenu|
(shell := UlcShell new) setMenuBar: (menuBar := UlcMenuBar new). "(1)"
(fileMenu := UlcMenu new) "(part of step 2)"
  label: 'File';
  add: (UlcMenuItem new      "(3)"
    label: 'Quit';
    addActionCallbackFor: shell
    ulcApplication selector: #close clientData: nil;
    yourself).
menuBar add: fileMenu      "(remainder of step 2)"
```

---

## Chapter 32. Menubar

### Category

 Ulc Menus

### Palette icon

 Menubar

### Class name

*UlcMenuBar*

---

## Menubar attributes

### **backgroundColor**

The background color of the part.

### **toolTipText**

The text displayed in hover help for the part.

**self**     The class instance itself.

---

## Menubar events

---

## Menubar general advice

Several classes implement menu support. For more information, see “Menu general advice” on page 133.



---

## Chapter 33. Menu Item

### Category

 ULC Menus

### Palette icon

 Menu Item

### Class name

*UlcMenuItem*

---

## Menu Item attributes

### accelerator

The shortcut key combination associated with this part.

### backgroundColor

The background color of the part.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

### enabled

Indicates whether the part can be selected.

### enabler

The part whose state prompts activation of this part at run time. For more information on connecting enablers, see “Chapter 5. Building ULC applications visually” on page 33.

### foregroundColor

The color of any text shown in the part, if applicable.

**icon** The name of the GIF file displayed in the part.

**label** The text displayed on the part.

### mnemonic

The shortcut letter associated with this part.

**self** The class instance itself.

### toolTipText

The text displayed in hover help for the part.

---

## Menu Item events

**action** The menu item has been clicked.

---

## Menu Item general advice

Tooltips, enablers, and mnemonics are features supported by all button-like classes.

### Enablers

A feature required for buttons and menu items is to change their enable or disable state depending on the state of another widget. In ULC, this is done with **enablers**. An enabler informs its registered widgets that it has changed its state, and effects the change of the enable or disable state on those widgets. The following classes can serve as enablers:

- Entry Field
- Form Model
- List
- Table
- Tree

The following classes can serve as concatenated enablers:

- UlcAndEnabler
- UlcNotEnabler
- UlcOrEnabler
- UlcXorEnabler

Several classes implement menu support. For more information about assembling menus, see “Menu general advice” on page 133.



---

## Chapter 34. Menu Separator

Category

 Ulc Menus

Palette icon

 Menu Separator

Class name

*UlcSeparator*

---

### Menu Separator attributes

**backgroundColor**

The background color of the part.

**decoration**

The set of colors and fonts owned by this part.

**enabled**

Indicates whether the part can be selected.

**foregroundColor**

The color of any text shown in the part, if applicable.

**self**

The class instance itself.

---

### Menu Separator events

---

### Menu Separator general advice

#### Code example

*UlcSeparator* groups menu items, as follows:

```
UlcMenu new  
  label: 'File';  
  add: (UlcMenuItem new label: 'New';yourself);  
  addSeparator;  
  add: (UlcMenuItem new label: 'Close'; yourself)
```

Several classes implement menu support. For more information, see “Menu general advice” on page 133.



---

## Chapter 35. Notebook

The Notebook part defines a collection of subforms (that is, pages), any of which users can see by selecting the appropriate notebook tab. The Page part defines each page.

### Category

 Ulc Canvas

### Palette icon

 Notebook

### Class name

*UlcNotebook*

---

## Notebook attributes

### backgroundColor

The background color of the part.

### currentPage

The page being displayed.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

### enabled

Indicates whether the part can be selected.

**font** The typeface used for text displayed on this part.

### foregroundColor

The color of any text shown in the part, if applicable.

**height** Initial height of the part in pixels. This attribute is explicitly necessary because pages can be loaded on demand, and dimensions of this part must be known before any pages are loaded.

**label** The name of the part instance.

**pages** The collection of *UlcPage* instances associated with this part.

### popupMenu

An instance of *UlcMenu* that supports this part.

**self** The class instance itself.

**tab** The position of the current page in the page stack, expressed as an Integer.

### tabPlacement

The orientation of notebook tabs. Possible settings are **Top**, **Bottom**, **Left**, and **Right**.

### toolTipText

The text displayed in hover help for the part.

- veto** Indicates whether the application should be polled before the UI changes the page being shown:
- If *veto* is set to true, interception and polling is possible. This is the default setting, but additional work is required to implement this function. See “Notebook general advice” for details.
  - If *veto* is set to false, pages are changed automatically without polling.
- width** Initial width of the part in pixels. This attribute is explicitly necessary because pages can be loaded on demand, and dimensions of this part must be known before any pages are loaded.

---

## Notebook events

### **tabChanged**

The user has changed the page.

### **tabChangedVeto**

The user has attempted to change the page, and you want to poll the application for access before changing the page.

---

## Notebook general advice

### Code example

The following code creates a notebook showing the account details of a customer on one page and the credit details on another:

