Chris Whealy

Inside Web Dynpro for Java
Contents at a Glance

PART I Background
1 Introduction ................................................................................. 29

PART II The Fundamental Concepts
2 The Web Dynpro Component Concept ........................................ 53
3 Designing a Web Dynpro Application ........................................... 89
4 Web Dynpro Phase Model ............................................................ 141

PART III Basic Development
5 The Context at Design Time ....................................................... 163
6 Context Structure at Runtime ..................................................... 207
7 UI Elements ............................................................................... 237
8 Writing Multilanguage Applications .......................................... 287

PART IV Advanced Development
9 Dynamic Context Manipulation .................................................. 297
10 Advanced Use of the UI ............................................................. 311
11 Navigation and Error Handling ................................................. 359
12 Components at Runtime ........................................................... 387
13 The Adaptive RFC Layer ........................................................ 407
14 JCo Connection Management ................................................... 467
Contents

Author’s Apology — Preface to the Second Edition ........................................ 21

Part I: Background

1 Introduction ................................................................................................. 29

1.1 Why Did SAP Decide to Build Their Own Web Development Framework? ........................................................................................................ 29

1.2 What Is Web Dynpro? ........................................................................... 30

1.3 Causes of Poor Web Dynpro Implementations .................................... 32

1.4 How Web Dynpro Differs from Other Web Development Tools .......... 32

1.5 The Fundamental Difference Between Web Dynpro and Other Web Development Tools .......................................................... 34

1.6 A Quick History Lesson .................................................................... 35

1.6.1 The Original Design Concept .................................................... 35

1.6.2 The Web Dynpro Implementation ........................................ 36

1.7 The Design Philosophy Behind Web Dynpro .................................... 38

1.7.1 Design Influences ...................................................................... 38

1.7.2 Why Do I Need an SAP-Supplied Server To Run a Web Dynpro Application? ........................................................................ 39

1.8 The Influence of MVC on Web Dynpro ............................................ 40

1.8.1 The User’s Mental Model .............................................................. 40

1.8.2 A Top-Down Approach ................................................................ 41

1.8.3 Interacting with Business Objects ............................................. 41

1.8.4 Simplifying the Editor .................................................................. 42

1.8.5 The Essence of the MVC Design Concept .................................. 43

1.8.6 The Data Consumer-Generator Concept in Web Dynpro .......... 45

1.9 The Influence of SAP’s Requirements on Web Dynpro .................... 45

1.9.1 The Consequences of a Declarative Approach to UI Design .......... 45

1.9.2 Decoupling Data Presentation from Data Processing ................. 46

1.10 The Influence of Focus Groups on Web Dynpro .............................. 47

1.11 What Do I Need To Understand To Be Successful with Web Dynpro? ......................................................................................... 47

1.11.1 Using Web Dynpro to Reproduce the User’s Mental Model of the Data ..................................................................................... 48
Part II: The Fundamental Concepts

2 The Web Dynpro Component Concept ................................ 53

2.1 A Typical MVC Architecture ............................................. 53
2.2 The Web Dynpro Component Concept ................................ 55
  2.2.1 How Does a Web Dynpro Component Alter
  Reenskaug’s Original Design? ............................................. 56
  2.2.2 Why Bother with Components? .................................... 57
2.3 Inside the Web Dynpro Component ..................................... 58
2.4 What Do the Different Parts of the Component Do? ............. 60
  2.4.1 The Programmatic Parts of a Component ...................... 60
  2.4.2 The Visual Parts of a Component ................................. 62
2.5 Inside a Web Dynpro Controller ......................................... 65
  2.5.1 Architecture of Non-Visual Controllers ....................... 65
  2.5.2 Architecture of Visual Controllers .............................. 72
  2.5.3 Controller Interaction .............................................. 78
2.6 Communication Between Controllers .................................. 79
  2.6.1 Usage Declarations .................................................. 79
  2.6.2 Context Mapping ................................................... 80
  2.6.3 Data Binding ........................................................ 81
  2.6.4 Summary of the Roles Played by the Different
  Controllers ................................................................. 82
2.7 Web Dynpro Applications or “Where Does My Component
  Sit in the Big Scheme of Things?” ...................................... 83
  2.7.1 Accessing Web Dynpro Functionality from
  a Browser ................................................................. 83
  2.7.2 Simple Application Structure .................................... 84
  2.7.3 More Realistic Application Structure ......................... 84
  2.7.4 Application Properties ............................................ 85
  2.7.5 Application Parameters .......................................... 86
  2.7.6 Start Up and Shut Down ......................................... 86
2.8 Managing Web Dynpro Components .................................... 87
Contents

3.5.9 Location of Business Processing ........................................ 127
3.6 Implementation ....................................................................... 129
3.6.1 Management of Child Component Instances ............... 129
3.6.2 Intercomponent Communication ................................. 129
3.6.3 Internal Data Storage Within Web Dynpro Components ......................................................... 135
3.6.4 Centralized Functionality ............................................. 137
3.6.5 Correct Use of Standard Hook Methods ...................... 138

4 Web Dynpro Phase Model ................................................... 141
4.1 Phase Model Processing and Component Instantiation ........ 143
4.2 Web Dynpro Hook Methods .................................................... 143
4.3 The Relationship Between Phase Model Processing and Web Dynpro Windows ......................................................... 144
4.4 Phase Model Processing .......................................................... 145
4.4.1 Transport Data to Data Container ............................... 145
4.4.2 doApplicationStateChange: The Floating Step — Part 1 .......................................................................... 145
4.4.3 Validate and Transport Data ........................................ 147
4.4.4 Call System Event Handlers ......................................... 149
4.4.5 Call Service Event Handlers ......................................... 149
4.4.6 Application Event Handlers ......................................... 151
4.4.7 doBeforeNavigation .................................................... 152
4.4.8 Navigation and View Initialization ............................... 153
4.4.9 Dynamic UI Manipulation ........................................... 155
4.4.10 Client Specific UI Validation ........................................ 156
4.4.11 doPostProcessing ........................................................ 157
4.4.12 doApplicationStateChange: The Floating Step — Part 2 .......................................................................... 158
4.4.13 Response Rendering .................................................... 159
4.5 Summary ................................................................................. 159

Part III: Basic Development

5 The Context at Design Time ................................................... 163
5.1 Nodes ...................................................................................... 163
5.2 Attributes ................................................................................ 164
5.3 Node and Attribute Data Types ............................................... 164
5.4 Context Attribute Data Types that Can Supply Data to UI Elements ......................................................... 167
### 5.5 Terminology Concerning Nodes and Attributes ........................ 168
### 5.6 What's the Difference Between Value Nodes and Model Nodes? ......................................................................... 169
  5.6.1 Value Nodes ............................................................... 169
  5.6.2 Model Nodes .............................................................. 170
### 5.7 The Most Important Node Properties to Understand ............... 171
  5.7.1 Node Cardinality ......................................................... 172
  5.7.2 Node Selection Cardinality .......................................... 174
  5.7.3 The Singleton Property ................................................ 175
### 5.8 Should Node Names Be Singular or Plural? .............................. 177
### 5.9 General Naming Conventions .................................................. 178
### 5.10 Identifying a Specific Node Attribute ....................................... 179
### 5.11 Calculated Attributes ............................................................... 180
  5.11.1 Implementing a Getter Method .................................. 182
  5.11.2 Implementing a Setter Method ................................... 183
  5.11.3 Coding Principles for Getter and Setter Methods ...... 185
### 5.12 Internal Context Mapping ........................................................ 186
### 5.13 External Context Mapping ....................................................... 187
### 5.14 Supply Functions ..................................................................... 189
### 5.15 Dispose Functions ................................................................... 194
### 5.16 Coding Principles for Supply and Dispose Functions ............. 200
### 5.17 Recursive Context Nodes ......................................................... 200
  5.17.1 Creating Recursive Context Nodes .............................. 201
  5.17.2 Mapping Recursive Context Nodes ............................. 203
  5.17.3 Restrictions on the Use of Recursive Nodes ................. 204
### 5.18 Fundamental Principles of the Context .................................... 205

### 6 Context Structure at Runtime .............................................. 207
### 6.1 Generated Classes .................................................................... 207
  6.1.1 The Typed Context API ............................................... 207
  6.1.2 Background to the typedAccessRequired Flag ............. 209
  6.1.3 The Byte Code Minimizer ............................................ 210
  6.1.4 Generated Classes and Context Mapping .................... 211
### 6.2 Working with Value Nodes ...................................................... 213
  6.2.1 Accessing the Context Through the Typed API ............ 213
  6.2.2 Accessing a Node Element .......................................... 214
  6.2.3 Accessing all Elements in a Node ................................. 215
  6.2.4 Creating a New Node Element ..................................... 216
  6.2.5 Treating the Node LineItems as a Singleton Node ...... 218
  6.2.6 Treating the Node LineItems as a Non-Singleton Node .......... 221
6.2.7 Accessing the Context Through the Generic API .......... 224
6.3 Working with Model Nodes .................................................... 230
  6.3.1 Adding Elements to a Model Node Collection ............. 230
  6.3.2 Binding Elements to a Model Node Collection .......... 232
6.4 Working with Recursive Nodes ................................................. 233
6.5 Using the WDCopyService ....................................................... 233

7 UI Elements .......................................................................... 237
  7.1 How are UI Elements Stored Within a View Controller? .......... 237
  7.2 Editing a View Layout .............................................................. 238
    7.2.1 Adding UI Elements to the View Layout ...................... 238
    7.2.2 Editing Existing UI Elements on the View Layout .......... 239
  7.3 Putting Data on the Screen ...................................................... 239
    7.3.1 Data Binding ............................................................... 239
    7.3.2 Binding UI Properties to the Context ......................... 240
    7.3.3 Controlling UI Element Behavior from the Context .......... 242
    7.3.4 Impact of Data Binding on the Coding ......................... 243
  7.4 Avoiding a Possible Source of Confusion ................................. 244
  7.5 Layout Managers .................................................................. 246
    7.5.1 Flow Layout ................................................................. 247
    7.5.2 Row Layout ................................................................. 247
    7.5.3 Matrix Layout ............................................................. 248
    7.5.4 Grid Layout ................................................................. 249
    7.5.5 Layout Manager Properties ......................................... 251
    7.5.6 Principles for the Efficient Use of Layout Managers ....... 253
  7.6 Composite UI Elements ........................................................... 254
    7.6.1 A Simple Composite UI Element ................................. 254
    7.6.2 Composite, Composite UI Elements ............................. 254
    7.6.3 Binding Table UI Elements to the Context ................... 255
    7.6.4 Runtime Behavior of a Table UI Element ..................... 256
  7.7 UI Element Events and View Controller Actions ....................... 258
    7.7.1 Declaration and Association with a UI Element Event .......... 258
    7.7.2 Implementation .......................................................... 260
    7.7.3 Action Types ............................................................... 261
    7.7.4 Event Types ............................................................... 262
    7.7.5 Coding Generated by an Action Declaration .................. 263
  7.8 Parameter Mapping ................................................................. 264
    7.8.1 Multiple UI Element Events Handled by One Action Event Handler ........................................... 267
## 7.8 Further Decoupling of the UI

- **7.8.2 Further Decoupling of the UI** ...................................... 269
- **7.8.3 Advanced Parameter Mapping Example** .......................... 271

## 7.9 Generic UI Services

- **7.9.1 Simple Value Selector** ................................................. 273
- **7.9.2 Extended Value Selector** ............................................. 273
- **7.9.3 Extending the List of Values Shown in a Simple or Extended Value Selector** ............................................. 275
- **7.9.4 Object Value Selector** ................................................. 276
- **7.9.5 Implementing an Object Value Selector** ...................... 277

## 7.10 Using Messages and the Message Manager

- **7.10.1 Location of Messages on the Screen** ............................ 278
- **7.10.2 Accessing the Message Manager** .......................... 279
- **7.10.3 Reporting a Simple Message** ................................... 279
- **7.10.4 Locale-Dependent Text at Runtime** .......................... 280
- **7.10.5 Reporting a Message Held in the Message Pool** .......... 281
- **7.10.6 Relating Messages with UI Elements** ......................... 282
- **7.10.7 Locale-Dependent Text Defined in the Java Dictionary** ................................................................. 283
- **7.10.8 Defining Placeholders Within a Message Text** ............. 285
- **7.10.9 Process Control Using the Message Manager** .............. 286

## 8 Writing Multilanguage Applications

- **8.1 Internationalization** .......................................................... 287
- **8.2 Externalization** ................................................................. 287
- **8.3 Web Dynpro i18n Concept** .................................................. 288
- **8.4 S2X: SAP’s Use of the XLIFF Standard** ............................ 289
- **8.5 Storing Language-Specific Text in XLF Files** .................. 290
- **8.6 Translating XLF Files** ...................................................... 291
- **8.7 Use of the S2X Editor Within NWDS** ............................. 291
- **8.8 Editing Message Pool XLF Files** ..................................... 292
- **8.9 Runtime Locale Determination** ........................................ 294

## Part IV: Advanced Development

## 9 Dynamic Context Manipulation

- **9.1 Dynamic Addition of an Unmapped Context Node** .......... 297
  - **9.1.1 Before the Coding Details Are Discussed, Though** ... 298
  - **9.1.2 Now for the Coding** .................................................. 299
  - **9.1.3 Dynamic Creation of a Hierarchy of Context Nodes** .... 302
### 9.2 Dynamic Addition of a Mapped Context Node

