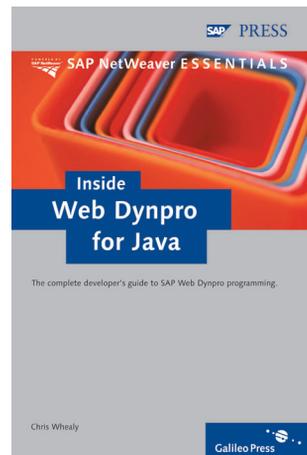


Chris Whealy

Inside Web Dynpro for Java



SAP PRESS

*Out of intense complexities,
intense simplicities emerge.*

Sir Winston Churchill

*The voice of ignorance speaks loud and long,
But the words of the wise are quiet and few.*

Ancient Proverb

Contents

Preface	15
Target audience	15
Author's Apology	15
Conventions	17
Errata	17
Acknowledgements	17
1 Introduction	19
1.1 What is Web Dynpro?	19
1.2 What is the design philosophy behind Web Dynpro?	19
1.3 Improving the user's experience	19
1.4 Building a high-fidelity Web user interface	20
1.5 How is Web Dynpro different from other Web development tools?	20
1.6 What is the underlying design concept?	21
1.6.1 The roots of Web Dynpro—Model-View-Controller	21
1.6.2 The model	21
1.6.3 The view	22
1.6.4 The controller	22
1.6.5 A more detailed description	23
2 Designing a Web Dynpro application	27
2.1 Analysis phase	27
2.2 Design phase	27
2.2.1 Architecture design	29
2.2.2 Detail design	30
2.3 Implementation phase	31
3 General architecture	33
3.1 Web Dynpro Framework	33
3.2 Application	33
3.2.1 Definition	33

3.2.2	Startup	34
3.2.3	Properties	34
3.2.4	Parameters	34
3.2.5	Shutdown	36
3.3	Component	36
3.3.1	Lifespan	37
3.3.2	Interaction	37
3.3.3	Reuse	38
3.4	Controller	38
3.4.1	Controller types	38
3.4.2	Interaction	40
3.4.3	Structure	41
3.5	View	43
3.5.1	Lifespan	44
3.5.2	Interaction	45
3.5.3	Layout	45
3.5.4	Navigation	46
3.6	Model	46
3.7	Event handling	46
3.7.1	Actions	47
3.7.2	Events	48

4 Web Dynpro naming placeholders 49

4.1	Development entities	49
4.2	Context entities	50
4.3	Generic and composite abbreviations	50
4.4	Subscripts for composite placeholders using the SAP recommended suffixes 51	
4.5	J2EE Engine placeholders	51

5 Naming conventions 53

5.1	General rules for naming	53
5.1.1	Permitted characters	53
5.1.2	Length of entity names	53
5.2	Naming conventions for coding entities	56

6 The context 61

6.1	Context structure at design time	61
6.1.1	Nodes	61
6.1.2	Attributes	62

6.1.3	Terminology concerning nodes and attributes	62
6.1.4	Relationship between nodes and attributes	62
6.1.5	Calculated attributes	63
6.1.6	The element collection	64
6.1.7	An element's selection status	65
6.1.8	Summary of the context at design time	65
6.2	Context structure at runtime	66
6.2.1	The cardinality property	66
6.2.2	The singleton property	67
6.2.3	Selection cardinality	69
6.2.4	Context attributes that can supply data to UI elements	70
6.2.5	Summary of the context at runtime	71
6.3	Should node names be singular or plural?	72
6.4	General naming standards	73
6.4.1	Value nodes: {vn}	73
6.4.2	Value attributes: {va}	74
6.4.3	Model nodes: {mn}	74
6.4.4	Model attributes: {ma}	74
6.4.5	Recursive nodes: {rn}	75
6.4.6	Supply functions: supply{cn}()	75
6.5	Classes generated as a result of design time declarations	76
6.6	What's the difference between the various types of context nodes? ...	77
6.6.1	Value nodes	77
6.6.2	Model Nodes	78
6.6.3	Recursive Nodes	80
6.7	Fundamental principles of the context	82
6.8	Context mapping	84
6.8.1	What is mapping?	84
6.8.2	Selection mapping	85
6.8.3	Normal mapping	85
6.8.4	External mapping	86
6.8.5	General points about external mapping	89
6.8.6	What impact does context mapping have on my coding?	89
6.9	Accessing the context through the typed API	91
6.9.1	Accessing a node element	92
6.9.2	Accessing all elements in a node	93
6.9.3	Creating a new node element	94
6.9.4	Treating node Lineltems as a singleton node	96
6.9.5	Treating node Lineltems as a non-singleton node	99
6.10	Dynamic context manipulation	103
6.10.1	Accessing the context through the generic API	103
6.10.2	Dynamic addition of an unmapped context node	108
6.10.3	Dynamic addition of a mapped context node	113

7 Coding principles in Web Dynpro 125

7.1	User-defined code	125
7.2	Problems with binding UI elements to context nodes	125
7.3	Building a context node hierarchy suitable for a tree UI element	127
7.4	Parameter mapping	131
7.4.1	Basic parameter mapping example	133
7.4.2	Action handler generalization	133
7.4.3	Further decoupling of the UI	135
7.4.4	Advanced parameter mapping example	136
7.5	Efficient use of actions to enable and disable UI elements	138
7.6	Layout managers	139
7.6.1	Flow layout	139
7.6.2	Row layout	140
7.6.3	Matrix layout	141
7.6.4	Grid layout	142
7.6.5	Layout Manager Properties	144
7.7	Principles for the efficient use of layout managers	146
7.8	Locale-specific text	147
7.8.1	Introduction to multilingual support in software products	147
7.8.2	Internationalization	147
7.8.3	Externalization	147
7.8.4	Web Dynpro i18n concept	148
7.8.5	S2X—SAP's use of the XLIFF standard	149
7.8.6	Storing language specific text in .XLF files	150
7.8.7	Translating XLF files	151
7.8.8	Use of the S2X editor within NWDS	151
7.8.9	Editing MessagePool XLF files	152
7.8.10	Runtime Locale Identification	154
7.8.11	Locale-dependent text at runtime	155
7.8.12	Defining placeholders within a message text	157
7.9	Accessing any parameter in the query string	158

8 Dynamic UI generation 161

8.1	General points about Web Dynpro user interfaces	161
8.2	Background information about the design of UI elements	162
8.3	Accessing existing UI elements in a view layout	163
8.3.1	Accessing UI elements by name	164
8.3.2	Accessing UI elements generically	165
8.4	The principles of dynamic view construction	173
8.4.1	Before you start... ..	173
8.4.2	The fundamental principles of dynamic view layout construction	173
8.4.3	Controlling UI element behavior from the context	175
8.5	Dynamic Construction of a UI element hierarchy	176

9 The Common Model Interface1 189

9.1	CMI terms and definitions	189
9.1.1	Model	189
9.1.2	Model class	190
9.2	Access from a CMI object to the underlying business logic	191
9.2.1	Typed access	191
9.2.2	Generic access	191

10 The Adaptive RFC layer 193

10.1	General introduction to BAPIs	193
10.1.1	What is a BAPI?	193
10.1.2	The interface of an ABAP function module	194
10.2	Custom written RFC modules	196
10.3	Background to the adaptive RFC layer	198
10.3.1	The Java Connector (JCo)	198
10.3.2	The Enterprise Connector	199
10.4	Introduction	201
10.5	Explanation of Generated Model Classes	204
10.5.1	Model object hierarchies	206
10.5.2	Executable and non-executable model objects	207
10.6	Using model objects in a Web Dynpro controller	207
10.6.1	Model objects and the context	208
10.6.2	The relationship between the contents of a model object and its corresponding context model node	210
10.7	A simple example using context model nodes at runtime	211
10.7.1	Preparing the context at design time	212
10.7.2	View controller coding	222
10.7.3	Look at what has not been done	229
10.8	Adapting to changes in an RFC interface	230
10.9	Connection management	238
10.9.1	Language	239
10.9.2	Client	239
10.9.3	How does a model object know which SAP system to connect to?	240
10.10	Relationship between JCO destinations and ABAP sessions	247
10.10.1	Logging on to an SAP system	247
10.10.2	ABAP sessions	247
10.10.3	The impact of an ABAP session change on an external program using the JCo layer	249
10.10.4	How do I perform ABAP database updates without causing a session change?	250
10.10.5	Can I use any statements I like in an RFC module?	251

10.11	Avoiding the Read-Write-Read problem	252
10.12	Why can't I use two different JCo connections?	255

11 Web Dynpro phase model 257

11.1	Transport data to DataContainer	258
11.2	Transport modified data into context	258
11.3	Validate modified data	258
11.4	Call system event handler	259
11.5	Call service event handler	259
11.6	Application event handler	260
11.7	doBeforeNavigation	260
11.8	Navigation and view initialization	260
11.9	Dynamic UI manipulation	261
11.10	doPostProcessing	261
11.11	Response rendering	262

12 Class and Interface Reference 263

12.1	Controllers, their methods, and self reference	263
12.1.1	Controller constructors	263
12.1.2	Standard hook methods	263
12.1.3	User-defined methods	264
12.1.4	Self reference and shortcuts	265
12.1.5	User modifications to generated code	266
12.2	Controller classes	266
12.2.1	{n _c }—Component controller	266
12.2.2	{n _c }Interface—Component interface controller	267
12.2.3	{n _v }—View controller	268
12.2.4	Internal{n _{ctl} }, Internal{nc}Interface—Generated controllers	269
12.3	Controller interfaces	271
12.3.1	IExternal{n _{ctl} }Interface—External public interface controller	271
12.3.2	IPublic{n _{ctl} }—Internal public interface controller	272
12.3.3	IPrivate{n _{ctl} }—Internal private interface controller	272
12.3.4	IWDComponent—Generic interface for all component controllers	273
12.3.5	IWDComponentUsage—Generic component usage interface	274
12.3.6	IWDController—Generic interface for all controllers	277
12.3.7	IWDView—Generic interface for a view layout	277
12.3.8	IWDViewController—Generic interface for all view controllers	278
12.4	Application interfaces	279
12.4.1	IWDApplication—Generic interface for all applications	279