```
UlcNotebook new
  add: self createAccountPage;
  add: self createCreditPage;
  yourself
```

The two create-methods each answer an instance of *UlcNotebookPage*.

### Setting up page polling

The Notebook part enables you to poll the application for permission to change to a specific page before the notebook actually changes the page. An example of this might be to restrict access for the Administration page of a configuration notebook to authenticated administrators only. Implementing this function involves the following steps:

- Set the *veto* attribute to true.
- Listen for the *tabChangedVeto* event.
- When the event is signaled, evaluate whether the page change should be allowed. If so, do nothing special. Otherwise, redirect program flow (for example, put up an alert).

Usage of a Form Model part and *notificationPolicy* settings for the various parts also influence the behavior of the UI.

---

## Chapter 36. Page

The Page part is used in notebooks and pagebooks.

### Category

 Ulc Canvas

### Palette icon

 Page

### Class name

*UlcNotebookPage*

---

## Page attributes

### backgroundColor

The background color of the part.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

### enabled

Indicates whether the part can be selected.

**font** The typeface used for text displayed on this part.

### foregroundColor

The color of any text shown in the part, if applicable.

**label** The name of the part instance.

### popupMenu

An instance of *UlcMenu* that supports this part.

**self** The class instance itself.

### toolTipText

The text displayed in hover help for the part.

---

## Page events

### getContents

The contents of the selected page have been received from the application.



---

## Chapter 37. Pagebook

The Pagebook part defines a collection of subforms (that is, pages), one of which can be activated dynamically depending on some state within the program. Think of a pagebook as a notebook without tabs, where control over which page is shown lies exclusively with the program, not the user. The Page part defines each page.

### Category

 Ulc Canvas

### Palette icon

 Pagebook

### Class name

*UlcPagebook*

---

## Pagebook attributes

### backgroundColor

The background color of the part.

### currentPageLabel

The label for the page currently being displayed.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

**font** The typeface used for text displayed on this part.

### foregroundColor

The color of any text shown in the part, if applicable.

### formAttributeName

The name of the attribute from an associated domain object that is used to set the state of this part.

### formModel

An instance of *UlcFormModel* associated with this part. You must also set *formAttributeName*.

**height** Initial height of the part in pixels. This attribute is explicitly necessary because pages can be loaded on demand, and dimensions of this part must be known before any pages are loaded.

### popupMenu

An instance of *UlcMenu* that supports this part.

**self** The class instance itself.

### toolTipText

The text displayed in hover help for the part.

**width** Initial width of the part in pixels. This attribute is explicitly necessary

because pages can be loaded on demand, and dimensions of this part must be known before any pages are loaded.

---

## Pagebook events

### pageChanged

The pagebook has changed which page it is displaying.

---

## Pagebook general advice

### Code examples

In the following code, a separate page is defined for each of three types of banking account:

```
| savings private loan |
(savings := UlcBox new)
  ulcName: 'savings';
  add: (UlcLabel new label: 'Special Rates on Savings accounts'; yourself)
  horizontalAlignment: UlcWidgetConstants::UlcBoxLeftAlignment
  verticalAlignment: UlcWidgetConstants::UlcBoxTopAlignment.

(private := UlcBox new)
  ulcName: 'private';
  alignVerticalTop;
  alignHorizontalLeft;
  add: (UlcLabel new label: 'Private Account Charges'; yourself);
  add: (UlcField new ulcName: 'charges'; columns: 10; yourself).

(loan := UlcHBox new rows: 2; yourself)
  ulcName: 'loan';
  add: (UlcLabel new label: 'Number of Credits'; yourself);
  add: (UlcField new ulcName: 'numberOfCredits'; columns: 10; yourself)
  horizontalAlignment: UlcWidgetConstants::UlcBoxLeftAlignment
  verticalAlignment: UlcWidgetConstants::UlcBoxTopAlignment.

UlcPageBook new
  add: savings;
  add: private;
  add: loan;
  yourself
```

The *ulcName* property of each box added to *UlcPagebook* can be passed in the *#setPage:* message to activate that particular page, as follows:

```
aPagebook setPage: 'private'
```

*UlcPagebook* can set the current page based on the value of a specified attribute. For example, suppose an application needs to support various address formats depending on the residency of a person. Each address format is defined as a page in *UlcPagebook*, with each page having as its name the country whose addresses it handles. When the country of residence is selected, the book automatically sets the correct current page. To define this interdependency, the form model of *UlcPagebook* must be set to the one holding the country attribute. In *UlcPagebook*, its form attribute-name is set to the form model's country-of-residence attribute name, as follows:

```
|person pageBook countries|

person := Person new.
countries := Array with: 'United States' with: 'Germany' with: 'Japan'.
pageBook := UlcPageBook new
  formModel: person;
```



```
formAttributeName: 'countryOfResidence';  
add: (UlchBox new ulcName: 'United States';...; yourself);  
add: (UlchBox new ulcName: 'Germany'; ...; yourself);  
add: (UlchBox new ulcName: 'Japan'; ...; yourself);  
yourself
```



---

## Chapter 38. Progress Bar

Use the Progress Bar part to display the progress of any task between a minimum (start) and a maximum (end) value.

### Category

 Ulc Buttons

### Palette icon

 Progress Bar

### Class name

*UlcProgressBar*

---

## Progress Bar attributes

### backgroundColor

The background color of the part.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

### enabled

Indicates whether the part can be selected.

**font** The typeface used for text displayed on this part.

### foregroundColor

The color of any text shown in the part, if applicable.

### maximumValue

The highest value represented on the part, expressed as an Integer.

### minimumValue

The lowest value represented on the part, expressed as an Integer.

### popupMenu

An instance of *UlcMenu* that supports this part.

**self** The class instance itself.

### toolTipText

The text displayed in hover help for the part.

**value** The current amplitude setting for the part, expressed as Integer.

---

## Progress Bar events

---

## Progress Bar general advice

### Coding example