- **9.2.1 Technique 1** .......................................................... 306
- **9.2.2 Technique 2** .......................................................... 307
- **9.2.3 Method replicateNodeInfo()** ................................. 308

### 10 Advanced Use of the UI

- **10.1 Working with Tree UI Elements** ............................ 311
  - **10.1.1 Types of Tree UI Element** .................................. 311
  - **10.1.2 Building a Tree Whose Depth Is Known at Design Time** ................................................................. 312
  - **10.1.3 Building a Tree Whose Depth Is Unknown at Design Time** ................................................................. 312
  - **10.1.4 Building a Context Node Suitable for a Tree UI Element** ................................................................. 313
  - **10.1.5 Making a Tree Node Interactive** .......................... 315
  - **10.1.6 Properties of Tree UI Elements** .......................... 318
- **10.2 Efficient Use of Actions to Enable and Disable UI Elements** ..... 319
- **10.3 Programmatic Assignment of Actions to UI Element Objects** .... 320
- **10.4 Accessing Existing UI Elements in a View Layout** .......... 322
  - **10.4.1 Accessing UI Elements by Name** ......................... 323
  - **10.4.2 Accessing UI Elements Generically** ...................... 324
- **10.5 Introduction to Dynamic View Generation** ...................... 331
- **10.6 The Principles of Dynamic View Construction** .................. 332
  - **10.6.1 Is There a Simpler Way of Doing It?** ...................... 333
  - **10.6.2 The Fundamental Principles of Dynamic View Layout Construction** ......................................................... 334
- **10.7 Dynamic Construction of a UI Element Hierarchy** .............. 335
  - **10.7.1 Implementation Example** .................................... 336
  - **10.7.2 Implementation of the Method wdDoModifyView()** ... 337
  - **10.7.3 Implementation of the Method showNode()** .......... 340
  - **10.7.4 Implementation of the Method showNodeAsColumns()** ................................................................. 343
  - **10.7.5 Implementation of the Method showNodeAsTable()** ... 344
  - **10.7.6 Implementation of the Method getUIElement()** ........ 346
- **10.8 Implementing an OVS** .................................................. 348
  - **10.8.1 Basic Concept** .................................................. 349
  - **10.8.2 Assigning the OVS Extension to a Context Attribute** .... 350
  - **10.8.3 OVS Processing** .................................................. 352
  - **10.8.4 OVS Implementation** ............................................ 353
11 Navigation and Error Handling ............................................ 359
11.1 Navigation Processing ................................................... 359
  11.1.1 The View Assembly .................................................. 360
  11.1.2 Outbound Navigation Plugs ....................................... 360
  11.1.3 Inbound Navigation Plugs ......................................... 361
  11.1.4 Navigation Processing ............................................ 361
  11.1.5 Re-Entrant Navigation Links ................................... 365
11.2 Suspend and Resume Plugs ............................................. 368
  11.2.1 Declaring Suspend and Resume Plugs ......................... 369
  11.2.2 Suspend and Resume Plug Processing ......................... 372
  11.2.3 Requirements for an External Website ...................... 373
  11.2.4 Configuring Time-Out Values .................................. 374
11.3 Error Handling .................................................................. 376
  11.3.1 The Input and Output Halves of the Phase Model
      Processing ................................................................... 377
  11.3.2 Centralized Error Handling ...................................... 378
  11.3.3 Error Handling Across a Component Hierarchy .......... 382
  11.3.4 Principles for Handling Errors Trapped Before
      Navigation .................................................................. 384
  11.3.5 Principles for Handling Errors Trapped After
      Navigation .................................................................. 385

12 Components at Runtime ................................................... 387
12.1 Web Dynpro Components and Polymorphism ..................... 387
12.2 Pop-Up Windows ............................................................. 392
  12.2.1 Single Round-Trip Pop-Up Window .......................... 393
  12.2.2 Multiple Round-Trip Pop-Up Windows ....................... 397
  12.2.3 Opening an External Window ................................... 400
12.3 Application and URL Properties ....................................... 403
  12.3.1 Application Properties ............................................ 403
  12.3.2 URL Properties ....................................................... 404

13 The Adaptive RFC Layer .................................................. 407
13.1 A Brief History Lesson ................................................... 407
  13.1.1 The Origins of RFC ................................................ 407
  13.1.2 Java Connector and the Adaptive RFC Layer ............ 408
  13.1.3 The Enterprise Connector ....................................... 410
  13.1.4 The Common Model Interface ................................. 412
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.2</td>
<td>General Introduction to BAPIs</td>
<td>413</td>
</tr>
<tr>
<td>13.2.1</td>
<td>What Is a BAPI?</td>
<td>413</td>
</tr>
<tr>
<td>13.2.2</td>
<td>ABAP Function Module Interface</td>
<td>414</td>
</tr>
<tr>
<td>13.3</td>
<td>Custom-Written RFC Modules</td>
<td>417</td>
</tr>
<tr>
<td>13.4</td>
<td>Introduction to the Adaptive RFC Layer</td>
<td>418</td>
</tr>
<tr>
<td>13.5</td>
<td>Creating Adaptive RFC Models</td>
<td>421</td>
</tr>
<tr>
<td>13.6</td>
<td>Structuring a Model</td>
<td>425</td>
</tr>
<tr>
<td>13.7</td>
<td>Explanation of Generated Model Classes</td>
<td>426</td>
</tr>
<tr>
<td>13.7.1</td>
<td>Model Naming Convention</td>
<td>427</td>
</tr>
<tr>
<td>13.7.2</td>
<td>Executable and Non-Executable Model Objects</td>
<td>427</td>
</tr>
<tr>
<td>13.7.3</td>
<td>Model Structures</td>
<td>428</td>
</tr>
<tr>
<td>13.7.4</td>
<td>The Model Dictionary</td>
<td>429</td>
</tr>
<tr>
<td>13.7.5</td>
<td>Model Object Hierarchies and Relation Roles</td>
<td>431</td>
</tr>
<tr>
<td>13.8</td>
<td>Web Dynpro Models and Development Components</td>
<td>433</td>
</tr>
<tr>
<td>13.9</td>
<td>Using Model Objects in a Web Dynpro Controller</td>
<td>434</td>
</tr>
<tr>
<td>13.9.1</td>
<td>Model Objects and the Context</td>
<td>434</td>
</tr>
<tr>
<td>13.9.2</td>
<td>The Relationship Between the Contents of a Model Object and Its Corresponding Context Model Node</td>
<td>436</td>
</tr>
<tr>
<td>13.10</td>
<td>Altering a Model Object’s Logical System After It Has Been Created</td>
<td>437</td>
</tr>
<tr>
<td>13.11</td>
<td>A Simple Example Using Context Model Nodes at Runtime</td>
<td>439</td>
</tr>
<tr>
<td>13.11.1</td>
<td>Preparing the Context at Design Time</td>
<td>440</td>
</tr>
<tr>
<td>13.11.2</td>
<td>Component Controller Coding</td>
<td>441</td>
</tr>
<tr>
<td>13.11.3</td>
<td>View Controller Coding</td>
<td>450</td>
</tr>
<tr>
<td>13.11.4</td>
<td>Look at What Has Not Been Done</td>
<td>457</td>
</tr>
<tr>
<td>13.12</td>
<td>Adapting to Changes in an RFC Interface</td>
<td>459</td>
</tr>
</tbody>
</table>

14 JCo Connection Management | 467  |

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.1</td>
<td>Connecting to an SAP System</td>
<td>467</td>
</tr>
<tr>
<td>14.1.1</td>
<td>Logon Parameters</td>
<td>467</td>
</tr>
<tr>
<td>14.1.2</td>
<td>Language</td>
<td>467</td>
</tr>
<tr>
<td>14.1.3</td>
<td>Client</td>
<td>468</td>
</tr>
<tr>
<td>14.2</td>
<td>Identifying the Correct Backend System</td>
<td>469</td>
</tr>
<tr>
<td>14.2.1</td>
<td>Logical JCo Destinations</td>
<td>469</td>
</tr>
<tr>
<td>14.2.2</td>
<td>Prerequisites for Creating a Technical System</td>
<td>470</td>
</tr>
<tr>
<td>14.2.3</td>
<td>Creating a Technical System</td>
<td>471</td>
</tr>
<tr>
<td>14.2.4</td>
<td>Creating a JCo Destination</td>
<td>472</td>
</tr>
<tr>
<td>14.3</td>
<td>The JCo Pool Concept</td>
<td>474</td>
</tr>
<tr>
<td>14.3.1</td>
<td>Pool Allocation</td>
<td>474</td>
</tr>
<tr>
<td>14.3.2</td>
<td>Pool Naming Conventions</td>
<td>475</td>
</tr>
<tr>
<td>14.3.3</td>
<td>Pool Definition</td>
<td>475</td>
</tr>
</tbody>
</table>
## Contents

14.4 Pool Usage at Runtime ............................................................ 477
  14.4.1 Correct Design of Web Dynpro Applications .......... 477
  14.4.2 Incorrect Design of Web Dynpro Applications ............ 479
  14.4.3 Correcting a Poor Model Architecture ..................... 480

14.5 Model Scope Type ................................................................. 482

14.6 Calculating JCO Pool Size ...................................................... 484

14.7 Metadata Connections ........................................................... 485
  14.7.1 Pool Optimization ....................................................... 485
  14.7.2 Metadata Connection Usage ....................................... 486

14.8 Impact of JCo Connections on the SAP Gateway Process .......... 486

14.9 Relationship Between JCO Destinations and ABAP Sessions .... 489
  14.9.1 Logging on to an SAP System ...................................... 489
  14.9.2 ABAP Sessions ............................................................ 489
  14.9.3 Performing ABAP Database Updates Without
  Causing an Unwanted Session Change ............................ 491
  14.9.4 Statements to Avoid in an RFC Module ....................... 492

14.10 Avoiding the Read–Write–Read Problem ................................. 492

## Appendix

A Web Dynpro Naming Placeholders ........................................... 499
  A.1 Web Dynpro Component Architecture ............................... 499
  A.2 Web Dynpro Controllers .................................................. 499
  A.3 Model Objects ................................................................. 500
  A.4 NetWeaver Development Infrastructure Entities ............... 500
  A.5 Context Entities ............................................................... 500
  A.6 Generic and Composite Abbreviations ............................... 501
  A.7 Subscripts for Composite Placeholders Using the
  SAP Recommended Suffixes .................................................. 502
  A.8 AS Java Placeholders ........................................................ 503

B Naming Conventions .................................................................. 505
  B.1 General Rules for Naming ................................................... 505
  B.2 Naming Conventions for Coding Entities ............................. 505
    B.2.1 Applications: $\{n_a\} = \{a\}App .......................... 505
    B.2.2 Components: $\{n_c\} = \{c\}Comp ......................... 506
    B.2.3 Component Interface Views: $\{n_{civ}\} =
    \{\text{wInterfaceView} \} .............................................. 506
    B.2.4 Component Usage: $\{n_u\} = \{n_c\}\{p\}\text{Inst or}
    \{\text{cid}\}\{p\}\text{Inst} ............................................. 506
    B.2.5 Custom Controllers: $\{n_\text{cc}\} = \{cc\}Cust .......... 506
    B.2.6 Inbound Plugs: $\{n_{pi}\} = \{p_{in}\}n ..................... 506
    B.2.7 Models: $\{n_m\} = \{m\}Model ............................. 507
Contents

B.2.8 Outbound Plugs: \( n_{po} = p_{out} \) Out ........................................... 507
B.2.9 Component Interface Definition: \( n_{cid} = cid \) Compl ................................................................. 507
B.2.10 Component Interface View Definition: \( n_{civ} = n_{cid} \text{<any_name>} \) ...................................................... 507
B.2.11 Views: \( n_v = v \) View .................................................. 507
B.2.12 Viewsets: \( n_{vs} = vs \) Viewset .................................................. 508
B.2.13 Windows: \( w = n_c \) or \( n_c \text{Window or any_name} \text{Window} \) .................................................. 508
C Exercise to Display a File System Using a Recursive Context Node Structure ................................................................................................................. 509
C.1 Create a New Web Dynpro DC ................................................ 509
C.2 Create a Web Dynpro Component ........................................... 510
C.3 Create a Java Bean to Represent a Directory Entry ............... 510
C.4 Import the Java Bean as a Model ............................................. 512
C.5 Component Controller Configuration ....................................... 513
C.5.1 Declare Model Usage .................................................. 513
C.5.2 Context Configuration ................................................. 513
C.5.3 Method Declarations and Coding ................................ 514
C.6 View Controller Configuration ................................................. 517
C.6.1 Context Configuration ................................................. 517
C.6.2 Create an Action ......................................................... 517
C.6.3 Editing the View Layout .............................................. 518
C.6.4 View Controller Implementation ........................................ 518
C.7 Create an Application .............................................................. 519
C.8 Comment on the Use of Standard SAP Icons ....................... 520
D ABAP Coding ........................................................................... 521
E Dictionary Structures .......................................................................... 525
F The Author ........................................................................................ 527
Index ......................................................................................................... 529
UI Elements

The abbreviation UI stands for user interface, and a UI element is any object that is used in the construction of the screen seen by the end user.

UI elements are usually objects with which the user interacts, such as drop-down lists, checkboxes, input fields, or radio buttons. However, many non-interactive UI elements are available in Web Dynpro such as captions, horizontal gutters, and legends. It is also possible for a UI element to be completely invisible. Examples include the iFrame or a transparent container. Figure 7.1 shows how a table UI element would look in the view layout editor at design time.