12.5	Context interfaces	279
12.5.1	I{c _n }Element—Node specific extension to IWDNodeElement	279
12.5.2	I{c _n }Node—Node specific extension to interface IWDNode	280
12.5.3	IContextElement—Controller specific extension to IWDNodeElement	281
12.5.4	IContextNode—Controller specific extension to IWDNode	282
12.5.5	IWDAttributeInfo—Interface for the metadata of a generic context attribute	283
12.5.6	IWDContext—Base interface for accessing context data in a controller	284
12.5.7	IWDNode—Interface implemented by all context nodes	286
12.5.8	IWDNodeElement—Interface implemented by all context node elements	291
12.5.9	IWDNodeInfo—Interface for the metadata of a generic context node	293
12.6	View layout interfaces	299
12.6.1	Generalized management of UI element properties	299
12.6.2	Generalized management of UI element actions	300
12.6.3	Appearance of UI elements	301
12.6.4	The UI element source property	302
12.6.5	Naming of UI element properties	302
12.6.6	IWDAbstractButton—Base interface for a push button	302
12.6.7	IWDAbstractCaption—Base interface for a text caption	303
12.6.8	IWDAbstractDropDown—Base interface for a drop-down list	303
12.6.9	IWDAbstractDropDownByIndex—Base interface for an index-based drop-down list	304
12.6.10	IWDAbstractDropDownByKey—Base interface for a key-based drop-down list	305
12.6.11	IWDAbstractInputField—Base interface for an input field	306
12.6.12	IWDAbstractTreeNodeType—Base interface type for all tree nodes and items	307
12.6.13	IWDButton—Interface for a push button	309
12.6.14	IWDCaption—Base interface for a text caption	309
12.6.15	IWDCheckBox—Interface for a check box	310
12.6.16	IWDCheckBoxGroup—Interface for a multiple checkbox selection group	311
12.6.17	IWDDropDownByIndex—Interface for an index-based drop-down list	312
12.6.18	IWDDropDownByKey—Interface for a key-based drop-down list	312
12.6.19	IWDFlowData—Interface for flow layout data	313
12.6.20	IWDFlowLayout—Interface for the flow layout manager	314
12.6.21	IWDGridData—Interface for grid layout data	314
12.6.22	IWDGridLayout—Interface for the grid layout manager	316
12.6.23	IWDIFrame—Base interface for an HTML IFrame	316
12.6.24	IWDImage—Interface for an image	317
12.6.25	IWDInputField—Interface for an input field	318
12.6.26	IWDLabel—Interface for a text label	318
12.6.27	IWDLayout—Base interface for all layout managers	319
12.6.28	IWDLayoutData—Base interface for all layout data	319

12.6.29	IWDLink—Base interface for a generic hypertext link	320
12.6.30	IWDLinkToAction—Interface for a hypertext link to a Web Dynpro action	321
12.6.31	IWDLinkToURL—Interface for a hypertext link to a URL	321
12.6.32	IWDMatrixData—Interface for matrix layout data	322
12.6.33	IWDMatrixHeadData—Interface for matrix layout head data	323
12.6.34	IWDMatrixLayout—Interface for the matrix layout manager	324
12.6.35	IWDRadioButton—Interface for a radio button	324
12.6.36	IWDTab—Interface for an individual tab page	326
12.6.37	IWDTabStrip—Interface for an aggregation of tab pages	327
12.6.38	IWDTable—Interface for a table	328
12.6.39	IWDTableColumn—Interface for a table column	332
12.6.40	IWDTextEdit—Interface for a multi-line text editor	334
12.6.41	IWDTextView—Interface for a read-only text display	335
12.6.42	IWDTree—Interface for the root node of a tree	336
12.6.43	IWDTreeItemType—Interface for a tree item	338
12.6.44	IWDTreeNodeType—Interface for a tree node	338
12.6.45	IWDUIElement—Base interface for UI elements	339
12.6.46	IWDUIElementContainer—Base interface for a container UI element	340
12.6.47	IWDViewElement—Base interface of all UI elements	342

A	ABAP coding	343
----------	--------------------	------------

B	Dictionary structures	347
----------	------------------------------	------------

C	The Author	349
----------	-------------------	------------

	Index	351
--	--------------	------------

Preface

Target audience

This book has been written for those people who have:

- ▶ Java programming experience
- ▶ Attended the standard SAP training course(s) for Web Dynpro for Java programming

This book is not a tutorial of "how to" style exercises and answers, but rather it discusses the design and coding principles required for the development of successful Web Dynpro applications. The focus of this book is the core of Web Dynpro technology.

Certain Web Dynpro related subjects are not covered in this edition because they are not fundamental to your understanding of the subject. This book is designed to lay a foundation upon which other publications can then build.

Knowledge of SAP's ABAP programming language would be beneficial (particularly when reading Chapter 10), but is not essential.

Author's Apology

Having been a technical SAP consultant since 1993, and a professional software developer since 1986, I have learned¹ that forming a rigid set of *rules* about how problems should be solved has a fundamental weakness. That is, the rules formed for solving problem A often cannot be applied to solving problem B, even though problems A and B are similar.

Instead, I have found that many diverse and seemingly unrelated problems can be solved using a common set of problem-solving *principles*. The problem-solving process then starts with an analysis of the particular situation to identify which of these principles are applicable.

This process of problem-solving is much like the process of abstraction that takes place during the design of an object-oriented program—that of condensing a problem down to its most fundamental, often abstract, elements—and I have spent much of my professional career developing these principles.

¹ By attending the school of hard knocks!

In writing this book, I do not wish to lay down an inflexible set of rules to which all developers should conform. Instead, I aim to equip the reader with an understanding of the fundamental principles that must be understood in order to build powerful and efficient Web Dynpro applications.

Since Web Dynpro is a programming toolset, it is not immune from abuse; in the areas in which problems could arise, it may appear that I am laboring the point. If I have described the same concept in several different ways, it is because I am stressing its importance. I trust that this approach will serve to communicate the necessary understanding to all readers, and not be too tedious for those people who "always grasp things the first time."

As with any set of tools, the results obtained from the use of Web Dynpro are determined by the skill of the operator, not the brand name on the handle! Therefore, to achieve the best results from the Web Dynpro toolset, each developer must have a thorough grasp of the principles that underpin its design and operation.

I leave it to each developer to apply these principles to his or her own specific situation. Using this approach, no two solutions will ever be the same, yet all will be derived from a common set of fundamental principles.

It is these principles that I aim to communicate in this book.

Note to readers with ABAP development experience

For those experienced with classical SAP software design, the design concepts used by Web Dynpro represent an entirely new way of thinking, the mastery of which will require a significant shift in your thinking. For this category of reader,² it is even more important that these principles are fully grasped and understood.

Please allow this book to alter and expand your thinking, and guide you through the mental transition that is required to move from classical R/3 design to Web Dynpro design. Failure to realize that such a transition is required will cause you to become frustrated by the fact that Web Dynpro does not meet your established expectations. This, in turn, can lead to all sorts of erroneous conclusions about the quality of the product!

² In which I include myself.

Conventions

Use of terminology

Certain words such as "component," "element," "context," and "interface" have specific meanings within the Web Dynpro context (sic!). Therefore, to avoid ambiguity, such words will be used only when their Web Dynpro meaning is intended.

Screenshots

For the sake of brevity, various graphical figures in this book have been cropped or resized. Therefore, when you look at the corresponding screen in your installation of the SAP NetWeaver Development Studio (NWDS), it may be larger than the image displayed in this book.

Errata

For those readers with access to SAP's Online Support System (OSS), please check note number 699531 for any corrections to errors or omissions discovered after publication.

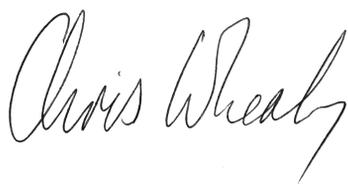
For readers who do not have access to the OSS system, please check the SAP PRESS Web sites www.sap-press.com and www.sap-press.de for corrections, omissions, or additional content that may be delivered after the publication of this book.

Acknowledgements

The author would like to thank everyone on the SAP Web Dynpro development team for their enthusiastic help and support while this document was being written. They all managed to find a few spare clock cycles in their very busy schedules to proofread and correct the many iterations through which this document passed before finally ending up on the printed page.

These are Björn Goerke, Stephan Ritter, Johannes Knöppler, Markus Cherdrön, Jens Ittel, Uwe Reeder, Harry Hawk, Thomas Chadzelek, Bertram Ganz, Arnold Klingert, Joseph Brown, Stephan Dahl, Werner Bächle, Harry Hawk, Malte Wedel, Timo Lakner, Jörg Singler, Thorsten Dencker, Stefan Beck, Reiner Hammerich, Armin Reichert and Harry Hawk.

I would also like to thank Simon Harbour, Karl Kessler, Masoud Aghadavoodi Jolfaei, Markus Tolksdorf, Peter Tillert, Carsten Brandt, Marco Ertel, Karin Schattka, and Peter Barker for their input and support during the writing of this book.

A handwritten signature in black ink that reads "Chris Whealy". The signature is written in a cursive, flowing style with a long, sweeping tail on the final letter.

Chris Whealy,
November 2004

7 Coding principles in Web Dynpro

7.1 User-defined code

Within each controller, you are free to add any code you require as long as it lies between a pair of `//@@begin` and `//@@end` comment markers. These markers are created automatically by the code generator within the NWDS and their locations cannot be changed.

Any user-defined code placed outside these special comment markers will be lost during code regeneration.

7.2 Problems with binding UI elements to context nodes

When a view layout is being designed, every interactive UI element must have a context binding of some sort. There are certain runtime situations that may occur in which your input fields may appear in a disabled state or you get a runtime error in the context, such as `Node(<some_node_name>): no active node to map to OR Mapping reference not found`.

All three situations are caused by the fact that, somewhere in the context, there is either a non-existent node element or a non-existent node.

- ▶ The simplest situation is the one in which a UI element, such as an input field, is bound to an attribute of a value node of cardinality `0..n`. The first part of the cardinality¹ immediately tells you that when your application first starts, this node will contain zero elements.

If, prior to displaying this screen, you forget to create element zero in the context node, then any input fields bound to attributes in this node will be disabled. This is simply because the node to which the UI element is bound contains no elements; therefore, there is no storage area to receive the user's input. Consequently, the input field will be disabled.

- ▶ The next situation is slightly more detailed. Imagine your view controller context has an unmapped value node of cardinality `0..n` called `SalesOrders`. This node also has a child node called `LineItems` (also of cardinality `0..n`). Remember that the cardinality of node `SalesOrders` will cause it to start life as an empty collection; i.e., the child node

¹ Call method `node{cn}.getNodeInfo().isMandatory()` to obtain this value.