```
UlcProgressBar new  
  minimumValue: 0;  
  maximumValue: 100;  
  value: 0;  
  yourself
```

---

## Chapter 39. RadioButton

### Category

 ULC Buttons

### Palette icon

 RadioButton

### Class name

*UlcRadioButton*

---

## RadioButton attributes

### backgroundColor

The background color of the part.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

### enabled

Indicates whether the part can be selected.

### foregroundColor

The color of any text shown in the part, if applicable.

### formAttributeName

The name of the attribute from an associated domain object that is used to set the state of this part.

### formModel

An instance of *UlcFormModel* associated with this part. You must also set *formAttributeName*.

**group** The radio group to which this button belongs.

**label** The text displayed on the part.

### mnemonic

The shortcut letter associated with this part.

### popupMenu

An instance of *UlcMenu* that supports this part.

### selected

Indicates whether the part has been toggled on.

**self** The class instance itself.

### toolTipText

The text displayed in hover help for the part.

**value** The string value that toggles this part on when it is connected to a Form Model part.

---

## RadioButton events

### **newlySelected**

The part has just been toggled **on**.

### **selectionChanged**

The part's toggle state has changed.

---

## RadioButton general advice

### Code example

The following code groups radio buttons inside *UlcBox* and turns on radio group mode:

```
UlcHBox new
  setRadioGroup: true;
  add: (UlcRadioButton new label: 'HighRisk'; yourself);
  add: (UlcRadioButton new label: 'MediumRisk'; yourself);
  add: (UlcRadioButton new label: 'LowRisk'; yourself);
  yourself
```

Tooltips, enablers, and mnemonics are features supported by all button-like classes.

### Enablers

A feature required for buttons and menu items is to change their enable or disable state depending on the state of another widget. In ULC, this is done with **enablers**. An enabler informs its registered widgets that it has changed its state, and effects the change of the enable or disable state on those widgets. The following classes can serve as enablers:

- Entry Field
- Form Model
- List
- Table
- Tree

The following classes can serve as concatenated enablers:

- UlcAndEnabler
- UlcNotEnabler
- UlcOrEnabler
- UlcXorEnabler

---

## Chapter 40. Radio Group

The Radio Group part enables you to associate RadioButton or CheckBox parts that are not located together on the user interface. (RadioButton or CheckBox parts within a single GroupBox or Box part are automatically assigned to the same radio group.)

### Category

 Ulc Models

### Palette icon

 Radio Group

### Class name

*UlcRadioGroup*

This part has only one feature on its public interface: *self*. Connect this attribute to the *group* attribute of the RadioButton or CheckBox parts that you wish to associate.

|





---

## Chapter 41. Row Model

The Row Model part maps widgets or data from a row of a Table Model onto a domain object.

### Category

 Ulc Models

### Palette icon

 Row Model

### Class name

*UlcRowModel*

---

## Row Model attributes

**model** The instance of *UlcRowModel* associated with this part.

### **tableModel**

The instance of *UlcTableModel* associated with this part.

### **widget**

The widget associated with this part.

---

## Row Model general advice

The Row Model provides the functionality of a Form Model. The main difference between the two classes is that the Row Model gets its data (attribute values) from a row of a Table Model. A row model must have a table Model defined. There are two ways to point a row model to a specific row.

- The widget attribute of the Row Model can be connected to a list widget (one of *UlcList*, *UlcTable*, or *UlcComboBox*). The row model uses the currently selected item of that widget as its row.

**Note:** The ULC proxy is not aware of the row its UI counterpart is using at any time.

- The *setRow:* method of Row Model allows the application to set the row to which the model should point. Since Row Model uses the *saveInput* mechanism of Table Model, all UI widgets using the table Model are automatically notified of changes made using the Row Model.



---

## Chapter 42. Shell

### Category

 Ulc Canvas

### Palette icon



### Class name

*UlcShell*

---

## Shell attributes

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### **enabled**

Indicates whether the part can be selected.

**font** The typeface used for text displayed on this part.

### **destroyonClose**

Indicates whether the resources (a shell together with all its children) on the UI and the ULC side will be freed when the shell is closed. The default is for the resources to be cached. You can set the *destroyOnClose* flag in the properties view of a shell or directly in the code.

**icon** The name of the GIF file displayed in the title bar of the shell.

### **menuBar**

An instance of *UlcMenuBar* owned by this part.

**modal** Indicates whether the shell is modal.

### **popupMenu**

An instance of *UlcMenu* that supports this part.

### **resizable**

Indicates whether the shell can be resized by the user.

**self** The class instance itself.

**title** The text shown in the title bar for this part.

---

## Shell events

### **aboutToOpen**

The shell has been created and is about to be shown.

### **closeConfirmation**

The shell has received confirmation that it can be closed.

### **moved**

The position of the shell has changed.

### **resized**

The size of the shell has changed.

**windowActivated**

The shell has moved to the top of the z-order.

**windowClosed**

The shell has been closed.

**windowDeactivated**

The shell is no longer at the top of the z-order.

**windowDeiconified**

The shell has been restored to the desktop from a minimized state.

**windowHidden**

The shell is now invisible.

**windowIconified**

The shell has been minimized off the desktop.

**windowShown**

The shell is now visible.

---

## Shell general advice

### Code example

The shell is owned and managed by *UlcApplication*. All shells under one instance of *UlcApplication* belong to the same end-user application.