**Important**

A UI element is not necessarily going to be visible. Certain UI elements are designed to act simply as containers that hold other (visible) UI elements. UI containers can hold anything from individual elements to a view controller to the entire visual interface of another component.

![Figure 7.1 An Example of a Table UI Element Seen in the NWDS at Design Time](image)

### 7.1 How are UI Elements Stored Within a View Controller?

All UI elements are represented as a hierarchy within a view controller's layout. At the top of the hierarchy, there is always a UI element called `RootUIElementContainer`. It is beneath this UI element that all other UI elements will be defined (see Figure 7.2).
The RootUIElementContainer is always of type TransparentContainer. This means this UI element is not visible.

7.2 Editing a View Layout

In this section we will look at the design time process of editing a view controller's layout.

7.2.1 Adding UI Elements to the View Layout

There are several ways of adding a UI element to the view layout:

1. By dragging a UI element from one of the categories on the left of the View Designer window across to the desired location in the view layout.
   The only drawback of this option is that the UI element will be given an automatically generated name such as InputField4 or DropDownByKey2. These are not the most meaningful names, so you will probably want to rename the UI element after it has been added, in which case it is just as quick to use the next option.

2. By dragging a UI element from one of the categories on the left of the View Designer window down to the UI element hierarchy seen in the Outline view. This has the same drawback in that the UI element will be named automatically and may therefore need to be renamed manually.

3. By right-clicking on the container UI element in the Outline view (shown in Figure 7.2) and then selecting Add Child. A pop-up window is then displayed from which you can select the required UI element. At this point you can specify the name of the UI element that is about to be added.

4. This UI element is then added at the last position within the parent UI container's list of children.
5. By right-clicking on the container UI element in the Outline view (shown in Figure 7.2) and then selecting **Apply Template**. A wizard then opens and asks you what type of template you would like to apply. It will create either a table or a set of label-input field pairs of UI elements based on the attributes that have already been defined in the context.

### 7.2.2 Editing Existing UI Elements on the View Layout

Once a UI element object has been added to the view layout, it always becomes a child of some container UI element. The view layout therefore has a hard-coded container UI element called `RootUIElementContainer` (of type `TransparentContainer`), and all UI element objects are ultimately the children of this parent object.

The UI element hierarchy is traversed in the normal top-to-bottom, left-to-right manner. If you want to move a UI element within this hierarchical structure, you can do so in one of two ways. In the Outline view of the view layout:

1. Right-click on the UI element and from the pop-up menu select **Move Up** or **Move Down**.
2. Or click on the UI element and drag it up or down in the hierarchy.

**Important**

Be careful when using the drag and drop method of repositioning a UI element within the hierarchy! Sometimes the UI elements do not want to be dragged. Furthermore, sometimes they won’t drop into the place you require.

In these situations, it’s always more reliable to use the **Move Up**, **Move Down** options from the pop-up menu.

### 7.3 Putting Data on the Screen

Two things must happen before data can be put on the screen. First, you must have defined the view controller’s context. Second, you must decide which UI elements will be used to display the data held in the context.

#### 7.3.1 Data Binding

Once these two decisions have been made, you must connect the UI element properties with the nodes and attributes in the context that contain the data. This is a process known as **data binding**, and is where you define which node or attribute in the context will act as a data source for that particular UI element property.
Can you see a familiar principle in operation here? It’s the same data generator–data consumer principle at work, except this time, it is happening within the scope of a single controller. The view controller's context is behaving as the data source (or data generator), and the UI element properties are the data consumers (see Figure 7.3).

The separation between the UI element properties and the context has been implemented to achieve client independence (see Section 2.5.3).

### 7.3.2 Binding UI Properties to the Context

Which UI element properties can be bound to the context? The answer is, almost all of them. The only UI element properties that can never be bound to the context (i.e., the only UI element properties that cannot have variable values) are the id property and the properties related to the UI element’s layout manager. Other than this, almost all UI element properties can obtain their value from the context (once a suitable data binding has been made).

In the following example, some name and address details are displayed on the screen. However, as you can see in Figure 7.4, the `readOnly` property of the `First_Name` input field contains the hard-coded value of `false` (1). This means this UI element will always be open for input (2).

By creating a dedicated context attribute (3) of the correct data type (4), you can programmatically control whether or not the `First_Name` field is open for input. This shown in Figure 7.5.
Now that a dedicated context attribute exists, the UI element’s readOnly property can be bound to this attribute (Figure 7.6, step ③). Now at runtime, the input field will only be open for input (④) if the corresponding context Boolean attribute is set to false.

Figure 7.4 A Hard-Coded Property Value Gives the UI Element a Fixed Behavior

Figure 7.5 A Dedicated Context Attribute Can Be Used To Alter the Behavior of a UI Element
There is a very important detail to notice about this new context attribute. The context node to which this attribute has been added (CurrentAddressData) is a mapped node, and all its attributes are mapped. However, the attribute First_Name_ReadOnly that has just been added is not mapped; it is local to the view controller’s context. This means the data in the First_Name_ReadOnly attribute will be local to the view controller’s context, but the rest of the data in the context will be derived from the mapping origin node living in a different controller.

It is perfectly permissible to create a context node that contains a mixture of mapped and unmapped attributes. In fact, this is a good design because even though data appears to belong to a single node, the attributes that hold the business data are mapped (i.e., they function as consumers of data from a non-visual controller), and the attributes holding data only concerned with the display of the business data are not mapped.

### 7.3.3 Controlling UI Element Behavior from the Context

Most of the properties of a UI element can be bound either to a node or an attribute of the context. The behavior of the UI element can then be controlled by manipulation of the context. Properties such as visible, enabled, and readOnly are the ones you will typically use for dynamic alteration of a user interface.
Many UI elements have an enabled property, and those capable of receiving input from the user will probably have a readOnly property. Both of these properties are Boolean and don’t affect whether a UI element is visible; they simply control whether the user can interact with it. However, if you want a UI element to be completely removed from the rendered screen, you need to manipulate the visible property. This property takes three possible values that are held in the constant WDVisibility.

- **VISIBLE**
  - Present on the screen and visible, but the user cannot necessarily interact with it
- **BLANK**
  - Present on the screen but invisible
- **NONE**
  - Completely absent from the screen

The visible property of a UI element must be bound to a context attribute of type Visibility. A context attribute can be assigned this data type as follows:

- Create a context attribute called, say, InputVisibility.1
- The default data type for any context attribute is string. Change this by clicking on the ellipsis button that appears to the right of the data type field.
- In the pop-up window, the radio button should be set to Dictionary Simple Type. Expand the tree node Local Dictionary.
- Expand the tree node com.sap.ide.webdynpro.uielementdefinitions, scroll down the list, and select Visibility.
- Create your UI element in the view layout and bind the visible property to this context attribute.

From the coding in your view controller, you will now be able to control the value of this context attribute using the values found in the WDVisibility class.

### 7.3.4 Impact of Data Binding on the Coding

The consequences of data binding mean that when you want to write some code that gets or sets the value of a UI element property, there is no need to access the UI element object itself. As long as you have bound the particular property of the UI element to a suitable attribute of the context, your coding need only manipulate the context attribute value.

Because the UI element property has been bound to the context, the Web Dynpro Framework will automatically transport the data from the context to the UI element.

---

1 This name is completely arbitrary. You could call it ChickenSoup if you want, but that would probably not be very helpful.
when the screen is rendered and then automatically transport the user's input from the UI element back to the context when a round trip is initiated.

<table>
<thead>
<tr>
<th>Caveat Confect or</th>
</tr>
</thead>
<tbody>
<tr>
<td>One of the tell-tale signs that an untrained developer has been using Web Dynpro is that when they want to put data on the screen, they write coding that obtains a direct reference to the UI element, and then they hard-code the values from the context directly into the UI element properties.</td>
</tr>
<tr>
<td>This style of coding is completely redundant 99% of the time!</td>
</tr>
<tr>
<td>Do not write coding this way, as it will greatly increase the complexity of your application without providing any benefits. This will unnecessarily increase the maintenance effort required to modify and support the application.</td>
</tr>
<tr>
<td>In other words, you've just raised the application's total cost of ownership.</td>
</tr>
<tr>
<td>Bad news. Don't do it.</td>
</tr>
</tbody>
</table>

Certain UI element properties do not correspond to the simple data types such as integer or string or boolean. Such properties are ones such as visibility or design.

If you want to create a context attribute for controlling, say, the visibility of a UI element, you should set the context attribute's data type to a class found in one of the following packages:

1. com.sap.dictionary.predefined.*
2. com.sap.ide.webdynpro.uielementdefinitions

These classes are available when you define a context attribute's data type by clicking on the ellipsis button that appears to the far right of the field when the data type is selected. Once you click on this button, you will see a pop-up window that gives you access to all the data types available from the local dictionaries (see Figure 7.7).

For instance, if you create a context attribute to control the visibility of one or more UI elements, this attribute must be of type com.sap.ide.webdynpro.uielementdefinitions.Visibility.

### 7.4 Avoiding a Possible Source of Confusion

As already stated, the Web Dynpro design paradigm is based on the separation of those parts of the program that generate data from those parts that consume data. This is the fundamental concept that underlies the more well-known principle of separating data processing from data presentation. The latter concept is just one use case of the former concept.
Avoiding a Possible Source of Confusion

Every Web Dynpro UI element has a set of properties, most of which can be bound to different context nodes or attributes. Once a UI element property has been bound to a context node or attribute, you can directly control its behavior and appearance without ever needing to reference the UI element object itself. In other words, by means of data binding, you can manipulate one or more UI elements via the data held in the context.

However, developers who are experienced in Web development will have become accustomed to ambiguities that exist within HTML. These ambiguities concern the function performed by certain HTML tag parameters.

A good example is the difference between the size and maxlength properties of the HTML `<input>` tag. The size parameter controls the visible appearance of the UI element itself, but the maxlength parameter controls the maximum number of characters you can type into that input field.

"Well, so what?" you may say. And that's exactly my point: you're so used to the ambiguity that you probably don't even recognize it as a problem. Web Dynpro is more rigorous than this, though. In Web Dynpro you are not permitted to have UI element parameters that mix the control of data presentation (e.g., the size parameter) with the control of data processing (e.g., the maxlength parameter). In Web Dynpro UI element properties strictly control the presentation of data, not the processing.

The problem originates from the fact that when Tim Berners-Lee created the first version of HTML, he never envisaged that a metadata repository would ever be needed to define the data being displayed. Consequently, such a facility does not exist in HTML, and this makes the resulting ambiguities inevitable.
This blurred distinction in the use of UI element properties is not permitted in Web Dynpro, so when you look at the length parameter of an InputField UI element, for instance, you are looking at a property that controls only the appearance of the UI element — not the length of the data it displays. The metadata of the information being displayed by the UI element is held independently in the Java dictionary.

<table>
<thead>
<tr>
<th>Important</th>
</tr>
</thead>
<tbody>
<tr>
<td>UI element parameters control only the appearance of the UI element itself, not the data being displayed.</td>
</tr>
</tbody>
</table>

To recap on some UI element design principles:

1. The properties of a UI element control only its appearance and behavior, not the data being displayed.
2. The majority of UI element properties can be bound to context nodes or attributes.
3. The metadata of the information displayed through a UI element is determined by the context node or attribute to which it is bound, which can be controlled by a structure or simple type held in the Java Dictionary.

### 7.5 Layout Managers

The purpose of a layout manager is to manage the arrangement of UI elements within their parent container. All Web Dynpro UI element containers make use of a layout manager of some sort.

Every Web Dynpro view is represented as a hierarchy of UI elements. The root UI element of this hierarchy is created automatically whenever a view controller is declared and always 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, a set of property values must be specified for each child UI element within that container. It is within the layout data object that you specify how that child UI element should appear when rendered with the given layout manager.
7.5.1 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 sequence. If more UI elements have been defined than will fit horizontally across the screen, the extra UI elements will flow onto a new row.

As you resize the window within which the FlowLayout container lives, you will see the UI elements wrap or flow automatically within the available screen space (see Figure 7.8 and Figure 7.9). It is not possible to define any form of vertical alignment within a FlowLayout container.

![Figure 7.8 UI Elements Arranged in a Container Using a Flow Layout Manager: Narrow Screen](image1)

![Figure 7.9 UI Elements Arranged in a Container Using a Flow Layout Manager: Wide Screen](image2)

7.5.2 Row Layout

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

If you want to subdivide some area of the view into horizontal rows, but you do not require any vertical alignment between the resulting columns, 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 property aggregation called `layoutdata` and determine whether the UI element will be rendered at the start of a new row or just be a row member. The default is that all child UI elements contain `RowData` objects (i.e., they do not occur at the start of a row).

If you change a child element to contain a `RowHeadData` object, you are telling the `RowLayout` layout manager that this element must be rendered at the start of 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 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 UI element on the screen.

In Figure 7.10 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 7.10 UI Elements Arranged in a Container Using a Row Layout Manager](image)

### 7.5.3 Matrix Layout

The `MatrixLayout` layout manager is a further enhancement of the capabilities of the `RowLayout` layout manager.\(^2\) A `RowLayout` layout manager allows you to specify when
new rows should start but provides no facility for the vertical alignment of elements into columns. 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 elements 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.