`LineItems` will not even exist until such time as element zero of its parent node (`SalesOrders`) is created.²

At design time, it is perfectly valid to create UI elements in the view layout that are bound to attributes of the child node `LineItems`. However, at runtime, you must ensure that node `SalesOrders` contains at least one element. This element can then act as the parent for the child node `lineItems`.

If you forget to do these data-creation steps yourself, you will end up trying to display the value of a non-existent attribute, belonging to a non-existent element of a non-existent child node! Hence the error, `Node(...): no active node to map to`.

- The last situation involves mapping context nodes between different controllers. Imagine that the value nodes for `SalesOrders` and `LineItems` now live in the context of a custom controller (such as your component controller) and not the view controller. Your view controller can gain access to this data by declaring a mapping relationship between its own value node³ and the corresponding value nodes in the component controller.⁴

If you now create UI elements in the view layout that are bound to the view controller's `SalesOrders` node, the following situation could cause an error to occur:

Again, in the component controller, you forget to add any elements to the `SalesOrders` value node and you completely forget to create the value node `LineItems`. The difference now is that processing to add the data to these node elements is the responsibility of the component controller. The view controller is simply referencing the element collection that exists in the component controller. Mapped nodes do not maintain their own element collections.

Therefore, in order to find the correct value to display in the UI element, a mapping reference from the view controller to the component controller must be traversed. The mapping reference to the child node

-
- 2 This is a slight oversimplification. If a child node `{chn}` is a singleton with respect to its parent node `{cn}`, then `{chn}` can be created, but cannot be populated until `{cn}` has at least one element. If, however, the child node `{chn}` is a non-singleton with respect to its parent node `{cn}`, then `{chn}` cannot even be created until `{cn}` contains at least one element.
 - 3 The value node in the view controller does not need to have the same name as the value node in the custom controller to which it is mapped. However, it helps greatly with code legibility if the two names are the same.
 - 4 This type of mapping is possible only after you have explicitly stated that the component controller is a required controller for the view controller.

`LineItems` will return a null pointer exception, and this in turn is interpreted as the error `Mapping reference not found`.

See Section 6.8 for more details.

7.3 Building a context node hierarchy suitable for a tree UI element

To make correct use of the tree UI element, it is most important to understand the requirements this UI element imposes on the view controller's context. As has been stated earlier, the structure of data supplied by a model object is rarely suitable for immediate display on the user interface. This is particularly true when using the tree UI element, which requires the context data to be structured in a very particular way. It is almost a certainty that a model object will *not* supply data in this specialized structure!

Here is a perfect example of where a custom controller (such as the component controller) should perform a structural data transformation on the context data to make it suitable for requirements of the UI elements on a view layout.⁵

The properties of trees and tree nodes can be found in classes `IWD-AbstractTreeNodeType`, `IWDTreeNodeType`, and `IWDTree`.

The following tree and tree node properties can be bound to context attributes:

Tree	TreeNode
<code>defaultItemIconAlt</code>	<code>iconAlt</code>
<code>defaultItemIconSource</code>	<code>iconSource</code>
<code>defaultNodeIconAlt</code>	<code>ignoreAction</code>
<code>defaultNodeIconSource</code>	<code>Tooltip</code>
<code>minHeight</code>	<code>Text</code>

Table 7.1 Bindable Properties of Tree UI Elements

⁵ It can also be argued that a view controller should receive the raw data supplied by a model object and then perform its own transformation, thus relieving the custom controller of the need to know anything about the requirements of any particular view layout. Both approaches are plausible, and the decision to use one method over another should be judged both on the complexity of the transformation process and the number of views that require the same restructured model data.

Tree	TreeNode
rootText	Design
rootVisible	dataSource
title	expanded
defaultNodeIconSource	tooltip
minHeight	text
rootText	design
rootVisible	dataSource
title	expanded
titleVisible	hasChildren
width	
dataSource	

Table 7.1 Bindable Properties of Tree UI Elements (cont.)

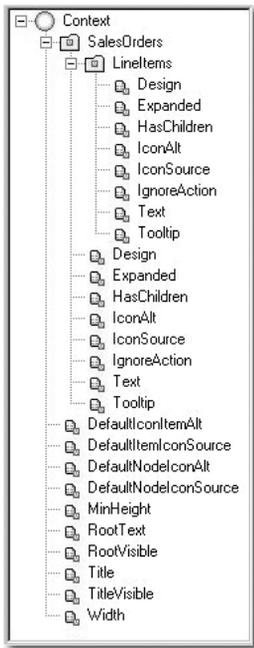


Figure 7.1 A Context Structure Suitable for a Tree UI Element

Depending on your business requirements, it may not be necessary to bind all the properties listed in Figure 7.1, but for the purposes of this

illustration, it has been assumed that you want to have full, programmatic control over every aspect of the tree's appearance.

If we assume that we want to display a list of sales orders as a tree, the view controller's context will look something like Figure 7.1.

Important: Notice that the names of the node attributes in the context reflect the properties of the tree UI elements, *not* the attributes found in a sales order. In fact, the only attributes that have anything to do with the business data being displayed are `Text` and `Tooltip`. All the other attributes control the appearance and behavior of the tree itself.



We start with the independent node (`SalesOrdersForTree`) to represent the tree UI element for sales orders. Under this, there is a dependent node (`LineItemsForTree`) to represent the tree UI element for line items.

In this example, the properties belonging to the entire tree UI element have been created as independent attributes because there is only one tree in this view.⁶

The properties belonging to the tree node UI elements have been created as dependent attributes of node `SalesOrdersForTree` and `LineItemsForTree`.

It is *vital* important that the singleton property of node `LineItemsForTree` is set to `false`! Think about how a tree UI element is capable of displaying its information, and then think about the implications of having the child node `LineItemsForTree` as a singleton node.

If you haven't figured it out, let me explain. First, consider the data held in the element collection of node `SalesOrdersForTree`. In business terms, it is a collection of sales order headers. Under this is the node `LineItemsForTree`. This node holds the line items for each sales order. In other words, there must be a whole new instance of child node `LineItemsForTree` for each element of node `SalesOrdersForTree`.

Now consider how information can be presented through a tree structure. The parent node of a tree may have, let's say, five children. This corresponds to the node `SalesOrdersForTree` having five elements in its element collection. A tree UI element allows you to expand each one of

⁶ If you have multiple tree UI elements in the same view layout, then you would need to create a parent node (of cardinality 1 . . 1) for each tree.

these five child nodes and see all their children simultaneously. This immediately implies that each element of node `SalesOrdersForTree` must have its own distinct instance of child node `LineItemsForTree` in existence simultaneously.

Therefore, node `LineItemsForTree` must be a non-singleton with respect to `SalesOrdersForTree`; otherwise, you would only ever be able to see the children of the one element at the lead selection of node `SalesOrdersForTree`.

As you should now be able to appreciate, this very small configuration step has an enormous impact on the behavior of a context node. This, in turn, can make or break your view layout!

To create the corresponding tree UI elements in a view layout, proceed as follows:

1. Create the context structure shown above in Figure 7.1 — not forgetting to set the singleton flag of node `LineItemsForTree` to `false`.
2. Create a Tree UI element in your layout with a name such as `SalesOrderTree`.
3. Bind the various properties of the UI element `SalesOrderTree` to the appropriate independent attributes.
4. Bind the `dataSource` property of UI element `SalesOrderTree` to the context node `SalesOrdersForTree`.
5. Add two `TreeNodeType` elements as children of `SalesOrderTree`, calling them something like `SalesOrders` and `LineItems`.
6. Bind the `dataSource` property of the `TreeNodeType` `SalesOrders` to context node `SalesOrdersForTree`.
7. Bind the `dataSource` property of `TreeNodeType` `LineItems` to the context node `LineItemsForTree`.
8. Bind the required properties of each `TreeNodeType` UI element to the appropriate context attributes.

A common mistake is to think that the structure of the `TreeNodeType` UI elements under the tree UI element should reflect the structure of the data being displayed. No! This is not necessary. All you need to do is declare that various nodes will exist somewhere in the tree hierarchy. The response rendering stage of the phase model (see Section 11.11) will then automatically determine the hierarchical position of any given node on the basis of its `dataSource` binding.

All that is required now is that functionality exists⁷ that can transform the business data into the structure required by the tree UI element.

To make the tree interactive, it will be necessary to implement the `onAction` event, and possibly also the `onLoadChildren` event.⁸

7.4 Parameter mapping

To ensure that server-side controllers can react intelligently to user actions on the client, it is often necessary to associate parameters with certain client-side events. For instance, when the user selects an item from a dropdown list, the server-side controller needs to know more than just the fact that *an* item has been selected; it needs to know exactly *which* item has been selected. Therefore, when a dropdown list fires its client-side `onSelect` event, the index of the selected item can be received by the server-side action handler.

Important: Event parameter names are hard coded within each UI element.



If an event has an associated event parameter, the UI element will automatically place a value into the event parameter. This part of the coding is done for you automatically; however, you must ensure that the value of the client-side event parameter is received by the server-side action handler.

In the case of the `DropDownByIndex` UI element, the hard-coded parameter is called `index`. You must now add the coding that retrieves the event parameter and passes it to your action handler. This is known as parameter mapping, and is done as follows:

- ▶ Obtain the name of the parameter associated with the client-side event. This can be found by looking in the javadoc of the relevant UI element. The parameter can be found in the comment above method `mappingOf(uiEvt)()`; i.e., method `IWDCheckBox.mappingOfOnToggle()` has a boolean parameter called `checked`.
- ▶ Create an action `{act}` in the view controller.

⁷ Either in the view controller itself, or the custom (component) controller.

⁸ The `onLoadChildren` event allows you to calculate what the children of a specific node will be at the time the user *first* expands the node.

- ▶ Define a parameter for the action handler of the same data type as the event parameter.⁹ You will often find it helpful to make the server-side action parameter name the same as the client-side event parameter name, though this is not mandatory.
- ▶ Associate the event parameter with the action parameter.