```
(UlcApplication on: UlcContext default)
  add: (UlcShell new label: 'Customer Information'; yourself)
```

A shell can open other subshells (children) for which this shell acts as parent. The following code opens an instance of *UlcShell* in modal state:

```
(UlcShell new label: 'Model Shell') modal: true
```

When a shell is opened as modal, most parts now support the *BlockInputTrigger* action. This trigger displays the busy cursor and blocks further user input till an answer has been received from the ULC server.

**Note:** When setting the *destroyOnClose* flag to true, make sure that the receiver and all of its children are no longer used anywhere. For example, if you set the *destroyOnClose* flag to true then you can no longer use *show* after you have hidden and closed the shell, since the resources have already been freed.

**Note:** When a Shell is closed it automatically notifies its *ULCApplication* and if there are no other shells opened the *ULCApplication* will terminate itself.

---

## Chapter 43. Slider

### Category

 ULC Buttons

### Palette icon

 Slider

### Class name

*UlcSlider*

---

## Slider attributes

### backgroundColor

The background color of the part.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

### enabled

Indicates whether the part can be selected.

**font** The typeface used for text displayed on this part.

### foregroundColor

The color of any text shown in the part, if applicable.

### formAttributeName

The name of the attribute from an associated domain object that is used to set the state of this part.

### formModel

An instance of *UlcFormModel* associated with this part. You must also set *formAttributeName*.

### horizontal

Indicates whether the slider is horizontal. If set to false, the slider is vertical. The default value is true.

**label** The text displayed on the part.

### maximumValue

The highest value represented on the part, expressed as an Integer.

### minimumValue

The lowest value represented on the part, expressed as an Integer.

### popupMenu

An instance of *UlcMenu* that supports this part.

**self** The class instance itself.

### toolTipText

The text displayed in hover help for the part.

**value** The current amplitude setting for the part, expressed as Integer.

---

## Slider events

### **valueChanged**

The value of the slider has been reset.

---

## Slider general advice

### Code example

```
UlcSlider new  
  horizontal: true;  
  minimumValue: 0;  
  maximumValue: 100;  
  value: 44;  
  yourself
```

If the values are not in the allowed range (0 to 100 in this example), the corresponding extreme value is used (that is, if the value is set to 144, the widget shows the value as 100).

---

## Chapter 44. Split Pane

The Split Pane part provides two separately scrollable components that are separated by a user-adjustable divider.

### Category

 Ulc Canvas

### Palette icon

 Split Pane

### Class name

*UlcSplitPane*

---

## Split Pane attributes

### backgroundColor

The background color of the part.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

### dividerLocation

The position of the divider, expressed as a number from 0.0 (left/top) to 1.0 (right/bottom). A setting of 0.5 produces two pane components of equal size.

### enabled

Indicates whether the part can be selected.

**font** The typeface used for text displayed on this part.

### foregroundColor

The color of any text shown in the part, if applicable.

### leftComponent

In a horizontally oriented split pane, the left cell. (In a vertically oriented split pane, this cell appears at the top.)

### popupMenu

An instance of *UlcMenu* that supports this part.

### rightComponent

In a horizontally oriented split pane, the right cell. (In a vertically oriented split pane, this cell appears at the bottom.)

**self** The class instance itself.

### toolTipText

The text displayed in hover help for the part.

### vertical

Indicates whether the part is oriented vertically. The default setting is **false**; the part appears horizontal.

---

## Split Pane events



---

## Chapter 45. Table

### Category

 Ulc Lists

### Palette icon

 Table

### Class name

*UlcTable*

---

## Table attributes

### autoResize

Determines the resizing behavior of the table:

- **Resize all** distributes the dimensional change equally across all columns.
- **Resize last column** applies the dimensional change only to the rightmost column.
- **Resize off** does not apply the dimensional change at all. Extra space is left empty around the columns.

### backgroundColor

The background color of the part.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

### enabled

Indicates whether the part can be selected.

### enabler

The part whose state prompts activation of this part at run time. For more information on connecting enablers, see “Chapter 5. Building ULC applications visually” on page 33.

**font** The typeface used for text displayed on this part.

### foregroundColor

The color of any text shown in the part, if applicable.

### headerBackgroundColor

The background color for each column header.

### headerForegroundColor

The color of text in each column header.

**height** Initial height of the part in pixels. This attribute is explicitly necessary because dimensions of this part must be known before the part can be created at run time.

### heightInRows

The number of entries that should initially be visible.

**label** Settable but not used in this part.

<b>model</b>	The instance of <i>UlcTableModel</i> associated with this part.
<b>popupMenu</b>	An instance of <i>UlcMenu</i> that supports this part.
<b>preloadColumns</b>	The names of the columns to be loaded into the part at initialization (as a collection of Strings).
<b>rowHeight</b>	The height of each row (in pixels).
<b>rows</b>	The collection of domain objects associated with this part. This attribute is read-only.
<b>selectedItem</b>	The highlighted item.
<b>selectedItems</b>	The highlighted items (as a collection).
<b>selectionBackgroundColor</b>	The background color of text that has been highlighted in the part.
<b>selectionForegroundColor</b>	The foreground color of text that has been highlighted in the part.
<b>selectionMode</b>	Determines how many items may be selected from the list at once. Allowable values are Multiple, Single, and Single Interval.
<b>self</b>	The class instance itself.
<b>toolTipText</b>	The text displayed in hover help for the part.
<b>width</b>	Initial width of the part in pixels. This attribute is explicitly necessary because dimensions of this part must be known before the part is created at run time.

---

## Table events

<b>doubleClicked</b>	The user has double-clicked an item in the part.
<b>selectedItemChanged</b>	A different item has been highlighted.
<b>selectedItemsChanged</b>	Different items have been highlighted.
<b>selectionChanged</b>	The user has selected a different item.

---

## Table general advice

### Code example

To display a list of addresses, you can use *UlcTable* with *UlcTableModel*. *UlcTableModel* uses the same mechanism as *UlcFormModel* to access attributes from the domain models, as follows:

```

|tableModel box|

(tableModel := UlcTableModel new) model: someModel.