In a manner similar to RowLayout-managed UI containers, each child UI element assigned to a MatrixLayout-managed container will contain either a MatrixData object or MatrixHeadData object. Again, these objects are stored in the layoutdata property aggregation. The default object type is MatrixData, but if you want to start a new row, you must change this to MatrixHeadData.

In Figure 7.11 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, and the two input fields have been pushed across to the right-hand side of the screen. This is because these two UI elements occupy the second column of the grid, with the width of the first column being defined by the widest UI element it contains — the table.

### Figure 7.11 UI Elements Arranged in a Container Using a Matrix Layout Manager

#### 7.5.4 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.

<table>
<thead>
<tr>
<th>Important</th>
</tr>
</thead>
<tbody>
<tr>
<td>The time taken for browsers to render a screen rises significantly if the HTML contains multiple levels of nested &lt;table&gt; tags. Since the GridLayout and MatrixLayout layout managers are implemented in a browser using an HTML &lt;table&gt;, if possible, you should try to avoid nesting these layout managers within each other.</td>
</tr>
</tbody>
</table>

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 TransparentContainer. 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 7.12 is the same layout as seen in the previous figures, but now that the view container is using the GridLayout layout manager with the colCount parameter set to 2, all the UI elements have been assigned an arbitrary position in the table, on a left-to-right, top-to-bottom basis. Also, the table UI element has had its colSpan parameter set to 2.

![UI Elements Arranged in a Container Using a Grid Layout Manager: colCount = 2](image)

**Figure 7.12** 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 the GridLayout layout manager, you will need to pad the empty grid cells with invisible UI elements. These are shown Figure 7.13.

If you require a tabular layout for your UI elements, SAP recommends that the MatrixLayout layout manager should be used in preference to the GridLayout layout manager.
Each of the layout managers has a set of associated properties (see Table 7.1 to Table 7.4). Many of these properties are then inherited by the UI elements within the container.

<table>
<thead>
<tr>
<th>FlowLayout</th>
<th>Layout Manager Properties</th>
<th>Data Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>defaultPaddingBottom</td>
<td>String</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td></td>
<td>defaultPaddingLeft</td>
<td>String</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td></td>
<td>defaultPaddingRight</td>
<td>String</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td></td>
<td>defaultPaddingTop</td>
<td>String</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td></td>
<td>wrapping</td>
<td>Boolean</td>
<td>True</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Layout Data Properties</th>
<th>Data Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>paddingBottom</td>
<td>String</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>paddingLeft</td>
<td>String</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>paddingRight</td>
<td>String</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>paddingTop</td>
<td>String</td>
<td>&quot;&quot;</td>
</tr>
</tbody>
</table>

Table 7.1 Properties of the FlowLayout Layout Manager
### RowLayout

<table>
<thead>
<tr>
<th>Layout Head Data Properties</th>
<th>Data Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>hAlign</td>
<td>WDCellHAlign</td>
<td>LEFT</td>
</tr>
<tr>
<td>rowBackgroundDesign</td>
<td>WDCellBackgroundDesign</td>
<td>TRANSPARENT</td>
</tr>
<tr>
<td>rowDesign</td>
<td>WDLayoutCellDesign</td>
<td>R_PAD</td>
</tr>
<tr>
<td>vGutter</td>
<td>WDLayoutCellSeparator</td>
<td>NONE</td>
</tr>
</tbody>
</table>

**Table 7.2** Properties of the RowLayout Layout Manager

### MatrixLayout

<table>
<thead>
<tr>
<th>Layout Manager Properties</th>
<th>Data Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>stretchedHorizontally</td>
<td>Boolean</td>
<td>True</td>
</tr>
<tr>
<td>stretchedVertically</td>
<td>Boolean</td>
<td>True</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Layout Data Properties</th>
<th>Data Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cellBackgroundDesign</td>
<td>WDCellBackgroundDesign</td>
<td>TRANSPARENT</td>
</tr>
<tr>
<td>cellDesign</td>
<td>WDLayoutCellDesign</td>
<td>R_PAD</td>
</tr>
<tr>
<td>colSpan</td>
<td>Integer</td>
<td>1</td>
</tr>
<tr>
<td>hAlign</td>
<td>WDCellHAlign</td>
<td>LEFT</td>
</tr>
<tr>
<td>height</td>
<td>String</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>vAlign</td>
<td>WDCellVAlign</td>
<td>BASELINE</td>
</tr>
<tr>
<td>vGutter</td>
<td>WDLayoutCellSeparator</td>
<td>NONE</td>
</tr>
<tr>
<td>width</td>
<td>String</td>
<td>&quot;&quot;</td>
</tr>
</tbody>
</table>

**Table 7.3** Properties of the MatrixLayout Layout Manager

### GridLayout

<table>
<thead>
<tr>
<th>Layout Manager Properties</th>
<th>Data Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cellPadding</td>
<td>Integer</td>
<td>0</td>
</tr>
<tr>
<td>cellSpacing</td>
<td>Integer</td>
<td>0</td>
</tr>
<tr>
<td>colCount</td>
<td>Integer</td>
<td>1</td>
</tr>
<tr>
<td>stretchedHorizontally</td>
<td>Boolean</td>
<td>True</td>
</tr>
<tr>
<td>stretchedVertically</td>
<td>Boolean</td>
<td>True</td>
</tr>
</tbody>
</table>

**Table 7.4** Properties of the GridLayout Layout Manager
7.5.6 Principles for the Efficient Use of Layout Managers

Please observe and implement the following principles when using 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 UI elements into one row as you like.

5. The matrix layout allows some predefined values for cell padding. The property cellDesign can have the predefined values shown in Figure 7.14. The Standard option is also referred to as rPad.

<table>
<thead>
<tr>
<th>Layout Data Properties</th>
<th>Data Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>colSpan</td>
<td>Integer</td>
<td>1</td>
</tr>
<tr>
<td>hAlign</td>
<td>WDCellHAlign</td>
<td>LEFT</td>
</tr>
<tr>
<td>height</td>
<td>String</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>paddingBottom</td>
<td>String</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>paddingLeft</td>
<td>String</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>paddingRight</td>
<td>String</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>paddingTop</td>
<td>String</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>vAlign</td>
<td>WDCellVAlign</td>
<td>BASELINE</td>
</tr>
</tbody>
</table>

Table 7.4 Properties of the GridLayout Layout Manager (cont.)
7.6 Composite UI Elements

A composite UI element is any UI element that requires further child UI element objects to function correctly. The composite UI elements that you will use quite regularly are Group, Table, and Tree (among others).

7.6.1 A Simple Composite UI Element

A simple composite UI element is the Group. This UI element is a transparent container that has a Caption UI element defined within it to hold the text seen in the title bar (see Figure 7.15). Beyond that, you can add any other UI elements you choose. Remember that the Group UI element is an extended TransparentContainer, so there is no need to embed another transparent container before adding more UI elements.

7.6.2 Composite, Composite UI Elements

No, that’s not a misprint in the heading; there are certain composite UI elements that contain other composite UI elements as children. The Table UI element is a frequently used example of such a UI element.

You will always start with a Table UI element under which must be at least one TableColumn UI element.3 The TableColumn UI element is a composite UI element that contains a Caption and some other UI element that functions as the cell editor. Notice in Figure 7.16 that both CheckBox and TextView UI elements are used as cell editors.

3 The Table UI element must contain at least one TableColumn UI element, but there are many other UI elements that can be inserted into a Table. These include a MasterColumn (for embedding a tree into column 1), a GroupedColumn, a ToolBar, and a Popin.
By default, whichever UI element you nominate to be the cell editor will appear in every cell down that table column. Therefore, if you select a TextView UI element to be a column’s cell editor, every value in that column will be presented using this UI element.

It is possible to modify the cell editor on a per-row basis. Thus, rather than presenting a table with every cell in every row open for input, you can use a default cell editor that presents the data in a read-only manner, and then when the user selects a particular row, the cell editors can be switched to input fields.

7.6.3 Binding Table UI Elements to the Context

When a Table UI element is added to a view layout, you need to make multiple data bindings for both the table and all its child composite UI elements to function correctly.

To start with, the Table itself must be bound to the context node that is being visualized. You cannot take data from multiple context nodes and display them in a single table. The node to which the Table is bound must have multiple cardinality. That is, the cardinality must be set to 0..n or 1..n. This is shown in Figure 7.17.

Now that the Table knows which context node will act as its data supplier, each TableColumn UI element can be associated with a particular attribute. Each attribute in the context node now acts as a potential data supplier for a TableColumn UI element (see Figure 7.18).
Once a Table UI element is displayed on the screen, whichever node element is at the lead selection will be highlighted as the selected table row. If the context node does not have an element at the lead selection (i.e., lead selection = –1), the table will be displayed with no rows highlighted.

Certain UI elements raise events that cause round trips to the server, even if the UI element event has no action associated with it. The Table is such a UI element. Figure 7.19 shows that when a user selects a row from a Table UI element, this will cause a round trip to the server to take place. If the Table’s onLeadSelect event has
not been associated with an action in the view controller, only the node’s lead selection value will be updated, and control will return to the screen.\(^5\)

In certain circumstances the change of lead selection could result in the loss of data. Therefore, a Table UI element has a `selectionChangeBehaviour` parameter that controls the order in which processing takes place in the server. This property has three possible settings:

- **auto**
  First, the Web Dynpro Framework updates the node’s lead selection automatically. Then, if it exists, the associated action event handler is called. This is the default behavior.

- **manual**
  The Web Dynpro Framework does not update the node’s lead selection. If it exists, the associated action event handler method is called. It is then up to this method to change the node’s lead selection manually.

- **ifNoLoss**
  This behaves in the same manner as `auto`, but with the added condition that all data must first have been successfully copied to the context. If this condition is met, `ifNoLoss` behaves in the same way as `auto`.

All of the above situations may cause supply functions to be called if the node whose lead selection is being changed has child nodes. So even if no action handler

---

4 Row selection from a table causes a non-validating event to be raised.
5 If the node has child nodes with supply functions, the change of lead selection could trigger their execution.
is associated with the table’s `onLeadSelect` event, it is still possible for supply function coding to be executed as a result of the change in lead selection.

### 7.7 UI Element Events and View Controller Actions

In any browser-based development situation, the events raised by UI elements are the mechanism for triggering some type of processing to happen. This processing is typically implemented by adding JavaScript coding into the HTML sent to the client. Then, as a result of some action performed by the user, a JavaScript event is raised (such as `onClick`), to which the client then responds — typically by issuing an HTTP GET or POST request.

In Web Dynpro, all business processing takes place on the server, so when a UI element raises a client-side event, this will typically trigger a round trip to the server. Many Web Dynpro UI elements are capable of raising client-side events such as the button shown in Figure 7.20. These events relate to different actions the user can perform. For instance, the Table UI element can raise three different events: `onFilter`, `onLeadSelect`, and `onSort`.

Whether a round trip takes place depends on two things:

- What type of event was raised?
- Does the Web Dynpro Framework know what to do in response to the event?

Some UI element events always cause round trips to the server (such as the Table’s `onLeadSelect` event). Other events, however, only cause a round trip to the server if you have told the Web Dynpro Framework what to do as a result of that event being fired.

#### 7.7.1 Declaration and Association with a UI Element Event

The next question is, “How do you tell the Web Dynpro Framework what to do when a client-side event is fired?” This is the role of an *action*.

Actions are defined in the view controller and are the runtime objects that associate a client-side event with a server-side response. Since actions are runtime objects, they can be instantiated, enabled, and generally manipulated programmatically.

When you declare an action, you are creating an object that can be associated with any number of UI element events. Figure 7.21 shows that as a result of the action’s declaration, an action handler method will typically be created. The creation of an action handler method specific to the action is the default behavior; this can be overridden if you want.

---

6 It is quite legal to assign the same action to multiple UI elements.
Once the action has been declared, it can be associated with a UI element event. This association is shown in Figure 7.22.

Actions can be given parameters that will appear in the signature of the action event handler method. See Section 7.8 for the full discussion of the use of action parameters.
7.7.2 Implementation

In our example, an action called `Save` has been created. By default, this will lead to the creation of an action handler method called `onActionSave`. If any parameters are declared for the action, these will also be added to the signature of the action event handler method.

In general, if you create an action called `${act}`, the action event handler will be called `onAction${act}`.

```java
public void onAction${act}(IWDCustomEvent wdEvent) {
    //@@begin onAction${act}(ServerEvent)

    //@@end
}
```

Listing 7.1 Stub coding created in a view controller for a default action handler method

The action event handler is the method in which you should implement the view controller’s response to the client-side action. Figure 7.23 shows that an action associates a client-side event with a server-side event handler method.
Caveat Confector

Do not implement any business logic directly in a view controller. This is very bad design because it creates a Web Dynpro application that is difficult to maintain.

Will this style of coding:
- Cause the application to crash? No.
- Cause runtime performance issues? Not normally.
- Cause the application to be non-functional? No.
- So therefore it’s a good architecture? NO!

The problems caused by this style of coding will not normally become visible until after the software has gone live. Then, several months later, the users start to ask for modifications. Now the design problems will jump up and bite you.

The reason is that the business logic has been spread out (inconsistently) between visual and non-visual controllers, so it is difficult to know where backend interaction takes place. This makes the modification process more time-consuming and error-prone.

These problems are completely avoidable if developers are disciplined enough to follow these design principles.

If, in response to the client-side event, the program needs to call some business logic in a backend system, do not put the coding directly in the view controller’s action event handler. Instead, the action event handler should simply call a method in a non-visual controller (say, the component controller), and this method then interacts with the backend system.