The UI element shown in Figure 7.2 is a `DropDownByIndex`, but the principles remain the same for all other UI elements:

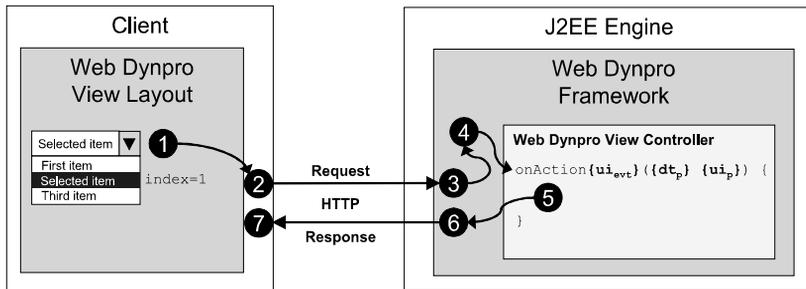


Figure 7.2 How a UI element parameter is passed back to an action handler

1. User selection raises the client-side `onSelect` event belonging to `IWDDropDownByIndex`.
2. An HTTP round trip is invoked to process the associated action handler. All available UI parameter values are passed back to the WDF.
3. On the basis of the source mapping declarations, the WDF matches UI parameters with action event handler parameters.
4. The action handler is invoked using any available parameters that match the source mapping declarations.
5. The action handler reacts to the event in an appropriate manner.
6. Control is passed back to the WDF which builds the HTTP response.
7. The client (in this case, a browser) receives the response and renders the processed screen.

⁹ All event parameter output from the client will be of type `String`. Parameters on action handlers can also be declared to be of type `String`, but if you want to use the data type specific to the event parameter (typically `boolean` or `int`), then you can declare the action parameter to be of the same type as the event parameter, and the WDF will make the conversion automatically.

7.4.1 Basic parameter mapping example

In this example, the `checked` parameter of the `onToggle` event of a checkbox UI element will be associated with a parameter called `checkBoxState` in the corresponding action handler.

- ▶ Create an action in a view controller to handle the change of state in a checkbox UI element. The checkbox is called `myCheckBox` and will be associated with an action called `HandleCheckBox`.
- ▶ Define a boolean parameter called `checkBoxState` for the action handler `onActionHandleCheckBox()`.
- ▶ Place the following coding in the `wdDoModifyView()` method of the view controller. This coding must be executed only *once* during the view controller's lifecycle; therefore, it is imperative that we first check the `firstTime` flag.

```
if (firstTime) {
    // Get a reference to the checkbox UI element
    IWDCheckBox cb = (IWDCheckBox)view.getElement("myCheckBox");

    // Link the client-side event parameter "checked"
    // to the server-side action parameter "checkBoxState"
    cb.mappingOfOnToggle().
        addSourceMapping("checked", "checkBoxState");
}
```

Listing 7.1 Basic Parameter Mapping

Now, whenever this particular checkbox is toggled, the client-side event parameter `checked` that belongs to the `onToggle` event will be passed through to the server-side action handler `HandleCheckBox` as the boolean variable `checkBoxState`.

7.4.2 Action handler generalization

Since the event-to-action parameter mapping is specific to the UI element and not the action, it is perfectly possible to generalize the use of an action handler so that it can respond to events from multiple UI elements. You can extend the first example so that the action handler `HandleCheckBox` can process *any* `onToggle` event from *any* checkbox on the current view.

In order to make an action handler work in this generic manner, you must define an extra parameter for `HandleCheckBox` that identifies *which* checkbox raised the `onToggle` event.

This extra parameter has nothing to do with the client-side event itself; therefore, it is completely independent from the client layer. The following code extends the above coding example.

- ▶ Against the action handler `HandleCheckBox`, define a new parameter called `checkBoxName` of type `String`.
- ▶ The `HandleCheckBox` action handler will now process the `onToggle` events from three different checkboxes.
- ▶ For each checkbox UI element, you must now define a fixed value for the `checkBoxName` parameter. In the following example, we will use three checkboxes that all trigger the same server-side event handler. The coding is as follows:

```
if (firstTime) {
    // Get references to all three checkbox UI elements
    IWDCheckBox cb1 = (IWDCheckBox)view.getElement("checkBox1");
    IWDCheckBox cb2 = (IWDCheckBox)view.getElement("checkBox2");
    IWDCheckBox cb3 = (IWDCheckBox)view.getElement("checkBox3");

    // Link the client-side event parameter "checked" to the
    // server-side action parameter "checkBoxState"
    // This parameter is UI-element specific, and therefore
    // identical for all three checkboxes.
    cb1.mappingOfOnToggle().
        addSourceMapping("checked", "checkBoxState");
    cb2.mappingOfOnToggle().
        addSourceMapping("checked", "checkBoxState");
    cb3.mappingOfOnToggle().
        addSourceMapping("checked", "checkBoxState");

    // Now hard code the checkbox names that enable the server-
    // side event to distinguish between each checkbox.
    cb1.mappingOfOnToggle().
        addParameter("checkBoxName", cb1.getId());
    cb2.mappingOfOnToggle().
        addParameter("checkBoxName", cb2.getId());
    cb3.mappingOfOnToggle().
        addParameter("checkBoxName", cb3.getId());
}
```

Listing 7.2 Generalized Action Handler

7.4.3 Further decoupling of the UI

The degree of abstraction between the event parameter and action parameter can be taken a degree further if desired. Rather than hard coding the specific UI element name into a custom action parameter, you could use a reference to a context attribute.

For instance, consider the following situation. You want to display the contents of a context node as a table, but the number of columns to be displayed is unknown until runtime. You also want to give the user the ability to sort the table simply by clicking on a column header. This type of situation calls for a generic action handler that can sort the table based, not on the name of the table column *UI element*, but on the name of the *context attribute* being visualized by that table column. In other words, the information provided to the sort algorithm needs to be the name of the context attribute that will act as the sort key, *not* the name of the table column UI element that is visualizing the information.

This level of disassociation between the table UI element and the sort algorithm allows you to put any table on the screen, made up of any number of columns of any name, and the sort logic will still function.

In this scenario, the standard client-side event parameter `col` (provided by the `onAction` event of `IWDTableColumn`) is ignored because this contains the name of the UI element on which the user clicked. Instead, the name of the context attribute to which the UI element is bound will be used as the parameter value.

The following example describes a situation in which a table of sales orders is displayed, but the number of columns is configurable and therefore not known until runtime. The user can sort the table by clicking on the header of the column he or she wishes to use as the sort key. The name of the action handler that performs the sort is `HandleSortRequest`, and it receives a single `String` parameter called `colAttrib`.

- ▶ The standard parameter `col` for the table column's `onAction` event must be ignored. This is easy to achieve—do nothing!
- ▶ Create a new parameter called `colAttrib` of type `String` on the action handler `HandleSortRequest`.
- ▶ The attributes that will be displayed as table columns live in a context node called `SalesOrders`. In this rather simplistic example, let's say that the attributes of this node are called `CustId`, `CustName`, and `Date`.

- ▶ In the `wdDoModifyView()` method of the view controller, the following code should be added. In this example, the coding to create the table column UI elements has been omitted, but they are called `tabCol1`, `tabCol2`, and `tabCol3`.

You should associate the context attribute supplying data to the table column with the static parameter `colAttrib`.

```
// Obtain references to the table column UI elements
IWDTableColumn tc1 = (IWDTableColumn)view.getElement("tabCol1");
IWDTableColumn tc2 = (IWDTableColumn)view.getElement("tabCol2");
IWDTableColumn tc3 = (IWDTableColumn)view.getElement("tabCol3");

// Hard code the value of the "colAttrib" parameter to be the dot
// delimited name of the context attribute.
// Notice this is a string value, not an object reference!
tc1.mappingOfOnAction().
    addParameter("colAttrib", "SalesOrders.CustId");
tc2.mappingOfOnAction().
    addParameter("colAttrib", "SalesOrders.CustName");
tc3.mappingOfOnAction().
    addParameter("colAttrib", "SalesOrders.Date");
```

Listing 7.3 Action Handler Using a Context Attribute Name

- ▶ Now when the action handler `HandleSortRequest` is called, it will receive a `String` containing the name of the context attribute that is to be used as the sort key in the string parameter `colAttrib`.
- ▶ The action handler must now use the value of the string parameter `colAttrib` to create a reference to the relevant context attribute. The sort algorithm should then be passed this context attribute reference as its sort key.

7.4.4 Advanced parameter mapping example

When processing the events raised by tree nodes, it is vitally important that the action handler know not only the name of the node on which the user clicked, but also the exact element within that node. Therefore, the event-to-action parameter mapping must be done in the following manner.

- ▶ As with the previous examples, the name of the event parameter raised by the `IWDTreeNodeType` needs to be known. (In the case of this particular interface, the `onAction` event is not defined in `IWDTreeNodeType`, but in the base class `IWDAbstractTreeNodeType`.)

The required event parameter is called `path` and is of type `String`.

- ▶ Before creating the action parameter, you must identify the name of the generated node element class that the `TreeNodeType` UI element represents.

For instance, the element of context node `WBSElements` in view controller `ShowProjectAsTreeView` will be called `IPrivateShowProjectAsTreeView.IWBSElementsElement`.

In general, for any element of node `{cn}` belonging to the context of view controller `{nv}`, the generated class name will always be `IPrivate{nv}.I{cn}Element`.

- ▶ Assuming that the action handler is called `HandleNodeClick`, create a parameter called something like `selectedNodeElement`.

It is most important that the data type of this parameter is *not* `String`! It must be of the data type of the generated node element class identified in the previous step.

- ▶ As in the previous examples, the event parameter must be associated with the action parameter using the following code:

```
IWDTreeNodeType tn = (IWDTreeNodeType)view.  
    getElement("WBSElements");  
tn.mappingOfOnAction().  
    addSourceMapping("path", "selectedNodeElement");
```

Listing 7.4 Action Handler Using a Context Node Reference

- ▶ Now when the user clicks on the displayed node, the client will pass the path name to the context element as a `String` value in the event parameter `path`. Before the value of `path` is transferred to the server-side action parameter `selectedNodeElement`, the WDF recognizes that the parameter to the action event handler is declared as a node element class, and will automatically convert the `String` value held in the event parameter `path` into the object reference required by the action handler.
- ▶ The action handler method now has an object reference to the exact node element on which the user clicked.

7.5 Efficient use of actions to enable and disable UI elements

Certain UI elements trigger client-side events. In order for these events to be processed on the server, there must be an association between the client-side event and an event handler on the server. This association is performed by instances of class `IWDAction`.

Instances of the class are known as actions, and these can be enabled and disabled at runtime as required by the functionality of your application. As you have seen in the previous section, it is possible to have many different client-side events all associated with the same action; thus, they will all trigger the same generic event handler method. At this point, it would be worthwhile to ensure that you fully understand the difference between a primary and secondary event. If you can't remember, go back and reread Section 3.7.1.

If you wish to stop a user from triggering a particular action (for instance, the user has insufficient authorization), the simplest way to achieve this is to disable the action. This is done by calling the action's `setEnabled()` method and passing it the boolean value `false`.