(box := UlcBox new)
add: (UlcTable new
      tableModel: tableModel;
add: (UlcColumn new
      attributeName: 'street'; yourself);
add: (UlcColumn new
      attributeName: 'zipCode'; yourself);
yourself).

UlcShell new add: box

```

*UlcColumn* is an integral part of *UlcTable*. *UlcColumn* uses the table's model and takes data from *UlcTableModel* using the *attributeName* property as key. Columns are children of a table. They can be added (using *#add:*) to a *UlcTable* instance.

The Table and Column parts support sorting with a button over each column that allows sorting by that attribute.

You can navigate and edit the table using the keyboard.



---

## Chapter 46. Table Model

The Table Model part maps multiple rows from a table to a collection of domain objects.

### Category

 Ulc Models

### Palette icon

 Table Model

### Class name

*UlcTableModel*

---

## Table Model attributes

### notificationPolicy

Determines how often the part signals the appropriate model-changed event:

- If *notificationPolicy* is set to Focus Change, the part defers the signal until the UI focus switches out of the table.
- If *notificationPolicy* is set to On Request, the part defers the signal until your code sends the *#saveInput* message.
- If *notificationPolicy* is set to Row Change, the part signals an event every time a row is changed.

### prefetch

The number of rows sent from the application to the UI at initialization.

### rowCount

The number of rows represented in the model.

**rows** Collection of rows in the model. This attribute is typically associated with the underlying domain object.

**self** The class instance itself.

**veto** Indicates whether the application should be polled before changes to rows are committed:

- If *veto* is set to true, interception and polling is possible. In this case, your code must explicitly send changes back to the application.
- If *veto* is set to false, changes are posted to the UI Engine cache automatically.

---

## Table Model events

### inputCanceled

Data held for the table has been cleared from the table model.

### inputSaved

Data has been sent from the UI to the domain object.

### rowsAdded

Rows have been added to the table model.

**rowsChanged**

Rows have been changed in the table model.

**rowsRemoved**

Rows have been removed from the table model.

**setData**

Data has been changed in the model, and the domain object is about to be updated. Use this event to trigger adjustment of the data before it is used to update the domain object.

---

## Table Model general advice

The following parts (all of which inherit from *UlcTableList*) can be associated with Table Model:

- ComboBox
- List
- Table

The List part is used to display a single attribute from a collection of objects. The Table can display multiple attributes in separate columns.

Subclasses of *UlcItemListAbstract* can be used to control which rows of a Table Model should be displayed, and in what order. ItemLists must be connected to a Table Model. They may be used by multiple widgets at the same time. This ULC Release includes a default implementation of an ItemList (see *UlcSortedList*). Additional ones may be designed by developers to satisfy specific application needs.

For code examples, see general advice for those parts.

For an example of using this part in the Composition Editor, see “Working with Table Model parts” on page 42.

---

## Chapter 47. ToolBar

The ToolBar part is a menu-like widget that contains Button parts rather than menu items. It can also contain Menu Separator parts; they appear in the running tool bar as extra space between groups of buttons.

### Category

 Ulc Menus

### Palette icon

 ToolBar

### Class name

*UlcToolBar*

---

## ToolBar attributes

### backgroundColor

The background color of the part.

### borderPainted

Indicates whether a visible margin is painted around each button. Setting this to true produces a more three-dimensional appearance.

### buttonMargin

The space around the perimeter of each button (in pixels).

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

### enabled

Indicates whether the part can be selected.

### floatable

Indicates whether the tool bar can be dragged out of its container onto the desktop. When closed, a floating tool bar reappears at its original location in the container.

### margin

The space around the perimeter of the part (in pixels).

### popupMenu

An instance of *UlcMenu* that supports this part.

**self** The class instance itself.

### toolTipText

The text displayed in hover help for the part.

### vertical

Indicates whether the part is oriented vertically. The default setting is false; the part appears horizontal.

---

## ToolBar events



---

## Chapter 48. Tree

### Category

 ULC Lists

### Palette icon

 Tree

### Class name

*UlcTree*

---

## Tree attributes

### backgroundColor

The background color of the part.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by **<default>**.

### decoration

The set of colors and fonts owned by this part.

### enabled

Indicates whether the part can be selected.

### firstColumnLabel

The label for the default column of the tree.

**font** The typeface used for text displayed on this part.

### foregroundColor

The color of any text shown in the part, if applicable.

**label** Settable but not used in this part.

**model** The instance of *UlcTreeModel* associated with this part.

### popupMenu

An instance of *UlcMenu* that supports this part.

### rootVisible

Indicates whether the root node is displayed in the tree.

### rowHeight

The height of each row in pixels.

### selectedItem

The highlighted item.

### selectedItems

The highlighted items (as a collection).

### selectionBackgroundColor

The background color of text that has been highlighted in the part.

### selectionForegroundColor

The foreground color of text that has been highlighted in the part.

### selectionMode

Determines how many items may be selected from the list at once. Allowable values are Multiple, Single, and Single Interval.

**self** The class instance itself.

**toolTipText** The text displayed in hover help for the part.

**treeColumn** Set the tree column of the receiver to a `TreeColumn`. This column normally is the first column of the receiver. The receiver instantiates a default tree column when created. This tree column may be replaced by a custom one, or the default column may be modified by code. Tree Columns conform to `UlcColumn`.

**width** Initial width of the part in pixels. This attribute is explicitly necessary because dimensions of this part must be known before the part is created at run time.

---

## Tree events

**doubleClicked**  
The user has double-clicked an item in the part.

**nodeCollapsed**  
The user has collapsed a node of the tree.

**nodeExpanded**  
The user has expanded a node of the tree.

**selectedItemChanged**  
A different item has been highlighted.

**selectedItemsChanged**  
Different items have been highlighted.

**selectionChanged**  
The user has selected a different item.

---

## Chapter 49. Tree Model

The Tree Model part maps the nodes of *UlcTree* to a collection of domain objects.

### Category

 Ulc Models

### Palette icon

 Tree Model

### Class name

*UlcTreeModel*

---

## Tree Model attributes

### childCountSelector

The method implemented by each node in the tree that answers the number of children at that node.

### childrenSelector

The method implemented by each node in the tree that answers the collection of children at that node.

### iconSelector

The method implemented by each node in the tree that answers the icon used to represent that node.

### labelSelector

The method implemented by each node in the tree that answers the text used to represent that node.

**model** The domain object for which Tree Model acts as proxy.

### rootSelector

The method that locates the root node.

**self** The class instance itself.

---

## Tree Model events

### nodesAdded

Leaves (nodes) have been added to the tree model.

### nodesRemove(d)

Leaves have been removed from the tree model.

---

## Tree Model general advice

### Code example

This model implementation differs somewhat from the other ULC model classes in that it assumes the domain object is an adapter for the tree displayed: *UlcTreeModel* expects its domain instance to represent the entire tree. Model attributes must be set to selectors implemented in the domain model rather than to a node directly. The node is passed as a parameter with each of the selectors. Example code follows:

```
|treeModel box|

(treeModel := UlcTreeModel new)
  childCountSelector: #numberOfSubclassesOf;;
  childrenSelector: #subclassesOf;;
  iconSelector: #iconOf;;
  labelSelector: #labelFor;;
  model: someObject.

(box := UlcBox new)
  add: (UlcTree new treeModel: treeModel; yourself).

UlcShell new add: box
```

The following attributes of *UlcTreeModel* must be configured:

- *childCountSelector*
- *childrenSelector*
- *iconSelector*
- *labelSelector*
- *rootSelector*

The values are all selectors that take one parameter. They are sent to the *UlcTreeModel* domain model with the object for which the information is required.

To display the Smalltalk class hierarchy, the model representing the entire hierarchy implements the following:

```
#numberOfSubclassesOf: aClass "attribute <childCountSelector>"
  "answer the number of direct subclasses of aClass"

  ^aClass subclasses size

#subclassesOf: aClass "attribute <childrenSelector>"
  "answer the collection of aClass' subclasses"

  ^aClass subclasses

#hierarchyRootOf: aClass "attribute <rootSelector>"
  "answer the root class of aClass"

|theClass|

theClass := aClass.

[theClass superclass notNil] whileTrue: [theClass := theClass superclass].

^theClass
```

For an example of using this part in the Composition Editor, see “Working with Tree Model parts” on page 42.

---

## Chapter 50. TreeTable

### Category

 Ulc Lists

### Palette icon

 TreeTable

### Class name

*UlcTreeTable*

---

## TreeTable attributes

### autoResize

Determines the resizing behavior of the table:

- **Resize all** distributes the dimensional change equally across all columns.
- **Resize last column** applies the dimensional change only to the rightmost column.
- **Resize off** does not apply the dimensional change at all. Extra space is left empty around the columns.

### treeColumn

Set the tree column of the receiver to a `TreeColumn`. This column normally is the first column of the receiver. The receiver instantiates a default tree column when created. This tree column may be replaced by a custom one, or the default column may be modified by code. Tree Columns conform to `UlcColumn`.

### backgroundColor

The background color of the part.

**cursor** The ULC cursor that is in effect when the mouse pointer is over this part. Possible settings are **Crosshair**, **Text**, and **Wait**. The default setting for this part is indicated by `<default>`.

### decoration

The set of colors and fonts owned by this part.

### enabled

Indicates whether the part can be selected.

### firstColumnLabel

The label for the default column of the tree.

**font** The typeface used for text displayed on this part.

### foregroundColor

The color of any text shown in the part, if applicable.

**label** Settable but not used in this part.

### popupMenu

An instance of *UlcMenu* that supports this part.

### rootVisible

Indicates whether the root node is displayed in the tree.

### rowHeight

The height of each row in pixels.

**selectedItem**  
The highlighted item.

**selectedItems**  
The highlighted items (as a collection).

**selectionBackgroundColor**  
The background color of text that has been highlighted in the part.

**selectionForegroundColor**  
The foreground color of text that has been highlighted in the part.

**selectionMode**  
Determines how many items may be selected from the list at once.  
Allowable values are Multiple, Single, and Single Interval.

**self** The class instance itself.

**toolTipText**  
The text displayed in hover help for the part.

**width** Initial width of the part in pixels. This attribute is explicitly necessary because dimensions of this part must be known before the part is created at run time.

---

## TreeTable events

**doubleClicked**  
The user has double-clicked an item in the part.

**nodeCollapsed**  
The user has collapsed a node of the tree.

**nodeExpanded**  
The user has expanded a node of the tree.

**selectedItemChanged**  
A different item has been highlighted.

**selectedItemsChanged**  
Different items have been highlighted.

**selectionChanged**  
The user has selected a different item.

---

## Part 3. Reference





---

## Chapter 51. Hide Margin/Show Margin

Select **Hide Margin** to render the dotted outline of the selected part invisible. The margins of all cells within the part are also hidden.

After the margin has been hidden, this menu item toggles to **Show Margin**.



---

## Chapter 52. Resize Window

Select **Resize Window** to control the size of a Shell part.

### Menu summary

#### **Preferred Extent**

Changes the size of parts within the Shell to the smallest size allowable for each part, taking into account any property settings that affect displayed size.

#### **Minimum Extent**

Changes the size of parts within the Shell to the absolute smallest size allowable for each part.

One part that is affected by the difference between these two settings is Entry Field, because you can set the *columns* property to change the number of characters that the field is supposed to handle.