Remember, view controllers are consumers of data, not generators. They are not responsible for generating the data they display; that’s the role of a non-visual controller.

7.7.3 Action Types

There are two types of action: validating and non-validating. The difference between them defines how the Web Dynpro Framework reacts when validation errors occur. When you declare an action, you define it to be either with or without validation (see Figure 7.24).
When a user wants to perform the next step of a business process, they normally enter data on the screen and then press a button. The Web Dynpro Framework then copies the data the user has entered from the UI element properties back into the context attributes of all the view controllers involved in the current View Assembly. During the copy process, this data is validated by the Java Dictionary.

If the data fails the validation test, the Web Dynpro Framework will receive an error message from the underlying Java Dictionary. If the round trip was initiated by a validating action, the Web Dynpro Framework will abort any further processing and inform the user of the error.

If, however, the round trip was initiated by a non-validating action, you are saying to the Web Dynpro Framework that even if an error occurs, you don’t care — carry on processing anyway.

Non-validating actions are essential to process events such as logoff.

### 7.7.4 Event Types

Web Dynpro UI elements can raise two different types of client-side event. These are known as primary and secondary events. As mentioned above, an action is a runtime object that can be enabled or disabled. If you disable an action object that is associated with a UI element, this will affect the behavior of the UI element. However, not all UI elements react in the same way when the actions associated with their events are enabled or disabled.

Enabling or disabling an action has different consequences depending on whether the action is linked to a primary or secondary event.

- A primary event is one whose enablement or disablement affects the entire UI element.
  
  An example of a primary event is the `onSelect` event of a checkbox. If the action associated with this event is disabled, you have disabled the entire UI element.
Only primary events will be considered as evaluation candidates during a call to `IWDViewController.requestFocus()`.

- A secondary event is one whose enablement or disablement affects only part of the functionality of a UI element.

An example of a secondary UI element event is the `onLeadSelect` event of a table. If the action associated with this event is disabled, the table will be perfectly functional, except that selecting different rows of the table will not trigger a round trip.

### 7.7.5 Coding Generated by an Action Declaration

For each action declared in a view controller, a constant will be created that allows you programmatic access to the action object. In general, for any action `$act()`, the following constant will be created:

```
IPrivate$x$.WDActionEventHandler.$act()
```

Listing 7.2 Constant generated when an action is declared

Therefore, if two actions called `GenericAction` and `RowSelect` have been declared for a view controller called `ShowOrders`, the following constants will be available:

```
IPrivateShowOrders.WDActionEventHandler.GENERIC_ACTION
IPrivateShowOrders.WDActionEventHandler.ROW_SELECT
```

Listing 7.3 Constant generated when an action is declared

---

**Important**

Action names should be declared using the standard Java naming convention for a class. That is, the words of the name are concatenated together with each word starting with an upper case letter. This mixture of upper and lower case letters in one identifier is commonly known as *camel case*.

As a result of your action name being in camel case, the resulting constant name will be converted to upper case with underscore characters inserted between the words, as shown in Table 7.5.

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Generated Constant Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>DoNodeClick</td>
<td>DO_NODE_CLICK</td>
</tr>
<tr>
<td>SaveAddress</td>
<td>SAVE_ADDRESS</td>
</tr>
<tr>
<td>Action_Name</td>
<td>ACTION__NAME</td>
</tr>
</tbody>
</table>

Table 7.5 Conversion of Camel Case Action Names to Upper Case Constant Names
If you add underscore characters into the action name, the resulting constant name will contain two adjacent underscore characters as shown in the last row of Table 7.5.

### 7.8 Parameter Mapping

To ensure that server-side controllers can react intelligently to user actions on the client, it is often necessary to pass parameters back to the server with certain client-side events. For instance, when the user selects an item from a DropDownByKey UI element, the action event handler method in the view 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.

For instance, when an `onSelect` event is fired in the client, the key of the selected item is passed as a parameter back to the server (see Figure 7.25).

> **Figure 7.25** A DropDownByKey UI Element Passes a Parameter Called key to Indicate Which Item Has Been Selected

<table>
<thead>
<tr>
<th>Important</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event parameter names are hard-coded within each UI element.</td>
</tr>
</tbody>
</table>

If an event has an associated event parameter, the UI element will automatically place a value into the event parameter. The Web Dynpro Framework does this part of the coding for you automatically; however, you must ensure that this value is received by the server-side action handler.

The next step then is to find of the names the client-side event parameters. There are two ways to do this:
Parameter Mapping

- Read the help file for the UI element in question. Look for the heading Events. It may first be necessary to open the documentation of an abstract base class before the details you require are shown.

- Look in the UI element’s class definition. The name of the parameter will be found in the comment block above the mappingOf\$\{\text{ui\_evt}\}\()\) method, where \$\{\text{ui\_evt}\}\) is the UI element event name such as OnSelect or OnToggle.

The help for any particular UI element can be accessed by right-clicking on the UI element icon in the View Designer toolbar (see Figure 7.26).

![Right mouse click on the icon to get help](image)

**Figure 7.26** Right-Click on an Icon To Get UI Element Help

Sometimes, the events are documented in the UI element’s abstract base class rather than the UI element class itself. The event parameter for the DropDownByKey UI element is shown in Figure 7.27.

In the case of the DropDownByIndex UI element, the hard-coded parameter is called index. You must implement the coding to pass this parameter from the UI element event to your action handler. This process is known as parameter mapping.

In the following example, we have a DropDownByKey UI element in the view layout. When this UI element’s onSelect event is fired in the client, a round trip to the server will take place and an action event handler method in the view controller will receive control. This method has a parameter called selectedKey. Although these parameter names are different, they refer to the same value; therefore, we must associate the two.
The purpose of parameter mapping is to ensure that the key parameter value of the DropDownByKey UI element is passed to the selectedKey parameter of the action event handler method.

```java
public static void wdDoModifyView(IPrivate ${nv} wdThis, IPrivate ${nv}.IContextNode wdContext, IWDView view, boolean firstTime) {
    //@@begin wdDoModifyView
    if (firstTime) {
        IWDDropDownByKey ddByKey = (IWDDropDownByKey)view.getElement("DropDownByKey");
        ddByKey.mappingOfOnSelect().addSourceMapping("key", "selectedKey");
    }
    //@@end
}
```

Listing 7.4 Coding to associate the DropDownByKey UI element’s key parameter with the action event handler’s selectedKey parameter

In addition to the coding that actually associates the two parameter values, there are two more things to notice about the coding in Listing 7.4:

1. The coding is contained within the `wdDoModifyView()` method of the same view controller as the action event handler method.
2. The `wdDoModifyView()` method is passed a Boolean parameter called `firstTime`. This will be set to `true` if this is the first time this instance of the view controller has had its screen rendered. In other words, if the user has never seen this screen before, `firstTime` will be set to `true`.

These coding details are not optional! First, the coding must be performed inside the `wdDoModifyView()` method because this is where you have access to the UI element hierarchy; second, you only need to map the parameters once per lifecycle of the view controller.

### 7.8.1 Multiple UI Element Events Handled by One Action Event Handler

Since the event-to-action parameter assignment is specific to the UI element and not the action, it is perfectly acceptable to generalize the use of an action handler so that it can respond to events from multiple UI elements.

Let’s say we have an action called `HandleCheckBox`. As the name implies, you would expect the associated action event handler to receive control when the `onToggle` event of a checkbox is raised. Let’s further assume that we have multiple checkboxes on the screen. It would be far more efficient to have a single event handler processing *all* the checkbox events than to write a separate event handler method for each checkbox UI element.

So now when an `onToggle` event is raised in the client, control is passed to the `onActionHandleCheckBox` event handler. The only question to be answered now is, “On which checkbox did the user click?” (see Figure 7.28).

![Figure 7.28](image_url) If Multiple UI Elements Are Assigned to the Same Action Handler, How Do You Tell Who Raised the Event?

This question is answered by adding another parameter to the `HandleCheckBox` action. This extra parameter has nothing to do with the client-side event itself; therefore, it is completely independent from the client layer.
1. Select the action `HandleCheckBox` and define a new parameter called `checkBoxId` of type `integer`.

2. The corresponding action event handler method `onActionHandleCheckBox()` will now have its signature automatically modified to include this new parameter.

3. Now that the action event handler method can receive a new parameter, these parameters must be assigned values. In effect, we have created an extra parameter that can be used for any purpose we like. In this case we will assign each checkbox UI element an arbitrary integer value.

4. The coding to do this will be found in the `wdDoModifyView()` method of the view controller in which the UI elements are defined.

The coding is as follows:

```java
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 "state"
    // This parameter is UI-element specific, and therefore
    // identical for all three checkboxes.
    cb1.mappingOfOnToggle().
        addSourceMapping("checked", "state");
    cb2.mappingOfOnToggle().
        addSourceMapping("checked", "state");
    cb3.mappingOfOnToggle().
        addSourceMapping("checked", "state");

    // Now set the checkbox ids that enable the server-
    // side event to distinguish between each checkbox.
    cb1.mappingOfOnToggle().addParameter("checkBoxId", 0);
    cb2.mappingOfOnToggle().addParameter("checkBoxId", 1);
    cb3.mappingOfOnToggle().addParameter("checkBoxId", 2);
}
```

Listing 7.5 Adding a custom parameter to identify which check box UI element was clicked

Notice that the `addSourceMapping()` method is used to associate a parameter from the event source with a parameter on the action event handler, but the `addParameter()` method is used to give a custom parameter a hard-coded value.

---

7 The data type of this parameter is deliberately chosen to be a primitive so that we can use a Java `switch` statement.
5. Now in the associated action event handler method, you can implement a finite state engine to process each UI element appropriately:

```java
public void onActionHandleCheckBox(
    IWDCustomEvent wdEvent,
    int checkBoxId) {
    //@@begin onActionHandleCheckBox(ServerEvent)

    switch(checkBoxId) {
        case 0:
            // First check box
            break;
        case 1:
            // Second check box
            break;
        case 2:
            // Third check box
            break;
        default:
            // That's odd. Why are we here?
            break;
    }
    //@@end
}
```

**Listing 7.6** Coding in the action event handler to distinguish which UI element was clicked

### 7.8.2 Further Decoupling of the UI

The degree of abstraction between the event parameter and action parameter can be taken a degree further if desired. Consider the following set of requirements:

- A context node is to be displayed as a table.
- The number of columns to be displayed is unknown until runtime.
- You want to give the user the ability to sort the table simply by clicking on a column header.

Situations like this call for a generic action handler that can sort the table based, not on some arbitrary integer value, or even 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.8

---

8 Here is another example of how important a role the context plays in Web Dynpro processing. UI elements are simply the superficial layer through which the context data is visualized. Actual data processing always takes places using the objects in the context, not the UI element objects used for visualization.
This level of abstraction 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 IWDTableColumn UI element provides a client-side event parameter called col, which is passed when the onAction event is raised. However, the col parameter value is not of much use to us because it contains the name of the UI element on which the user clicked. What we really need is the name of the context attribute to which the UI element is bound.

We will go back to our SalesOrders context node. In this situation, a table of sales orders is to be 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 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. To implement this type of parameter mapping, the following steps need to be taken:

- The standard parameter col for the table column’s onAction event is ignored. This is easy to achieve — do nothing.
- Create an action called HandleSortRequest and give it a parameter called colAttrib of type String.
- The attributes that will be displayed as table columns live in a context node called SalesOrders. As we have seen with earlier demos, the attributes of node SalesOrders are called LongText, OrderNo, SalesDate, and SalesRep.
- 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, tabCol3, and tabCol4.