Now the WDF runtime automatically disables or adapts all UI elements using this action. If the action is associated with a primary event, the entire UI element is disabled for user input. If it is associated with a secondary event, the UI element remains enabled for user interaction. Either way, though, if an action has been disabled, it is impossible for it to be triggered by a UI element.

As you will probably appreciate, this allows you to enable or disable all the UI elements on the screen using a single call to the `setEnabled()` method of the relevant action. Don't fall into the trap of thinking that, to disable a `Button` or a `LinkToAction` UI element, you have to access the UI element object directly within method `wdDoModifyView()` and then disable it explicitly. All UI elements using an action can be enabled or disabled automatically via their associated action object.



Important: If you disable an action associated with a secondary event, the UI element will remain enabled for user interaction. An example of this is the table UI element. If you have disabled the action associated with the secondary event `onSelect`, the table can still be scrolled, but now the action associated with the `onSelect` event will never be raised.

7.6 Layout managers

The purpose of a layout manager is to provide a structure within which UI elements can be presented. All Web Dynpro UI element containers must implement a layout manager of some sort.

Every Web Dynpro view is represented as a hierarchy of UI elements. This hierarchy is created automatically whenever a view controller is declared, and the view's root UI element has the following properties:

- ▶ It is always of type `TransparentContainer`.
- ▶ It is always called `RootUIElementContainer`.
- ▶ By default, the `RootUIElementContainer` always has the `FlowLayout` layout manager assigned to it.
- ▶ All UI elements subsequently added to the view become children of `RootUIElementContainer`.

When a layout manager is assigned to a UI element container, at design time a set of property values must be specified for each child UI element that is specific to the layout manager. It is within the layout data object that you specify how that child UI element should appear when rendered with the given layout manager.

7.6.1 Flow layout

The `FlowLayout` is the simplest of the layout managers in that it renders its child UI elements in a simple left-to-right horizontal sequence. If more UI elements have been defined than will fit horizontally across the screen, a new row is created.

As you resize the window within which the `FlowLayout` container lives, you will see the UI elements wrap automatically within the available screen space.

It is not possible to define any form of vertical alignment within a `FlowLayout` container.

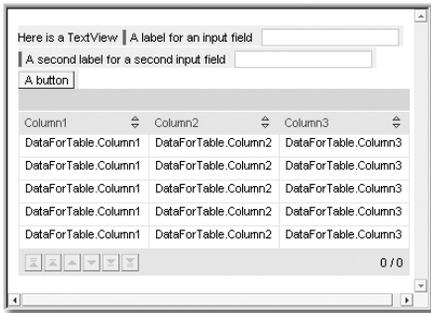


Figure 7.3 UI elements arranged in a container using a Flow layout manager; narrow screen

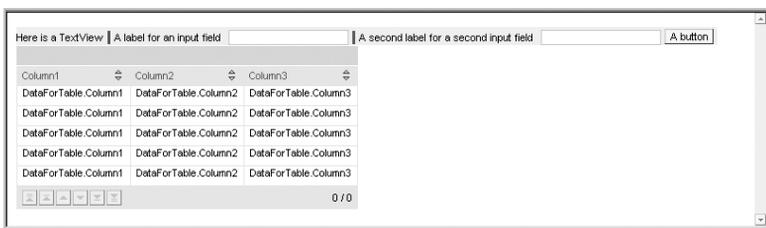


Figure 7.4 UI elements arranged in a container using a Flow layout manager; wide screen

7.6.2 Row layout

The `RowLayout` layout manager has been implemented primarily to overcome performance overhead incurred by browsers having to render multiple levels of nested HTML `<tables>`.

If you wish to subdivide some area of the view into horizontal rows, but you do not require any vertical alignment between the resulting columns, then you should use a `RowLayout` layout manager. This layout manager should be thought of as an enhanced form of `FlowLayout`.

Within a row of a `RowLayout` container, each child UI element will either contain a `RowHeadData` object or a `RowData` object. These objects are stored in the aggregation `layoutdata` and determine whether the UI element will start a new row or just be a row member. The default is that all child UI elements contain `RowData` objects.

Should you change a child element to contain a `RowHeadData` object, then you are telling the `RowLayout` layout manager that this particular element will forcibly start a new row. UI elements nominated to contain `RowHeadData` objects will always occupy the left-most position in a row.

A `RowHeadData` object has a set of general properties that apply to all UI elements in the row, that is, all UI elements up until the next `RowHeadData` object.

Once you have specified which UI elements will be that row's `RowHeadData` objects, the other UI elements in the row are free to rearrange themselves as if they lived in a `FlowLayout` container. Depending on the available screen width, you may very well see the contents of a `RowLayout` container wrapping around to form a new row. As with a `FlowLayout` container, the minimum width at which wrapping stops is imposed by the widest single UI element on the screen.

In Figure 7.5, the outlined UI elements are the ones with a layout data of `RowHeadData`. Notice that there is no vertical alignment of UI elements in corresponding columns.

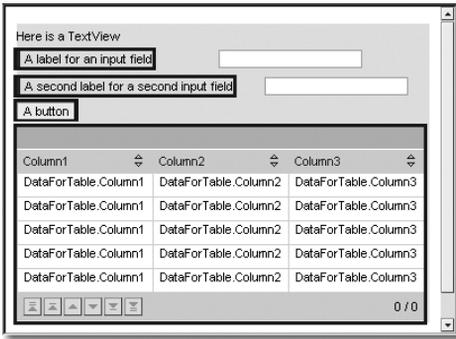


Figure 7.5 UI elements arranged in a container using a Row layout manager

7.6.3 Matrix layout

The `MatrixLayout` layout manager is a further enhancement of the capabilities of the `RowLayout` layout manager.¹⁰

A `RowLayout` layout manager allows you to specify when new rows should start, but provides no facility for the vertical alignment of elements within the row. This capability is provided by the `MatrixLayout` layout manager.

The `MatrixLayout` layout manager creates a tabular grid on the screen in which the cells are aligned both horizontally and vertically. As with the `RowLayout` layout manager, you still have to specify which child UI ele-

¹⁰ UI elements arranged in a `MatrixLayout` or `GridLayout` are implemented in a browser using an HTML `<table>`.

ments will be at the start of a new row, but now all the row elements will be vertically aligned into columns.

Using a `MatrixLayout` layout manager, you can produce a grid with a variable number of columns per row.

As with `RowLayout` managed UI containers, each child UI element assigned to a `MatrixLayout` container will contain either a `MatrixData` object or `MatrixHeadData` object. Again, these objects are stored in `layoutdata` aggregation. The default object type is `MatrixData`, but if you wish to start a new row, you must change this to `MatrixHeadData`.

In Figure 7.6, the outlined UI elements are the ones with a layout data of `MatrixHeadData`. Notice that there is now a tabular arrangement of the UI elements.

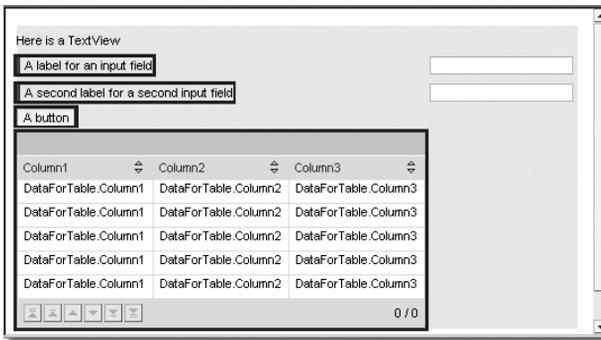


Figure 7.6 UI elements arranged in a container using a `Matrix` layout manager

7.6.4 Grid layout

The `GridLayout` layout manager divides the view area into a tabular grid with a fixed number of columns. As UI elements are added to a `GridLayout` container, they are positioned within the grid in a left-to-right, top-to-bottom manner.

The number of columns in the grid is determined by the value of the `colCount` property, and the number of rows is dependent upon the number of UI elements added to the container.

To achieve a uniform look and feel across all of your Web Dynpro applications, SAP recommends that the `MatrixLayout` be used in preference to the `GridLayout`.



Important: The time taken for browsers to render a screen can rise if the HTML contains multiple levels of nested `<table>` tags. Since the `GridLayout` and `MatrixLayout` layout managers are implemented in a browser using an HTML `<table>`, if possible you should try to avoid nesting these layout managers within each other.

A better approach when designing a screen layout is to divide the screen into horizontal areas as early as possible. The horizontal subdivisions can be implemented using a `RowLayout` layout manager, and each child added to the row could then be some sort of container such as a `Transparent-Container`. This will avoid the drop in browser rendering performance because you will not be using an HTML `<table>` to provide the major structural subdivisions of the screen.

The view shown in Figure 7.7 is the same layout as seen in the previous figures, but now that the view container is using a `Grid` layout manager with the `colCount` parameter set to 2, all the UI elements have assigned an arbitrary position in the table, on a left-to-right, top-to-bottom basis. The table UI element has had its `colSpan` parameter set to 2.

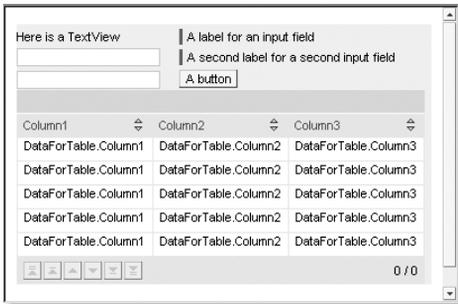


Figure 7.7 UI elements arranged in a container using a `Grid` layout manager; `colCount = 2`

This layout is obviously not satisfactory because we want some rows to have only one UI element in them. If you are using a `Grid` layout manager, then you will need to pad the empty grid cells with invisible UI elements. These can be seen in Figure 7.8.

If you require a tabular layout for your UI elements, then SAP recommends that the `Matrix` layout manager should be used in preference to the `Grid` layout manager.