---

## Chapter 53. Open <BoxClassName> Settings

Select **Open <BoxClassName> Settings** to display settings for the parent container (for example, Box) of the cell that is currently selected.

Contrast with **Open Settings**. Within a Box, selecting **Open Settings** displays settings for the cell that is currently selected.



---

## Chapter 54. Align

Select **Align** to change the orientation of the contents of a Box Cell part.

### Menu summary for horizontal alignment

- Left
- Center
- Right
- Expand

### Menu summary for vertical alignment

- Top
- Center
- Bottom
- Expand

For either type of alignment, selecting **Expand** causes the part to enlarge to the fullest extent possible in that dimension.





---

## Chapter 55. Hide Box Margin/Show Box Margin

Select **Hide Box Margin** to render the dotted outline of the selected part invisible.

This menu item is available for Box and related parts, such as Horizontal Box, Vertical Box, ToolBar, and Split Pane.

After the margin has been hidden, this menu item toggles to **Show Box Margin**.



---

## Chapter 56. Row

Select **Row** to change the number of Box Cell rows in a Box part.

### Menu summary

#### **Add Before**

Inserts a row immediately above the selected Box Cell part.

#### **Add After**

Inserts a row immediately below the selected Box Cell part.

**Delete** Removes the row indicated by the selected Box Cell part.



---

## Chapter 57. Column

Select **Column** to change the number of Box Cell columns in a Box part.

### Menu summary

#### **Add Before**

Inserts a column immediately to the right of the selected Box Cell part.

#### **Add After**

Inserts a column immediately to the left of the selected Box Cell part.

**Delete** Removes the column indicated by the selected Box Cell part.



---

## Chapter 58. Switch to page

Select **Switch to page** to change the page being displayed in a Notebook or Pagebook part. This option is available only if at least one page exists in the Notebook or Pagebook part. **Switch to page** does not affect the order in which pages occur in the page stack.

Select another page from the cascade menu provided. A check mark indicates the page currently being displayed. Each item in the cascade menu is formatted as follows: *label — partName (currentOrder)*. For example, suppose the second page (named ConfigurationPage) is labeled Configuration. This item appears in the cascade menu as **Configuration — ConfigurationPage (2)**.





---

## Chapter 59. Font prompt

Use this window to change the font used for the selected part.

### Fields in this window

#### **Font size**

Size in points, ranging from 7 to 50. The system's default setting is indicated by **<default>**

#### **Font name**

Type of font. Valid selections include **Dialog**, **Monospaced**, **Sans Serif**, and **Serif**. The system's default setting is indicated by **<default>**

To specify bold or italic face, select the appropriate check box.

To reset values in the prompter window to the system defaults, select the **Default** button.



---

## Chapter 60. Icon property

Use this window to assign a GIF image to be displayed in the selected part. Click **Select File** to browse the file system through a standard file dialog. When you select a file from the dialog, the dialog closes, and the icon appears in the **Preview** pane. If the icon is acceptable, select **OK** to return to the settings view.



---

## Chapter 61. Converter prompt

Use this window to change the converter used to display values in the selected part. The default setting for the part is indicated by **<default>**

To assign another converter, select one from the **Converter type** list. If the selected converter has settings of its own, the **Settings** button to the right is enabled.



---

## Chapter 62. Accelerator prompt

Use this window to set a mnemonic key sequence for the selected part. To set the accelerator, select items from the fields listed below. Then select **OK** to return to the settings view.

### Fields in this window

#### **Modifiers**

Helper key used with the mnemonic. Valid choices are **Alt**, **Ctrl**, and **Shift**. By default, none of these are checked; however, you must check one in order to select a mnemonic key.

#### **Key select**

Mnemonic key. Valid choices include the letters A–Z, the numerals 0–9, the function keys F1–F35, and others. By default, the list is set to **None**.





---

## Part 4. Appendixes



---

## Index

### Special Characters

#debugPrintString, configuring 72

## A

Application Controller, running 17  
applications, deploying 49

## B

border layout, description 21  
Box part  
    description 93  
    example of using visually 37  
    layout and 21  
Browser Context part 97  
Button part 99

## C

CheckBox Menu Item part 103  
CheckBox part 101  
Column part  
    description 105  
    example of using visually 42  
ComboBox part 107  
command options, UI Engine 14  
command prompt, running ULC  
    applications from 52  