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

```java
// 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");
IWDTableColumn tc4 = (IWDTableColumn)view.getElement("tabCol4");

// 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.LongText");
tc2.mappingOfOnAction().
    addParameter("colAttrib", "SalesOrders.OrderNo");
tc3.mappingOfOnAction().
    addParameter("colAttrib", "SalesOrders.SalesDate");
tc4.mappingOfOnAction().
    addParameter("colAttrib", "SalesOrders.SalesRep");

Listing 7.7 Action handler using a context attribute name

Now when the action event handler method `onActionHandleSortRequest()` 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.

### 7.8.3 Advanced Parameter Mapping Example

When processing the events raised by tree nodes, it is vitally important that the action handler knows not only the name of the node on which the user clicked, but also the exact element within that node. This is particularly important because all context nodes displayed using trees must be non-singletons. Thus, there will very likely be multiple instances of the same node name at runtime.

Therefore, parameter mapping must be done in the following manner:

1. As with all previous examples, you need to know the name of the event parameter raised by the `IWDTreeNodeType` UI element. The required event parameter is called `path` and is of type `String`.

2. Before creating the action parameter, you must identify the name of the generated node element class that the `TreeNodeType` UI element represents. For instance, an element of context node `WBSElements` in view controller `ShowProjectAsTreeView` will be called `IPrivateShowProjectAsTreeView.IWBSElementsElement`.

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

4. Assuming that the action is called `HandleNodeClick`, create a parameter called `selectedEl`. 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.

---

9 In the case of this UI element, the `onAction` event is not defined in the class `IWDTreeNodeType`, but in the base class `IWDAbstractTreeNodeType`. 

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

```java
// Get a reference to the tree node WBS_Els
IWDTreeNodeType tn =
(IWDTreeNodeType)view.getElement("WBS_Els");
 tn.mappingOfOnAction().addSourceMapping("path", "selectedEl");
```

**Listing 7.8** Action handler using a context node reference

6. 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`.

7. Before the value of `path` is transferred to the server-side action parameter `selectedNodeElement`, the Web Dynpro Framework 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.

8. The action handler method now has an object reference to the exact node element on which the user clicked.

### 7.9 Generic UI Services

Any time a context attribute of a certain data type is bound to, say, an `InputField` UI element, if it is available, a UI service appropriate for that data type will be provided to help you enter data. For instance, if you have a context attribute of type `date` and you display this value through an `InputField`, Web Dynpro will immediately create a date picker UI service to help you enter a valid date.\(^{10}\)

In addition to the date picker, three generic UI services are available for use. Which one you use depends first on whether the context attribute possesses an enumeration and, second, on the type of UI element being used.

The three types of generic UI service are:

- The Simple Value Selector
- The Extended Value Selector
- The Object Value Selector

The values displayed by a Simple or an Extended Value Selector can be obtained as a static list from the Java Dictionary, the values can be constructed dynamically at runtime, or you could have a combination of static and dynamic values.

---

\(^{10}\) As opposed to something like October 32\(^{nd}\).
7.9.1 Simple Value Selector

The Simple Value Selector is the name given to the values that appear in a drop-down list. The runtime part of Figure 7.29 shows an example.

To obtain this list, you must do the following:
1. Create a dictionary simple type containing an enumeration. (e.g., Color).
2. Create a context attribute based on this simple type.
3. Bind the selectedKey property of a DropDownByKey UI element to the context attribute.

This chain of connections is illustrated below in Figure 7.29.

A Simple Value Selector should be used when the user is presented with no more than 30 or so options. Any more options than this and this method for presenting the user with a choice of inputs becomes increasingly user unfriendly.

7.9.2 Extended Value Selector

An Extended Value Selector is created in exactly the same way as a Simple Value Selector, the only difference being that an InputField UI element is used to display the data rather than a DropDownByKey UI element. Figure 7.30 shows an example of the Extended Value Selector including the process of defining its values.
The Extended Value Selector creates a pop-up table on the screen. The table always has two columns called **Key** and **Display Text**. At the top of each of these columns is a filter field. Any value entered here will act as a filter, reducing the number of visible rows in the table to only those that match the filter value.

One thing that must be understood about the Extended Value Selector is that once the user has selected the required value, the **Key** value is placed in the input field rather than the **Display Text** value. This may or may not be what the user expects or wants.

In my experience, the Extended Value Selector is either loved or hated. It is hated by some users because the **Key** value placed back in the input field is a technical value that has no meaning to them. Other users love it because the text descriptions from which they are selecting contain duplicates. Therefore, to ensure that the correct value has been selected, the users need to see the **Key** value.

Since it is not possible to replace the **Key** value with the **Display Text** value in the input field once the user has selected the required value, you should carefully evaluate the user reaction to such a UI element before building it into your visual interface.

**Figure 7.30** Defining the Values for an Extended Value Selector

The Extended Value Selector creates a pop-up table on the screen. The table always has two columns called **Key** and **Display Text**. At the top of each of these columns is a filter field. Any value entered here will act as a filter, reducing the number of visible rows in the table to only those that match the filter value.

One thing that must be understood about the Extended Value Selector is that once the user has selected the required value, the **Key** value is placed in the input field rather than the **Display Text** value. This may or may not be what the user expects or wants.

In my experience, the Extended Value Selector is either loved or hated. It is hated by some users because the **Key** value placed back in the input field is a technical value that has no meaning to them. Other users love it because the text descriptions from which they are selecting contain duplicates. Therefore, to ensure that the correct value has been selected, the users need to see the **Key** value.

Since it is not possible to replace the **Key** value with the **Display Text** value in the input field once the user has selected the required value, you should carefully evaluate the user reaction to such a UI element before building it into your visual interface.
### 7.9.3 Extending the List of Values Shown in a Simple or Extended Value Selector

As mentioned earlier, a Simple or an Extended Value Selector obtains a list of values from either:

- An enumeration defined in the Java Dictionary
- A set of dynamic values defined at runtime
- A set of static values that have been supplemented with dynamic values at runtime

If a context attribute is defined on a Simple Type found in the Java Dictionary, and this Simple Type has an enumeration defined for it, then the values from the enumeration will *always* be presented to the user as a Simple or Extended Value Selector.

Any enumeration values held in the Java Dictionary are immutable.

However, it is quite possible to create an enumeration dynamically at runtime. The following coding shows how to extend the range of colors seen in the above example.

```java
// Get the metadata for node SelectionValues
IWDNodeInfo selValsNodeInfo = wdContext.nodeSelectionValues().getNodeInfo();
// Get the metadata for the child attribute Colours
IWDAttributeInfo attrInfo = selValsNodeInfo.getAttribute("Colour");

// Get the modifiable simple type for this attribute
ISimpleTypeModifiable msType = attrInfo.getModifiableSimpleType();

// Set the column header caption seen in the EVS table
msType.setFieldLabel("Alt Colours");

// Get the modifiable simple type's value set
IModifiableSimpleValueSet valueSet = msType.getSVServices().getModifiableSimpleValueSet();

// Define a new value set
valueSet.put("Colour_10", "Pink");
valueSet.put("Colour_11", "Tan");
valueSet.put("Colour_12", "Saffron");
valueSet.put("Colour_13", "Lime");
valueSet.put("Colour_14", "Sky Blue Pink");
valueSet.put("Colour_15", "Turquoise");
valueSet.put("Colour_16", "Purple");
```

**Listing 7.9** Creating a dynamic set of values seen in a Simple or Extended Value Selector
Now, when the Colors context attribute is displayed, you will see not only the immutable set of values held in the Java Dictionary enumeration, but you will also see the additional values created by the above coding.

7.9.4 Object Value Selector

In many application scenarios a type of value help is needed in which the user searches for objects instead of values. In the example shown in Figure 7.31, the users are required to enter a customer id. If they do not know the correct value, the Object Value Selector will allow them to enter search criteria and then perform a search of a backend system. A table of results is then presented, from which the user can choose the item they require.

![Figure 7.31 The User Must Enter a Customer ID, but They Only Know the Customer’s Name](image)

In contrast to the Simple and Extended Value Selectors, the Object Value Selector is only partially based on a declarative approach. For embedding this type of value help into your Web Dynpro application you have to write a class that implements the `IWDOVSContextNotificationListener` interface. Once defined, the Web Dynpro Framework automatically renders a generic Object Value Selector pop-up window for you (see Figure 7.32).
Implementing an Object Value Selector

The following steps are needed to implement an Object Value Selector (OVS):

1. Create a custom controller that can supply an instance of your OVS object.
2. The custom controller must instantiate a subclass that implements `IWDOVSContextNotificationListener`. This requires an implementation of the following three methods:
   - `applyInputValues(IWDNodeElement datafromApp, IWDNodeElement inputToOvsQuery)`
   - `onQuery(IWDNodeElement inputToOvsQuery, IWDNode ovsQueryResults)`
   - `applyResult(IWDNodeElement dataForApp, IWDNodeElement selectedOvsResultElement)`
3. In the `wdDoInit()` method of the view controller, you must associate the relevant context attribute with the instance of the OVS object created by your custom controller. This is done by calling the following method:

   ```java
   WDValueServices.addOVSExtension(IWDAttributeInfo[] startupAttributes, 
   IWDNode queryInputNode, 
   IWDNode queryResultNode, 
   IWDOVSContextNotificationListener queryListener)
   ```

Figure 7.32 The OVS Pop-Up, Created by the Web Dynpro Framework
The input field will now be rendered with an OVS icon next to it. When the user clicks on it, a Value Help service event is raised, and the Web Dynpro Framework will then handle the pop-up screen and interaction with the OVS object for you. See Section 4.4.5 for more details on how service events are handled during Phase Model processing.

Since the implementation of this type of Value Selector is more involved, the details will be discussed in Section 10.8, where more advanced features of the user interface are explained.

7.10 Using Messages and the Message Manager

The MessageManager is the class that handles all messages for an entire component. Only one instance of the MessageManager is ever needed per Web Dynpro component.

7.10.1 Location of Messages on the Screen

Every view layout in Web Dynpro contains an implicit location in which messages will be displayed. By default, all messages will be displayed at the bottom of the screen as shown in Figure 7.33. However, it is quite possible to move this location.

Figure 7.33  The Default Location for Messages Is at the Bottom of the Screen
The message location can be moved by adding a `MessageArea` UI element to the view layout. Wherever this UI element is positioned, all messages for that View Assembly will be displayed at that location. It is often useful to insert a `MessageArea` UI element very near the top of your view layout. This means the user will not need to scroll the screen down to the bottom to see any messages that may be displayed.

**Important**

If defined, only one `MessageArea` UI element is required per View Assembly. If you have multiple `MessageArea` UI elements defined, the Web Dynpro Framework will use the first one it encounters for the message location, and the others will be ignored.

### 7.10.2 Accessing the Message Manager

From a programmatic point of view, there is only one instance of the `IWDMessageManager` class per component instance. It is therefore worth while to create a class-wide reference to the message manager in your component's component controller.

```java
//@@begin others
IWDMessageManager msgMgr;
//@@end
```

**Listing 7.10** Declare a class-wide reference to the message manager

The object reference can then be set by calling the following code in the `wdDoInit()` method of the component controller.

```java
public void wdDoInit() {
    //@@begin wdDoInit()
    msgMgr = wdComponentAPI.getMessageManager();
    //@@end
}
```

**Listing 7.11** Defining the message manager object reference

Notice that the shortcut variable used to create this object reference is `wdComponentAPI`, not `wdThis`.

### 7.10.3 Reporting a Simple Message

The simplest way to place a message on the screen is to call the message manager's `reportSuccess()` method. This method simply requires a string parameter as the message text.

```java
msgMgr.reportSuccess("Everything is fine");
```

**Listing 7.12** Using the message manager to display a simple message
When the screen is next rendered, the text string “Everything is fine” will appear in the message location for the current View Assembly.

7.10.4 Locale-Dependent Text at Runtime

It is not good practice, however, to hard-code text strings into your coding — especially if your application needs to support multiple languages. Therefore, Web Dynpro gives you the ability to create locale-dependent message objects within your component.

Each Web Dynpro component has something called a message pool, and within this pool you may create four types of message:

- Standard
- Warning
- Error
- Text

Important

For developers with ABAP programming experience, Web Dynpro messages are somewhat different from ABAP messages in that in Web Dynpro, the message type (i.e., Success, Warning, or Error) is part of the message object’s definition. This stands in contrast to ABAP, in which a message is just a predefined text string that has a type assigned to it at runtime when the MESSAGE statement is used.

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

To extract the information from an object of type IWDMessage, it is first necessary to cast it as a WDMessage object; then you will have access to all the message properties. In the following example, a message called ErrorMsg has been created in the message pool of component $ {nc}. This leads to the creation of a constant in class IMessage$ {nc} called ERROR_MSG.

```java
WDMessage errMsg = (WDMessage)IMessage$ {nc}.ERROR_MSG;

// Get the component name, the message key
// and the message text
String msgComp = errMsg.getCompName();
String msgKey = errMsg.getMessageKey();
String msgText = errMsg.getMessageText();

if (errMsg.getType().equals(WDMessageType.ERROR)) {
    // Its an error message
```
else if (errMsg.getType().equals(WDMessageType.WARNING)) {
    // Its a warning message
}
else if (errMsg.getType().equals(WDMessageType.STANDARD)) {
    // Its a standard message
}

Listing 7.13 Obtaining the properties of a message object

<table>
<thead>
<tr>
<th>Important</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only message pool messages of type Standard, Warning, and Error are addressable as constants in the generated class IMessage$.nc$.java. Messages of type text do not appear in this generated class.</td>
</tr>
</tbody>
</table>

Messages of type Text are not accessible to the IWDMessageManager class; instead, you can access them through 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.

7.10.5 Reporting a Message Held in the Message Pool

For applications in which multi-language support is needed, you should always make use of the message pool as a repository for message texts. This allows the text to be translated without you having to modify your coding in any way.

The following generalized code fragment shows how to call the message manager so that it displays a message derived from the message pool.

```
IWDMessageManager msgMgr =
    wdComponentAPI.getMessageManager();

    new Object[] {p1,...,pn},
    boolean);
```

Listing 7.14 Display a message held in the component’s message pool

In the above example, we can make direct reference to the message text by means of the message constant created for us in class IMessage$.nc$.MessageKey (where $.(nc).$ is the name of the current Web Dynpro component).

The object array is a list of parameters to be passed to the message, and the Boolean parameter tells the Web Dynpro Framework whether the entire navigation queue should be aborted. See Chapter 11 for more details on navigation processing.
7.10.6 Relating Messages with UI Elements

Very often, when an error message is sent to the screen, it will be as a result of some kind of erroneous input made by the user. Therefore, to help the user identify which field (or fields) contain the erroneous value(s), the relevant UI elements can be highlighted when the message is displayed (see Figure 7.34).

Although this may seem counterintuitive, when you want a UI element to be highlighted with the display of an error message, you don’t need to know anything about the UI element — the Web Dynpro Framework works that out for you.

Remember that UI elements are simply the means by which context data is visualized. Ask yourself this question: “Where does the data live that is visualized on the screen? In the UI element object itself or in the context?”

Therefore, to cause a UI element to be highlighted when an error message is displayed, you only need to know which attribute in the context holds the erroneous value. The Web Dynpro Framework then works out which UI element is bound to this attribute and highlights it for you. Remember, the UI element is only the superficial skin on top of the data held in the context.

Figure 7.34 The Associated UI Elements Are Highlighted in Red with the Message

11 If you said “In the UI element object,” then you’ve haven’t yet grasped a fundamental concept of Web Dynpro. Go back and reread Section 7.4.
The following code shows how to call the message manager so that in addition to the display of an error message, the UI element being used to visualize the context attribute holding the erroneous value is also highlighted. `${cn}` and `${ca}` are the respective names of the context node and attribute in question.