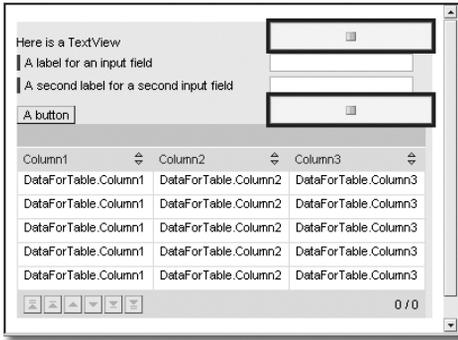


Figure 7.8 UI elements arranged in a container using a Grid layout manager; invisible elements used for padding

7.6.5 Layout Manager Properties

FlowLayout		
Layout Manager Properties	Data Type	Default Value
defaultPaddingBottom	String	" "
defaultPaddingLeft	String	" "
defaultPaddingRight	String	" "
defaultPaddingTop	String	" "
wrapping	Boolean	True
Layout Data Properties		
Layout Manager Properties	Data Type	Default Value
paddingBottom	String	" "
paddingLeft	String	" "
paddingRight	String	" "
paddingTop	String	" "
RowLayout		
Layout Head Data Properties	Data Type	Default Value
hAlign	WDCellHAlign	LEFT
rowBackgroundDesign	WDCellBackgroundDesign	TRANSPARENT
rowDesign	WDLayoutCellDesign	R_PAD
vGutter	WDLayoutCellSeparator	NONE

Table 7.2 Properties of Layout Manager Classes and Their Associated Data and HeadData Classes

GridLayout		
Layout Manager Properties	Data Type	Default Value
stretchedHorizontally	Boolean	True
stretchedVertically	Boolean	True
Layout Data Properties	Data Type	Default Value
cellBackgroundDesign	WDCellBackgroundDesign	TRANSPARENT
cellDesign	WDLayOutCellDesign	R_PAD
colSpan	Integer	1
hAlign	WDCellHAlign	LEFT
height	String	""
vAlign	WDCellVAlign	BASELINE
vGutter	WDLayOutCellSeparator	NONE
width	String	""
GridLayout		
Layout Manager Properties	Data Type	Default Value
cellPadding	Integer	0
cellSpacing	Integer	0
colCount	Integer	1
stretchedHorizontally	Boolean	True
stretchedVertically	Boolean	True
Layout Data Properties	Data Type	Default Value
colSpan	Integer	1
hAlign	WDCellHAlign	LEFT
height	String	""
paddingBottom	String	""
paddingLeft	String	""
paddingRight	String	""
paddingTop	String	""
vAlign	WDCellVAlign	BASELINE

Table 7.2 Properties of Layout Manager Classes and Their Associated Data and HeadData Classes (cont.)

7.7 Principles for the efficient use of layout managers

1. Wherever possible, try to avoid complex layouts involving multiple levels of nesting.

When you have the option of nesting UI containers within each other (each with its own layout manager), always opt for the design that results in the fewest levels of nesting. From a performance point of view, it is better to place multiple UI elements directly into one large UI container using a grid or matrix layout (with columns and rows that span where necessary) than to nest transparent containers within the individual cells of the parent container.

2. Only use a transparent container when it is genuinely required. Containers such as the Group control are composite UI elements based on a transparent container. Therefore, it makes no sense to embed a transparent container as the top level child into a Group container, because it already implements one!
3. If vertical alignment is not required, the row layout should be chosen in preference to the grid or matrix layout.
4. If vertical alignment is required, the matrix layout should be chosen in preference to the grid layout. This is not a performance consideration (both layout managers are ultimately implemented using an HTML `<table>`), but it is an easier layout manager to work with. You don't have to specify a column count and you can put as many controls into one row as you like.
5. The matrix layout allows some predefined values for cell padding. The property `cellDesign` can have the following predefined values. The `Standard` option is also referred to as `rPad`.

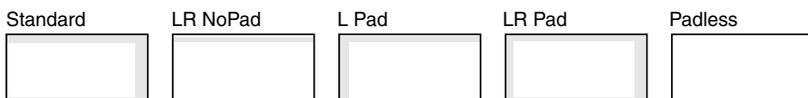


Figure 7.9 The Different Adding Options for the `cellDesign` Property

7.8 Locale-specific text

7.8.1 Introduction to multilingual support in software products

One of the age old problems with the distribution of software products within geographical regions such as Europe has been that of language support. The French don't want to speak German, the Italians don't want to speak Spanish, and the English (and Americans) can *only* speak English!

This problem has produced a variety of solutions; some vendors distribute entirely new versions of their products with the language-specific text embedded within the executable code, and other vendors have opted to separate the language-specific content from the executable code.

The requirement for multilingual operation has, from SAP's earliest days, been a fundamental design criterion of all its software products. In the R/3 system (and all of its derivative systems), the data stored in its relational database tables is organized in third normal form. This has the direct consequence that business data and the text that describes the business data are always stored in separate tables related by a foreign key.

7.8.2 Internationalization

The word "internationalization" is used to describe either the design process required to make a software product functional in all required languages, or the modification process by which existing software is adapted from single language operation to multilingual operation. The result is a software product in which all language-specific text is external to the executable code that uses it.

Because the word "internationalization" is so long, it is abbreviated to *i18n*—that is, the first letter "i", the last letter "n", and don't bother writing the other 18 letters in the middle!

From now on, we will talk about the "i18n process" or "Web Dynpro i18n" rather than using the full word.

7.8.3 Externalization

Externalization is the process by which hard-coded text strings are removed (i.e., externalized) from a source code file and placed into a `.properties` file. The original source code is then modified to access a generated resource bundle accessor class. Within the NWDS, an Exter-

nalize Strings wizard automates the extraction and code modification process.

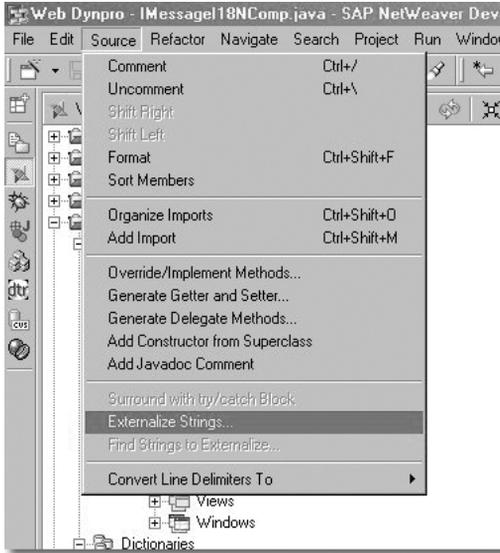


Figure 7.10 Externalize Strings Wizard for Removing Text Strings from Existing Code

For more information on this process, see the standard SAP documentation "Internationalization in the SAP NetWeaver Developer Studio."

7.8.4 Web Dynpro i18n concept

In keeping with the R/3 tradition of separating data from the text that describes the data, the Web Dynpro i18n concept separates text strings from the programs that manipulate those strings. Therefore, Web Dynpro Java class files, metadata files, and dictionary simple types do not contain any language-specific text. The standard Java class `java.util.ResourceBundle` is used for managing language-specific text at runtime. See the javadoc for more information on the exact details of the operation of this class.



Important: When a Web Dynpro project or DC is created, a language is required such as `British_English` or `Spanish` or `Hebrew`. This language setting serves several purposes:

- ▶ To inform a translator of the language in which the developer originally wrote the Web Dynpro project or DC.

- ▶ To determine the original language of all current and future metadata files in this project.

This value is *not* read at runtime by the WDF when determining the session locale of an application.

Caveat confector:

The project language cannot be changed after a project has been created, and you cannot copy metadata between projects of different languages!

Not all locales recognised by Java are permissible within a Web Dynpro project or DC. The permissible languages are *only* those found in the drop down list seen when a project or DC is created. The reason for this is that internally, SAP uses an R/3 system to serve as a translation engine. This immediately reduces the set of permissible Web Dynpro languages to the subset of languages (or dialects) within which R/3 operates.

Languages not known to R/3 are not permissible in Web Dynpro.



7.8.5 S2X—SAP's use of the XLIFF standard

SAP has taken the XML Language Interchange File Format (XLIFF) and produced a reduced and somewhat modified variant known as "SAP Supported XLIFF" (S2X). SAP's S2X compliant files all use the `.XLF` file name suffix and differ from standard XLIFF in the following ways:

- ▶ S2X imposes the following restrictions upon standard XLIFF:
 - ▶ XLIFF's mechanism for handling alternate translations from different sources, such as a Translation Memory System or a Machine Translation System, has not been implemented.
 - ▶ Certain textual content will be encoded using only the lower half of the ASCII character set, i.e., 7-bit ASCII.
- ▶ S2X extends standard XLIFF in the following areas:
 - ▶ Certain XLIFF extensions have been implemented that can accommodate the classification of SAP's software according to software component, development component, and release.
 - ▶ Certain XLIFF constraints have been made optional.

An S2X file contains two different types of data: header data and content data.

- ▶ Header data describes the properties of all the contents stored in the file.
- ▶ Content data are the text items accompanied by supplementary information, such as unique identifiers, that may be used in reuse or update strategies.

This can be seen in the S2X editor as the tabs **Header** and **Resource Text**. (See Figure 7.13 and Figure 7.14 below.)

7.8.6 Storing language specific text in .XLF files

For each type of entity that can hold language specific content, there will be a corresponding XLF file created. These XLF files will only be created if the developer adds some text, e. g. hard coding a value for the `text` property of a Label UI element.

In general, the following XLF files will be created when language specific content is added:

- ▶ **{nv}.wdcontroller.xlf**
Action texts in a view controller
- ▶ **{nv}.wdview.xlf**
UI element text, tooltip and imageAlt values in a view layout
- ▶ **{w}.wdwindow.xlf**
The value of the window's title property
- ▶ **{nc}MessagePool.wdmessagepool.xlf**
Component message pool content
- ▶ **{st}.dtsimpletype.xlf**
Enumeration display texts, field labels, column headers and tooltips for dictionary simple types

Notice that none of the above filenames contains a locale value. During the development of a Web Dynpro application, the developer will only be working in a *single* language—the one specified when the project was created. Consequently, all XLF files generated during the development of the application are assumed to belong to this locale.

7.8.7 Translating XLF files

In the SAP NetWeaver '04 version of the NWDS, there is not yet an IDE based tool for translating XLF files. The creation of an XLF file for any language other than the project default needs to be performed manually. However, this amounts to nothing more than locating the original XLF file in your NWDS workspace directory, and then duplicating it.¹¹ The important thing to remember is to include the new locale value at the correct position in the file name.

For any new locale {1}, the original file should be copied and renamed thus:

- ▶ {nv}.wdcontroller.xlf becomes {nv}.wdcontroller_{1}.xlf
- ▶ {nv}.wdview.xlf becomes {nv}.wdview_{1}.xlf
- ▶ {w}.wdwindow.xlf becomes {w}.wdwindow_{1}.xlf
- ▶ {nc}MessagePool.wdmessagepool.xlf becomes {nc}MessagePool.wdmessagepool_{1}.xlf
- ▶ {st}.dtsimpletype.xlf becomes {st}.dtsimpletype_{1}.xlf

Once new locale specific XLF files have been created, the project view in the Package Explorer should be refreshed, and then the S2X editor can be used to edit the contents.