common widget (CW) protocol,  
    comparison with ULC 19  
composite parts 40  
configuring #debugPrintString 72  
connections, comparison with standard  
    VisualAge 33  
Container part 109  
context, customizing exception handling  
    by 74  
contexts named in ULC 71  
converters, use of in ULC 33

## D

Data Model 111  
debugger, using with ULC 72  
deploying ULC-based applications  
    extending environment size in  
        Windows 95 49  
    packaging in XD 49  
    production mode from a development  
        image 51

## E

Enabler part 113  
enablers  
    example of using 39  
    general description 33

Entry Field part 115  
    description 115  
environment size, extending in Windows  
    95 49  
exception handling, customizing 74

## F

faceless half object  
    description 3  
    implementing 57  
Filler part 119  
Form Model part  
    description 121  
    example of using visually 41  
frequently asked questions 77

## G

GroupBox part  
    description 123  
    example of using visually 37

## H

half object, description 3  
Html Pane part 125  
HTTP server  
    running 15  
    setup 10

## I

image, tracing inside 72  
implementing  
    faceless half object 57  
    national language support 31  
    UI half object 55  
internationalization 30

## J

Java support, setup 9

## L

Label part 127  
layout  
    border, description 21  
    box, description 21  
    design tips 22  
    pile, description 22  
    properties in visual composition 35  
    summary of types 20  
    widgets that implement 23  
List part  
    description 129  
    example of using visually 39

## M

Menu Item part 137  
Menu part 133  
Menu Separator part 139  
Menubar part 135  
model-based widgets  
    architecture 24  
    using with model parts 25  
model parts  
    examples of using visually 40  
    general description 25  
Multi-Line Edit part 115

## N

named ULC contexts 71  
national language support  
    implementing 31  
    restrictions 31  
    use of Java Unicode support 30  
nonvisual parts 40  
Notebook part  
    description 141  
    pile layout and 22

## O

objects, implementing in ULC 55

## P

Page part 143  
Pagebook part 145  
pile layout, description 22  
pitfalls in ULC visual composition 34  
production mode  
    description 5  
    development image, setup 51  
    standalone setup 8  
    web setup 8  
Progress Bar part 149

## R

Radio Group part 153  
RadioButton part 151  
resetting the ULC system 71  
Row Model part 155  
running  
    Application Controller 17  
    summary table 13  
    UI Engine 13  
    ULC applications, command line  
        options 52  
    ULC HTTP server 15

## S

- Server Smalltalk
  - concurrency issues 26
  - process management in business logic 27
  - ULC and 26
- setup
  - development image for production mode 51
  - Java support 9
  - production mode, standalone 8
  - production mode, web 8
  - running application from development image 8
  - ULC HTTP server 10
- Shell part
  - feature description 157
  - general description 24
- Slider part 159
- Split Pane part 161

## T

- Table Model part
  - description 167
  - example of using visually 38, 42
- Table part
  - description 163
  - example of using visually 42
- test mode, description 5
- To-Do List example 36
- ToolBar part 169
- tracing inside the Smalltalk image 72
- Tree Model part
  - description 173
  - example of using visually 42
- Tree part
  - description 171
  - example of using visually 43
- TreeTab part
  - description 175

## U

- UI Engine
  - command options 14
  - definition 3
  - description 5
  - javadoc 55
  - running as applet 13
  - running as helper 14
  - running standalone 13
- UI half object
  - description 3
  - implementing 55
- ULC and Server Smalltalk 26
- ULC Monitor 15
- ULC objects, implementing 55
- ULC parts
  - Box 93
  - Browser Context 97
  - Button 99
  - CheckBox 101
  - CheckBox Menu Item 103
  - Column 105
  - ComboBox 107
  - Container 109
  - Data Model 111
  - Enabler 113
  - Entry Field 115
  - Filler 119
  - Form Model 121
  - GroupBox 123
  - Html Pane 125
  - Label 127
  - List 129
  - Menu 133
  - Menu Item 137
  - Menu Separator 139
  - Menubar 135
  - Multi-Line Edit 115
  - Notebook 141
  - Page 143
  - Pagebook 145

### ULC parts (*continued*)

- Progress Bar 149
- Radio Group 153
- RadioButton 151
- Row Model 155
- Shell 157
- Slider 159
- Split Pane 161
- Table 163
- Table Model 167
- ToolBar 169
- Tree 171
- Tree Model 173
- TreeTab 175
- ULC system, resetting 71
- UlcAlert class 91, 111
- UlcAndEnabler class 113
- UlcFont class 87
- UlcIcon class 87
- UlcMandatoryAndEnabler class 113
- UlcNotEnabler class 113
- UlcOrEnabler class 113
- UlcOXrEnabler class 113
- UlcProxy class 89
- UlcRGBColor class 87
- Unicode, support for in UI half objects 30

## V

- Variable parts 40
- visual composition pitfalls in ULC 34
- visual parts, description 34

## W

- widgets, general description 20
- widgets, model-based 24
- Windows 95, extending environment size 49