```java
IWDMessagerManager msgMgr =
    wdComponentAPI.getMessageManager();
IWNode errNode = wdContext.node${cn}();
IWNodeElement errEl = errNode.
    current${cn}Element();
IWDAttributeInfo errAttr = errNode.
    getNodeInfo().
    getAttribute("${ca}");
msgMgr.reportContextAttributeMessage(errEl,
    errAttr,
    IMessage${nc}.(MessageKey).
    new Object[] {p1,...,pn},
    boolean);  
```

**Listing 7.15** Highlighting a UI element when an error message is displayed is done by identifying the associated context attribute.

In this code fragment, the last three parameters are exactly the same as the parameters used in Listing 7.14. However, there are two new parameters. The first parameter is a reference to the element in the context node that holds the erroneous value, and the second parameter is the metadata of the context attribute in error.

If you think of a context node as being like a table with rows and columns, then the element reference identifies the table row, and the context attribute metadata reference identifies the table column. These two values taken together exactly identify the specific attribute holding the erroneous value.

### 7.10.7 Locale-Dependent Text Defined in the Java Dictionary

In addition to the four categories of message described above is a fifth category of locale-dependent text within a Web Dynpro application. These are texts that belong to dictionary simple types, and they can be accessed through the ISimpleType interface (see Figure 7.35).

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

The following example assumes that a message called Message1 of type text has been defined in a component’s message pool. To access this message, you need to use the following code.
// Get the text accessor from the current component
IWDTextAccessor textAccessor =
    wdComponentAPI().getTextAccessor();

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

Listing 7.16 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 IWDMessageManager interface is to supply the user with informative messages about the success or failure of the application’s functionality.

The code fragment below assumes a message called Message2 of type error is in the message pool. This message will be reported to the user with no parameters (the null parameter value), and navigation will not be cancelled as a result of the error (the false parameter value).

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

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

Listing 7.17 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:
7.10.8 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. Messages containing such placeholders are known as message text patterns.

For instance, if you are writing an application that creates business documents, you 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 7.36.

If you want to have more than one placeholder within the text message, simply increment the placeholder number as shown in Figure 7.37, where \(0\) is the document number, \(1\) is the time, and \(2\) is the date.

<table>
<thead>
<tr>
<th>Message Key</th>
<th>Message Type</th>
<th>Message Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>DocumentCreated</td>
<td>standard</td>
<td>Document (0) successfully created</td>
</tr>
</tbody>
</table>

Figure 7.36 A Placeholder in a Text Message

<table>
<thead>
<tr>
<th>Message Key</th>
<th>Message Type</th>
<th>Message Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>DocumentCreated</td>
<td>standard</td>
<td>Document (0) successfully created at (1) on (2)</td>
</tr>
</tbody>
</table>

Figure 7.37 Multiple Placeholders in a Text Message

Caveat Confector

All message placeholders must be sequentially numbered integers, and their 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:

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

// Get the document number from somewhere...
String docNo = getDocumentNo();
```
Listing 7.18 Code fragment for issuing a standard message with placeholder parameters

Please be aware that in Listing 7.17 and Listing 7.18 that the class name for the IMessage class has been generalized.

7.10.9 Process Control Using the Message Manager

The message manager must be used carefully; otherwise you could find that subsequent processing logic becomes disrupted — especially if you are implementing centralized error and navigation processing (see Chapter 11). There are two reasons for this:

1. Many of the message manager methods have a Boolean parameter called cancelNavigation. If this is set to true, no matter what other outbound navigation plugs are fired, processing of the entire navigation queue is aborted.

   If you are attempting to perform centralized error and navigation management and then you allow a view controller to issue a message with the cancelNavigation parameter set to true, you will find that your centralized processing logic will have been circumvented.

2. Any time you call a message manager method that starts with the word raise, you will immediately terminate all further application processing. These methods hand control to the Web Dynpro Framework, and it is never handed back again. Such methods should only be used in the event of very serious or fatal error situations.

   For non-fatal errors, always use a message manager method that starts with the word report.

There is one exception to the rule about methods starting with the word raise. The message manager has a method called raisePendingException() that can be used to act as a conditional exit point. This means that if method calls to the message manager have been made previously, then calling this method will conditionally terminate the round trip. If no pending messages are found in the message manager, this method exits silently.\(^{12}\)

See Section 11.3 for more information on how error and navigation processing can be disrupted by careless use of the message manager.

\(^{12}\) Call me old-fashioned, but I prefer not to use a method such as raisePendingException() because I believe a program should have only one entry point and one exit point. The use of this method creates an extra, conditional exit point.
Index

//@@begin others   337, 343
<SID>      502
7-bit ASCII 289

A

ABAP conversion routines 420
ABAP data dictionary 420
  Value tables 420
ABAP Dictionary 411
ABAP Dictionary Component Types 430
ABAP Dictionary structure 430
ABAP function module 39, 62, 468
ABAP function module interface 414
ABAP session change 414
ABAP sessions 489
ABAP Workbench 417
Accessing existing UI elements in a view layout 322
Accessing UI elements
  Generic traversal mechanism 327
  Generically 324
  Simple traversal mechanism 325
Action event handler 313, 316, 319, 320
Action object 262, 263
Actions 258, 499
  Action handler method 260
  Action types 261
  Advanced parameter mapping 271
  Event types 262
  Further parameter mapping 269
  Many UI element events, one action handler 267
  Parameter mapping 264, 265
Adapting to changes in an RFC interface 459
Adaptive RFC layer 407, 409, 412, 418
  Introduction 418
Adaptive RFC model 419
Application 499, 502, 505
Application and URL Properties 403
Application properties 403
Architectural concepts of the NWDI 95
Assembler 408

B

BAPI 407
BAPI Programming Guide 414
Berners-Lee, Tim 245
Building a context node suitable for a tree UI element 313
Building a tree whose depth is known at design time 312
Building a tree whose depth is unknown at design time 312
Building Web Dynpro applications using the NWDI 98

C

Calculated Attribute
  Getter method 182
  Setter method 183, 185
Calculating JCO pool size 484
cancelNavigation flag 363
Cascading Style Sheets 34, 39
Centralized Error Handling 378
Change Management Service 97
CHANGING parameters 416
Child Development Components 92
CICS 407
CICS transaction server 407
CIM data 470
Client Abstraction Layer 168
Client/Server 407
Client-side events
  Primary 262
  Secondary 262
Code reusability 89
Common Information Model (CIM) 470
Common Model Interface 412
Common Programming Interface for Communications 407
Component 499, 506
Component Build Service 97
Component Controller 60, 61, 62, 63, 73, 78, 79, 80, 81, 82, 85, 502
Component instantiation 143
Component interface definition 388, 499, 502, 507
Component interface view 502, 506
Component interface view definition 507
Component usage 502, 506
Component usage purpose 501
Components at Runtime 387
Composite placeholders 502
Composite UI elements
Composite 254
Data Binding 255
Runtime behaviour 256
Simple 254
Confirmation Dialog 393
Connecting to an SAP system 467
Client 468
Language 467
Logon Parameters 467
Context 55, 63, 68, 69, 73, 75, 76, 77, 78, 79, 80, 81, 82, 163, 179, 191
Accessing a node element 214
Accessing all elements in a node 215
Accessing non-singleton nodes using the generic API 228, 229
Accessing singleton nodes using the generic API 227
Adding elements to a model node 230
at Design Time 163
Attribute 69, 73, 76, 77, 81, 82, 163, 164, 179, 205
Attribute Pointer 179, 180
Binary Attributes 167
Binding elements to a model node 232
Byte Code Minimizer 210, 211
Calculated Attribute 180, 182, 183, 185
Child Node 163, 165, 175, 176, 177, 199, 200, 205
Coding Principles 200
Context Mapping 79, 80
Datatypes of nodes and attributes 164
Dependent Nodes 164, 168, 169, 176, 201, 205
Difference between Value Nodes and Model Nodes 169
Dispose Function 194, 195, 196, 197, 198, 199
External Mapping 81
Context (cont.)
Fundamental Principles 205
Generic API 224, 225, 226
Independent Attributes 169
Independent Nodes 168, 169, 176, 201, 204
Internal Mapping 81
Java Dictionary 165, 169, 170
Lead Selection 175, 183, 185, 189, 190, 191, 192, 199, 200, 205
Mapped Nodes 80, 205
Mapping Origin 80, 81
Mapping Origin Node 205
Model Binding 170
Model Node 164, 170, 178
Model Object 234
Naming Conventions 178
Node 69, 76, 80, 81, 82, 163, 177, 179, 202, 205
Node access in the hierarchy 224
Node Collection 163, 172, 173, 174, 179
Node Metadata 169, 173
Node Names 177
Node Properties 171
Node Terminology 168
Non-singleton child node 208, 224
Parent Node 175, 200
Recursive Node 200, 201, 202, 233
Root Node 163, 164, 168, 169, 176, 201, 202, 204
Singleton child node 208, 223, 224
Structure 163
Structure at runtime 207
Structure binding 165, 166, 169
Supply Function 176, 189, 190, 191, 192, 193, 194, 195, 197, 198, 199, 200, 205, 218, 221, 228
Supplying data to UI Elements 167
Typed API 207, 209, 213
UI Elements 167, 191
Unmapped Node 205
Value Attributes 169
Value Node 165, 166, 169, 170
Working with non-singleton nodes 221
Working with singleton nodes 218
Context API 171
Context at runtime
Using the WDCopyService 233
Index

Context at runtime (cont.)
  Working with Recursive Nodes  233
Context attribute  500
Context child node  500
Context entities  500
Context Mapping
  Data Modeller  203
  External Mapping  187, 189
  Internal Mapping  186
  isInputElement  187
  Mapped Node  186
  Mapping Origin Node  186, 187, 195, 204
  Recursive Nodes  203
Context Metadata  171
Context Node  500
  Base Class  173, 175, 225
  Contents flagged as dirty  228
  Metadata Class  173, 213
Context Node Properties
  Cardinality  171, 172, 174, 177, 190, 194, 202, 205
  Non-singleton  175, 204, 205
  Selection Cardinality  174
  Singleton  171, 175, 176, 189, 190, 205
typedAccessRequired  178, 179, 207, 209, 210, 211, 224
Context Structure
  Dynamic nodes  207
  Static nodes  207
Context visibility  207
Controller Architecture
  Component Usage  67
  Context  68
  Controller Usage  66
  Events  71, 72, 77
  Instance Methods  70
  Model Usage  67
  Non-visual controllers  65
  Required Controllers  66, 67, 79
  Usage declarations  79
  Visual controllers  72
Controllers
  Generator and consumer  45
CPI-C  407
createConfirmationDialog()  395
createConfirmationWindow()  395
createNewAddress()  448
Creating a JCo destination  472
Creating a technical system  471
  Prerequisites  470
Creating Adaptive RFC Model Objects  421
CSS  62
Custom Controller  61, 62, 499, 502, 506
Custom written RFC modules  417

D

Data generator/data consumer principle  331
Data Modeller  434, 440
Database update  491, 492
DC Metadata  433
Default HTTP port  470
deleteAddress()  449
Design Time Repository  94, 96
Development Component  90, 500
Development Component naming conventions  91
Development Component Project  98
Dictionary domain  420
Dictionary Simple Types  429
doBeforeNavigation  361
Dynamic addition of a mapped context node  303
Dynamic addition of an unmapped context node  297
Dynamic context manipulation  297
Dynamic creation of a hierarchy of context nodes  302
Dynamic nodes  297
Dynamic UI element hierarchy
  Construction  335
  Implementation example  336
  Implementation in wdDoModifyView()  337
  Implementation of getUIElement()  346
  Implementation of showNode()  340
  Implementation of showNodeAsColumns()  343
  Implementation of showNodeAsTable()  344
Dynamic UI generation  332
Dynamic view generation
  Introduction  331
Efficient use of actions 319
Empty view 365
Enterprise Archive 37
Enterprise Connector 410, 411
Error Handling 376
Centralised handling of errors that occur in child components 383
Error handling across a component hierarchy 382
Message Managers and message scope 383
Principles for handling errors trapped after navigation 385
Principles for handling errors trapped before navigation 384
Reporting errors from a view controller to the component controller 379
EXCEPTIONS 416
Executable model objects 427
EXPORTING parameters 415
External Window 400
Externalization 287
Externalize Strings 281, 288
Faceless Component 62
Flow layout 247
Focus Groups 38, 47
Full Component Hierarchy 93
Function group 413
Function module 413
General introduction to BAPIs 413
General rules for naming 505
Generated Model Classes 426
Generic and composite abbreviations 501
Generic UI Services 272
Extended Value Selector 273
Extending values sets 275
Implementing an Object Value Selector 277
Object Value Selector 276
Simple Value Selector 273
Grid layout 249
Group UI element 454
Holtzblatt, Karen 38, 56, 59, 73
HTML 34, 39, 58, 62, 245, 247, 253, 258, 505
HTTP 370
HTTP GET 258, 368, 404
HTTP POST 258, 368, 404
HTTP request 86
I18n 287
IBM Mainframe 407
lBroker 421
Identifying a specific node attribute 179
Implementing an Object Value Selector 348
Basic Concept 349
Implementation 353
OVS Processing 352
IMPORTING parameters 415
Inbound Navigation Plugs 361
Inbound plug method 364
Inbound plug parameter 405
Inbound plugs 499, 502, 506
Inner Class 92
Instance ID 502
Interface Controller 60, 61, 78, 81, 88, 187
Interface metadata 468
Interface View Controller 62, 63, 64, 65, 72, 73, 74, 75, 86, 87, 359, 364, 369, 405
Internationalization 287
IWDAbstractTreeNodeType 318
IWDAction 319
IWDAtributeInfo 351
IWDMessageManager 363, 377, 379, 383
IWDD-node.bind() 442
IWDDnodeInfo 300
IWDOVSContextNotificationListener 276, 351
IWDTree 318
IWDTreeNodeType 318
J