7.8.8 Use of the S2X editor within NWDS

SAP has created an editing tool within the NWDS that allows you to edit .XLF files in SAP's specific S2X format.

Caveat confector:

The S2X editor is *not* a Web Dynpro specific tool. It has been provided only to fill a functional gap in Eclipse.

The S2X editor should *never* be used to edit language specific content in the project's default language. If you have created your project in German, then all German text belonging to UI elements, dictionary simple types and message pools, should be edited using the standard Web Dynpro tools.



¹¹ In the same directory!

Using the S2X editor, it is possible to change the source language of an XLF file, but this change will not cause the filename to be updated (remember, all text belonging to languages other than the project language must have the locale `{1}` embedded in the file name); therefore, such changes will create inconsistencies within your Web Dynpro project.

If you use the S2X editor to change the XLF file associated with a Web Dynpro view (for instance) *and* you have already opened that view through the normal Web Dynpro editor, then you will not see your text changes in the view layout until you reload the project.

7.8.9 Editing MessagePool XLF files

To edit an XLF file, the SAP NetWeaver '04 version of the NWDS provides an S2X editor. If you double-click on the Message Pool belonging to a Web Dynpro project from the Web Dynpro Explorer menu, you will see a version of the S2X editor applicable for `MessagePool.s`.



Figure 7.11 S2X Editor for a Component MessagePool

To edit the XLF files associated with view controllers, windows, and dictionary simple types, you should change from the Web Dynpro Explorer view of the project to either the Package Explorer or Navigator views. Here you can expand the `src` directory and locate the XLF files. These files are marked with an  icon.

The Package Explorer provides the most direct route to the XLF files, and is shown in Figure 7.12.

By double-clicking on the view controller's XLF file (`I18NView.wdview.xlf`) shown above, you will see the editor screens in Figure 7.13 and Figure 7.14.



Figure 7.12 Navigator View of the XLF Files in a Web Dynpro Project

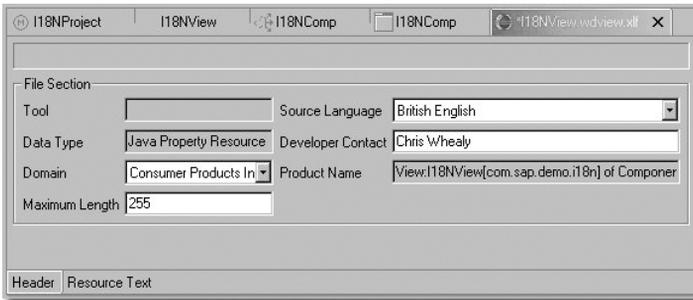


Figure 7.13 The Header Screen in the S2X Editor for a View Controller's XLF File

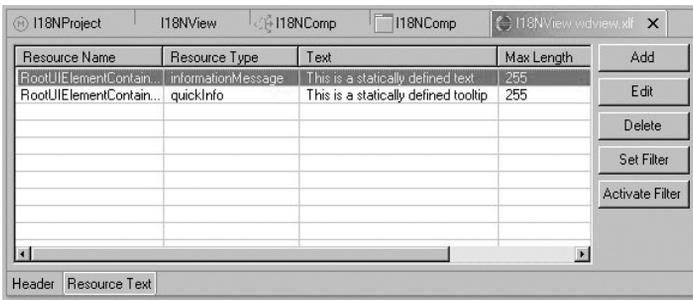


Figure 7.14 The Resource Text Screen in the S2X Editor for a View Controller's XLF File

The application developer can edit all the text resources for his or her project using the S2X editor. As was stated earlier, these XLF files are assumed to contain text that belongs to the language specified when the project was created.

Notice that the view of the S2X editor shown in Figure 7.11 is slightly different from the view seen in Figure 7.14. There is a good reason for this! The view of the S2X editor seen when editing a Message Pool from the Web Dynpro imposes two restrictions:

- ▶ The available message types are limited to Standard, Warning, Error, and Text because these are the only message types applicable for a Message Pool.
- ▶ The S2X header information has been suppressed.

However, when the S2X editor is started from either Navigator or Package Explorer views, you will see the full S2X editor, in which both the header information and the full range of message types are accessible.



Caveat confector: Do not use the S2X editor to change any text belonging to your project's default language! These changes should be made through the Web Dynpro tools in order to ensure the consistency of the underlying XLF files.

7.8.10 Runtime Locale Identification

Within the scope of a project, the locale of a Web Dynpro application can be defined using the application property `DefaultLocale`. This hard coded value will act as the application's default locale unless it is explicitly overridden.

Web Dynpro makes use of the standard fallback process within `java.util.ResourceBundle` to determine which locale value should be used for a particular application.

The following table shows how the fall back process works. The first column indicates the type of user accessing the system. The "Developer" user is the only user for which the `sap.locale` URL parameter is considered legitimate. Once a Web Dynpro application has been developed, the URL parameter `sap.locale` should not normally be used.

If a Web Dynpro application's authentication flag is set to true, then valid user credentials must first be supplied to the WDF before the application

can be run. If the user credentials are obtained from the User Management Engine (UME), then the required locale value will be supplied.

User	Locale specified in						
	URL	User Id	Browser	DefaultLocale property	WD system	VM default	Final locale
Developer	pr	de	en	fr	it	ru	pr
Authenticated		de	en	fr	it	ru	de
Anonymous			en	fr	it	ru	en
Anonymous				fr	it	ru	fr
Anonymous					it	ru	it
Anonymous						ru	ru

Table 7.3 Fall back process for locale determination

7.8.11 Locale-dependent text at runtime

There are five main categories of language-dependent text that you could need access to at runtime. The first four categories are the message types that exist within a Message Pool:

- ▶ Error
- ▶ Warning
- ▶ Standard
- ▶ Text

The first three types are the ones used by the `IWDMessagesManager` class and become runtime constants within a generated class `IMessage{nc}.java`, each message being of type `IWDMessages`.

Important: Only Message Pool messages of type `Standard`, `Warning`, and `Error` are addressable as constants in the generated class `IMessage{nc}`. Messages of type `text` *do not* appear in this generated class.



Messages of type `Text`, however, are not accessible to the `IWDMessagesManager` class; instead, you should use class `IWDTextAccessor`. Messages of type `text` are text strings that either have been created as language-specific texts or have been extracted from existing code using the NWDS Externalize Strings wizard.

The fifth category of locale-dependent texts is those that belong to dictionary simple types. These texts can be accessed through the `ISimpleType` interface.

Once the application is deployed, the locale-dependent texts are stored in standard resource bundle files.

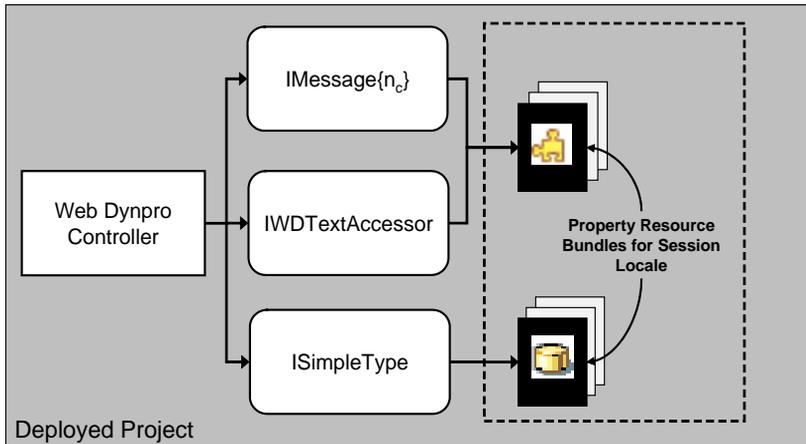


Figure 7.15 The Interfaces needed to Access Locale-Specific Text

The following example assumes that a message called `Message1` of type `text` has been defined in a component's message pool. To access this specific message, you need to use the following code.

```
// Get the text accessor from the current component
IWDTTextAccessor textAccessor =
    wdComponentAPI().getTextAccessor();

// Get the message by name from the message pool
String msgFromPool = textAccessor.getText("Message1");
```

Listing 7.5 Code Fragment for Obtaining a Message of Type `text` from the Message Pool

Remember that the component's message manager has no access to messages of type `text`, so these types of messages should be used only to supply UI elements. The purpose of the `IWDMessagesManager` interface is to supply the user with informative messages about the success or failure of the application's functionality.

The code fragment below assumes that there is a message called `Message2` of type `error` in the message pool. This message will be reported to the

user with no parameters (`null`), and navigation will not be canceled as a result of the error (`false`).

```
// Get the message manager from the current component
IWDMessagesManager msgMgr = wdComponentAPI().getMessagesManager();

// Issue a warning message using a text constant from the
// generated IMessage(nc) class
msgMgr.reportMessage(IMessage(nc).MESSAGE2, null, false);
```

Listing 7.6 Code Fragment for Issuing a Message of Type error from the Message Pool

Within any component `{c}`, any statically defined text, such as message pool texts or hard-coded text values for UI elements, will be placed into a generated resource bundle file called `Resource{c}.properties`.

This file can be viewed (but must not be edited!) from the Package Explorer view in the NWDS. The general path name is:

```
{pr} * gen_wdp/packages * {pkg1} ... {pkg_n}. {nc1}.wdp *
Resource{c}.properties
```

7.8.12 Defining placeholders within a message text

There is often the need to be able to substitute a variable value into a static message string. This can be achieved with numbered placeholders within the text string. The messages are now known as message text patterns.

For instance, if you are writing an application that creates business documents, you will probably want to inform the user what number the newly created document has. Therefore, you would enter the message text into the Message Pool as shown in Figure 7.16.

Message Key	Message Type	Message Text
DocumentCreated	standard	Document {0} has been successfully created

Figure 7.16 A Placeholder in a Text Message

If you wish to have more than one placeholder within the text message, simply increment the placeholder number as shown in Figure 7.17.

Message Key	Message Type	Message Text
DocumentCreated	standard	Document {0} successfully created at {1} on {2}

Figure 7.17 Multiple Placeholders in a Text Message

Where {0} is the document number, {1} is the time, and {2} is the date.



Caveat confector:

All message placeholders must be sequentially numbered integers.

All placeholder values must be supplied as Java strings.

Message text patterns use `java.text.MessageFormat` without using element formats.

The code to issue the above `documentCreated` message must now supply a parameter value:

```
// Get the message manager from the current component
IWDMessageManager msgMgr = wdComponentAPI().getMessageManager();

// Get the document number from somewhere...
String docNo = getDocumentNo();

// Issue a warning message using a text constant from the
// generated IMessage{nc} class
msgMgr.reportMessage(IMessage{nc}.DOCUMENT_CREATED,
    new Object[] {docNo}, false);
```

Listing 7.7 Code Fragment for Issuing a Standard Message with Placeholder Parameters

Notice in Listings 7.6 and 7.7 that the class name for the `IMessage{nc}` class has been generalized.

7.9 Accessing any parameter in the query string

Normally, any query string parameters will automatically be mapped to parameters of the same name in the startup plug of the component controller interface view.¹² (Earlier versions of Web Dynpro required that URL parameters be prefixed with `app`. This is no longer a requirement, but the syntax is still supported.)

However, you may find yourself in a situation in which the calling application passes a variable set of parameters in the query string. Under these circumstances, it will probably be easiest to parse the entire query string, rather than attempt to declare all possible query string names as parameters to the startup plug's event handler.

¹² The startup plug event handler will typically be called `onPlugDefault()`.

You can obtain the entire query string as a `java.util.Map` object using the following code:

```
public void onPlugDefault(IWDCustomEvent wdEvent) {
    //@@begin onPlugDefault(ServerEvent)
    // Get the entire query string as a Map object
    Map qsMap = WDWebContextAdapter.
        getWebContextAdapter().getRequestParameterMap();
    //@@end
}
```

Listing 7.8 Code Fragment for Obtaining a Variable Value from the Query String

This technique is particularly useful when Web Dynpro applications are called from SAP's Enterprise Portal.

Index

{n_c} 266
{n_c}Interface 267
{n_v} 268

A

ABAP database updates 250
ABAP Dictionary 200, 203
ABAP function module 194
ABAP sessions 247
Action handler generalization 133
Adaptive RFC (aRFC) 189, 201
Adaptive RFC layer 198
AdaptiveRFCComp 218
Analysis phase 27
application 33, 279
 parameters 34
 properties 34
 shutdown 36
Application event handler 260
Application interfaces 279
Architecture design 29

B

BAPI 21, 193
Binding 79
binding UI elements 125

C

Calculated attributes 63
cardinality 66
CHANGING parameters 195
check box 310
Coding principles 125
Common Model Interface (CMI) 189
Component 36
 interaction 37
 Java Classes 41
 lifespan 37
 reuse 38
Component controller 23, 39, 213, 266, 273
Component Interface controller 267
Component Interface
 Java Classes 43
Component usage 274

Connection management 238
container UI element 340
Context 61, 208, 258
 Attributes 62
 Nodes 61
Context Attribute Name 136
Context interfaces 279
Context manipulation 103
Context mapping 40, 84, 89
Context model nodes 79, 209, 211
Context node hierarchy 127
Context nodes 125
Context structure
 at design time 61
 at runtime 66
Controller 22, 38, 277
 Interaction 40
 structure 41
 types 38
Controller classes 266
Controller constructors 263
Controller interfaces 271
CPI-C 198
createNewAddress() 220
creating a technical system 243
Custom controller 24, 39, 43
Custom Controller
 Java Classes 42

D

Data Modeller 208
DataContainer 258
decoupling of the UI 135
deleteAddress() 221
Design phase 27
Detail design 30
development cycle 27
Dictionary Data Types 70
doBeforeNavigation 260
doErrorMsg() 221
doPostProcessing 261
drop-down list 303
 index-based 304, 312
 key-based 305, 312
Dynamic context manipulation 103

- Dynamic UI generation 161
- Dynamic UI manipulation 261
- dynamic view construction 173
- dynamic view layout 173

E

- element 64
 - selection status 65
- element collection 64
- Enterprise Connector 199
- entity names 53
- Event handling 46
 - actions 47
 - events 48
- EXCEPTIONS 195
- EXPORTING parameters 195
- External mapping 86, 89
- External public interface controller 271
- Externalization 147

F

- Flow layout 139
- flow layout data 313
- flow layout manager 314

G

- generated code 266
- Generated controllers 269
- Generated Model Classes 204
- generic API 103
- generic context API 105
- generic hypertext link 320
- getElement(String) 164
- Grid layout 142
- Grid layout data 314
- Grid layout manager 316

H

- hook methods 263
- HTTP layer 47
- HTTP servlet request 33
- hypertext link 320, 321

I

- I{cn}Element 279
- I{cn}Node 280
- IContextElement 281
- IContextNode 282
- IExternal{nctl}Interface 271
- IFrame 316
- image 317
- Implementation phase 31
- IMPORTING parameters 195
- individual tab page 326
- initAddressData() 228
- input field 306, 318
- Interaction 37
- interface controller 23
- Internal private interface controller 272
- Internal public interface controller 272
- Internal{nc}Interface 269
- Internal{nctl} 269
- Internationalization 147
- IPrivate{nctl} 272
- IPublic{nctl} 272
- items 307
- IWDAbstractButton 302
- IWDAbstractCaption 303
- IWDAbstractDropDown 303
- IWDAbstractDropDownByIndex 304
- IWDAbstractDropDownByKey 305
- IWDAbstractInputField 306
- IWDAbstractTreeNodeType 307
- IWDAction 138
- IWDApplication 279
- IWDAttributeInfo 283
- IWDButton 309
- IWDCaption 309
- IWDCheckBox 310
- IWDCheckBoxGroup 311
- IWDCComponent 273
- IWDCComponentUsage 274
- IWDContext 284
- IWDController 277
- IWDDropDownByIndex 312
- IWDDropDownByKey 312
- IWDFlowData 313
- IWDFlowLayout 314
- IWDGridData 314
- IWDGridLayout 316
- IWDIFrame 316
- IWDImage 317
- IWDInputField 318
- IWDLabel 318
- IWDLLayout 319

- IWDLayOutData 319
- IWDLink 320
- IWDLinkToAction 321
- IWDLinkToURL 321
- IWDMatrixData 322
- IWDMatrixHeadData 323
- IWDMatrixLayout 324
- IWDNode 280, 282, 286
- IWDNodeElement 279, 281, 291
- IWDNodeInfo 293
- IWDRadioButton 324
- IWDTab 326
- IWDTable 328
- IWDTableColumn 332
- IWDTabStrip 327
- IWDTextEdit 334
- IWDTextView 335
- IWDTree 336
- IWDTreeItem 338
- IWDTreeNodeType 338
- IWDUIElement 339
- IWDUIElementContainer 340
- IWDView 277
- IWDViewController 278
- IWDViewElement 342

J

- Java Connectivity Builder 199
- Java Connector (JCo) 198
- Java Development Infrastructure (JDI)
 - 29
- JCo destination 244
 - and ABAP sessions 247

L

- layout data 319
- Layout managers 139, 319
 - principles for efficient use 146
- Layout Managers
 - properties 144
- Lifespan 37
- LinItems 96, 99
- Locale-dependent text 155
- Locale-specific text 147

M

- mapped context node 113
- mapping 84

- Matrix layout 141
- matrix layout data 322
- matrix layout head data 323
- matrix layout manager 324
- MessagePool XLF files 152
- Model 21, 46, 189
- Model attributes 74
- Model class 190
- Model nodes 74, 78
- Model object hierarchies 206
- Model objects 79, 207
 - at runtime 217
- Model View Controller (MVC) 19, 21
- multi-line text editor 334
- multilingual support 147
- multiple checkbox selection group 311

N

- Naming conventions 53
- naming placeholders 49
- naming standards 73
- Navigation 260
- node names 72
- Nodes 61
- Normal mapping 85

P

- Parameter mapping 131, 136
- path length problem 54
- principles of the context 82
- push button 302, 309

Q

- query string 158

R

- radio button 324
- readAddressData() 219
- read-only text display 335
- readSingleAddress() 220
- Read-Write-Read problem 252
- Recursive Nodes 75, 80
- Remote Function Call 193
- replicateNodeInfo() 121
- Response rendering 262
- response rendering stage 130
- Reuse 38
- RFC Architecture 199

- RFC module 194
 - Custom written 196
- RFC module interface 200
- root node 336
- RootUIElementContainer 164
- Row layout 140
- Runtime Locale Identification 154

S

- S2X 149
- S2X editor 151
- SAP function modules 193
- SAP J2EE Engine 33
- SAP Supported XLIFF 149
- Selection cardinality 69
- Selection mapping 85
- Self reference 265
- service event handler 259
- shortcuts 265
- showNode() 178, 183
- showNodeAsColumns() 185
- showNodeAsTable() 187
- Simple Value 238
- SingleRecordView 225, 233
- singleton 67
- Supply functions 75
- System event handler 259

T

- tab page 326
 - aggregation 327
- table 328
- table column 332
- TableDisplayView 222
- TABLES parameters 195
- technical system 243
- Technical System Browser 245
- text caption 303, 309
- text display 335
- text editor 334
- text label 318
- this 265
- tree 336
- tree item 338
- tree nodes 307, 338
- tree UI element 127
- typed API 91

U

- UI element actions 300
- UI element hierarchy 176, 236
- UI element properties 299
 - naming 302
- UI elements 125, 138, 162, 339, 342
 - accessing 163
 - Appearance 301
 - controlling from the context 175
 - source property 302
- UI manipulation 261
- unmapped context node 108
- URL 321
- User modifications 266
- User-defined code 125
- User-defined methods 264

V

- Validate modified data 258
- Value attributes 74
- Value nodes 73, 77
- View 22, 43
 - Java Classes 42
- View controller 24, 39, 43, 222, 268, 278
 - interaction 45
 - layout 45
 - lifespan 44
 - navigation 46
- View initialization 260
- View layout 277
- View layout interfaces 299
- visitElement() 172

W

- wdDoBeforeNavigation() 260
- wdDoExit() 264
- wdDoInit() 264
- wdDoModifyView() 164, 173, 232, 261, 262
- wdDoPostProcessing() 262
- wdThis 265
- Web Dynpro
 - definition 19
 - design concept 21
 - philosophy 19
- Web Dynpro action 321
- Web Dynpro application 33

Web Dynpro component 23, 30, 36
Web Dynpro controller 24, 38, 207
Web Dynpro development
 components 29, 38
Web Dynpro Dictionary 70
Web Dynpro Framework (WDF) 33
Web Dynpro i18n 147
Web Dynpro i18n concept 148
Web Dynpro phase model 75, 257

Web Dynpro project 29, 30
Web Dynpro view 31
Web Dynpro window 31
Web Services 21, 189

X

XLIF 150
XML Language Interchange File
 Format (XLIFF) 149