Java Connectivity Builder 410
Java Connector 408
Java Dictionary 147, 148, 430, 501
Java package name 91, 501
Java RFC API 408
Java Server Page 30, 46
Java Virtual Machine 175
JavaScript 34, 39, 58, 62, 258
JCo 408, 412
JCo connection 479
JCo connection management 467
JCo destination 472
Relationship to ABAP sessions 489
JCo layer 475
JCo pool 474
JCo Pool Concept 474
JRFC.jar 408

L

Language ISO code 467
Layout manager 240, 246, 247, 248, 249, 250, 251, 253
Principles for efficient use 253
Layout Manager Properties 251
Lazy data access 61
Locale value 501
Logical destination names 421
Logical Dictionary 429
Logical JCo Destinations 469
Logical unit of work 489
LUW 489

M

Mapping origin node 303, 304, 305
Matrix layout 248
Message 499
Message Manager 278
Accessing the Message Manager 279
cancelNavigation parameter 385
Defining placeholders within a message text 285
Locale-dependent text at runtime 280
Locale-dependent text defined in the Java Dictionary 283
Location of messages on the screen 278
Message Manager (cont.)
Process control 286
raise...() methods 286
raisePendingException() 286, 385
Relating messages with UI elements 282
report...() methods 286
Reporting a message held in the message pool 281
Reporting a simple message 279
Message Pool XLF files 292
Message text patterns 285
Metadata connection usage 486
Metadata connections 485
Modal Window 397
Model 502, 507
Data generator 44
Model attribute 500
Model DC 433
Model Dictionary 429
Model Naming Convention 427
Model node 313, 500
Model Object 164, 170, 171, 178, 185, 500
Altering the logical system name 437
Relationship to context model nodes 436
Use in the context 434
Model object hierarchies 431
Model re-import 419
Model Scope Type 482
Model structures 428
Model wizard 421
Model-View-Controller
Component 54, 55
Controller 43, 54
Data consumers 44
Data generators 44, 46
Data suppliers 46
Design concept 34, 40
Editor 41, 42, 53, 54, 56, 57
Essence of the design concept 43
Model 41, 43
SAP's modification 54
User's mental model 40
View 43, 54

N

Naming conventions 505
Navigation and View Initialisation 361
Navigation link 360
Navigation plugs 360
Navigation processing 359, 361
Navigation queue 361
NetWeaver Developer Studio 117, 164, 165, 178, 207, 209, 211, 212, 281, 284, 288, 291, 331, 419, 421, 427, 428, 469, 499, 505
NetWeaver Development Infrastructure 89, 93, 500
   A brief introduction 95
   Development Component 121
   Public Part 121
Nightly builds 95
Node Cardinality 172
Node Selection Cardinality 174
Nodes 163
Non-executable model objects 427

O

Object Value Selector 62
onAction 316
onLoadChildren 313, 315
Origins of Remote Function Call 407
Outbound Navigation Plugs 360
Outbound plug 499, 502, 507

P

Package name 501
Phase Model 141, 142, 143, 144, 145, 147, 149, 151, 152, 153, 154, 155, 156, 157, 159, 312, 336, 361, 377
Action 147, 148, 149, 151, 154, 158
Action Event Handler 149
Action Handler Method 152
Application event handlers 151, 384
Client Abstraction Layer 156
Client Specific UI Validation 156, 386
Data binding 156
doApplicationStateChange 142, 145, 146, 151, 158
doBeforeNavigation 152, 153, 156, 384
doPostProcessing 157, 159
Dynamic UI manipulation 155, 386
Hook Methods 143, 144, 150, 152
Inbound plug 146, 153, 154, 155
Interface View Controller 144, 146, 151
Phase Model (cont.)
   Lifecycle Hook Methods 143
   MessageManager 155, 157, 159
   Modal pop-up window 144
   Navigation and view initialization 153, 378
   Navigation queue 153, 154, 155
   Non-validating Action 148, 149
   Object Value Selector 151
   Outbound plug 151, 153, 154, 155, 158
   Response rendering 159
   Resume request 145
   Root Component 143, 144, 146, 150, 152, 153
   Supply function 149, 150, 156, 157
   System Event Handlers 149
   Timeout 146
   Transport Data to Data Container 145
   UI elements 147, 148, 155
   Validate and Transport Data 147
   Validating Action 147, 148, 149
   Validation error 147
   View Initialisation 386
   wdDoApplicationStateChange() 144
   wdDoBeforeNavigation() 144, 152
   wdDoExit() 143, 144, 150
   wdDoInit() 143, 144, 155
   wdDoModifyView() 144, 155, 156, 158
   wdDoPostProcessing() 144, 157
   wdPostProcessing() 157
   Web Dynpro Framework 141, 143, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159
Phase Model Processing
   Input and Output halves 377
Pool Allocation 474
Pool Definition 475
   Connection Timeout 476
   Maximum Connections 475
   Maximum Pool Size 475
   Maximum Wait Time 476
Pool Naming Conventions 475
Pool optimization 485
Pool usage at runtime 477
Pop-up Windows 392
   External window 400
   Multiple round trips 397
Pop-up Windows (cont.)
  Single round trip  393
Product  94
Programmatic assignment of actions to UI element objects  320
Properties of a tree UI elements  318
Proxy object  410
Public Part  90, 433

Q
Query String  404

R
R/2  407
R/3  407
readSingleAddress()  447
Read-Write-Read problem  492
Recursive Nodes  500
  repeatedNode  202, 204
  Restrictions  204
Reenskaug, Trygve  38, 40, 56
Re-entrant navigation link  365
Re-import model  480
Relation roles  431
Remote callable ABAP function module  500
Remote callable function module  407, 429
Remote Function Call  39, 407
Request for navigation  361
Request Object  406
RFC  407
RFC interface  414
RFC modules  415
RFC programming in ABAP  414
RFM  407
Root Component  365, 403, 483
Root UI element  246
RootUIElementContainer  322
Row layout  247
Runtime Locale Determination  294

S
S2X  289, 291, 292
S2X Editor  291, 292, 293
S2X Header  289
S2X Resource Text  289
SAP Component Model  89
SAP Data Dictionary  417
SAP Gateway process  486
SAP GUI  470
SAP NetWeaver AS Java
  Installation  37
SAP NetWeaver Exchange Infrastructure  30
SAP NetWeaver Master Data Management  30
SAP Process Types
  Batch  487, 503
  Dialogue  486, 503
  Enqueue  487, 503
  Gateway  487, 503
  Message  487, 503
  Primary update  487
  Secondary update  487
  Spool  487, 503
  Update  487, 503
SAP SAP NetWeaver AS Java
  Deployment to  37
SAP Supported XLIFF  289
Secure Network Communication  473
Server Side Rendering  54, 168, 182, 312
setConnectionProvider()  481, 484
Sharing Web Dynpro functionality between multiple DCs  100
Simple Type  430, 456
Simple Value service  466
Smalltalk  40
SOAP  39
Software component  93, 500
Special Comment Markers
  //@@begin others  196
SSO2 cookie  467, 473
Standard hook methods  196, 454
Static nodes  297
Storing language specific text in .XLF files  290
Sun Microsystems  89
Suspend/Resume  368
  Configuring timeout values  374
  DefaultExpirationTime  374
  Interface View Controller  368, 369, 370
  Making an application suspendible  371
  Plug Processing  372
Suspend/Resume (cont.)
Requirements for an external website 373
Resume Plug 368, 370
Resume Plug Parameters 373
Resume Plug Processing 373
sap.expirationTime 374
sap.suspendExpirationTime 374
sap.suspensibility 371
Suspend plug 369
Suspend Plug Processing 372
Suspend Plugs 369
wdDoApplicationStateChange() 372, 373
System ID 502
System Landscape Directory 471

T
Table UI element 453
Change of lead selection 257
selectionChangeBehaviour 257
TABLES parameters 415
Technical system 471
Tight coupling 387
Transaction RZ10 487
Transaction SE11 417
Transaction SE16 417
Transaction SE37 417
Transaction SE80 417
Transaction SU01 473
Translating XLF files 291
Tree UI element 311, 312, 314
TreeItemType UI element 311, 312
TreeNodeType UI element 311, 312, 314
Types of Tree UI element 311

U
UI Design
Data binding 46
Declarative approach 45
UI Element
Avoiding confusion 244
Context acts as data generator 239
Controlling behaviour from the context 242
Data binding 239
UI Element (cont.)
Data binding and its effect on coding 243
Data consumer 46
Editing a view layout 238
Editing existing UI Element 239
help 265
hierarchy 237, 239
Layout Managers 246
properties 240, 501
Putting data on the screen 239
RootUIElementContainer 238
UI element properties act as data consumers 239
Unicode 505
URL parameter 405
URL properties 404
Used DC 433
Used Models 434
User Interface
Declaration of 34
Decoupled from business processing 34
User roll area 478, 489
Using model nodes at runtime 439
Using model objects at runtime 445

V
Value attribute 500
Value node 500
View 499, 502
View Assembly 262, 360, 378
View Controller 60, 62, 63, 64, 72, 73, 75, 76, 78, 80, 81, 82, 359
Action Handler 77
Actions 77
Data Binding 73, 76, 81
Design principles 63
Inbound plug 64, 74, 76, 86
Layout 63, 73, 81
Navigation Plugs 76
Outbound plug 64, 74, 76, 87
UI element 54, 62, 63, 71, 73, 74, 76, 77, 81, 82
UI Layout 76
ViewContainerUIElement 360, 365
Views 507
Data consumers 44
Index

Viewset  502, 508
Visual Administrator  375

W

wdContext  302
WDCopyService  233, 235
wdDoBeforeNavigation()  378
wdDoExit()  365, 385, 446
wdDoInit()  190, 196, 218, 221, 279, 298, 305, 364, 385, 441, 454
wdDoModifyView()  310, 323, 334, 335, 337, 460
wdDoPostProcessing()  446
WDModelScopeType  482
WDValueServices  351
Web Dynpro  30
Abstract modelling  35
Application  55, 57, 59, 63, 73, 74, 83, 84, 86, 87
Application declaration  116
Application Design  89
Application Parameters  86
Application Properties  85
Application time out  87
Causes of poor implementations  32
Code reuse  30
Code reuses at a business level  35
Component  34, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 65, 66, 67, 72, 78, 79, 83
Component instantiation  88
Consistent look and feel  45
Declarative design approach  35
Design criteria  35
Design philosophy  38
Differences with other toolsets  34
Eliminate the repetitive coding tasks  35
for ABAP  37
for Java  37
Fundamental principles  47
Generic services  35
Independent processing layer  34
Original design concept  35
Origins  29, 35
Reuse  59
SAP’s design requirements  38
Service Event Handlers  149
Standard for all future SAP software  35
Web Dynpro (cont.)
Suspend/Resume  87
Window  60, 64, 75
Web Dynpro Application
Shutdown  87
Start up  86
Web Dynpro Component
Controller Interaction  78
Navigation  64, 74, 75, 76, 83
View Assembly  73, 74, 75
Window  64, 73, 74, 75
Web Dynpro Component Architecture  499
Web Dynpro Components and polymorphism  387
Web Dynpro Content Administrator  472
Web Dynpro Controller  434, 499
Web Dynpro Design
Access to business functionality  110
Application Timeout  139
Calculated Attributes  137
Centralised Functionality  137
Child Component Lifecycle Management  117, 129
Clocks and business processes  111
Component Configuration  125
Component Hierarchy  115
Component Interface Definition  122, 123, 124, 125, 126
Context  135
Context Mapping  130, 136
Contract Staff  104
Creating a Story Board  111
Design Phase  113
Developer Experience  110
Faceless Component  119
Fragmentation  114
Functional Scope  106
Granularity  115
Implementation  129
Inter-component communication  129
Intermediate Components  118
Internal Data Storage  135
Language Requirements  126
Legal Requirements  126
Lifecycle hook methods  138
Localisation Requirements  126
Location of Business Processing  127
Loose Coupling  131
Index

Web Dynpro Design (cont.)
  Model 419
  Model Component 119, 120, 121, 122, 131, 132, 133, 134, 135, 136
  Monolithic Coding Structures 114
  Over-zealous sales pitch 105
  Phase Model 138, 140
  Planning 103
  Preparation 103
  Realistic Deadlines 107
  Referencing Mode 119, 120, 132, 134, 135
  Resume Request 139
  Root Component 116, 117, 118, 120, 123, 124, 125, 129, 133, 134, 135
  Round trip hook methods 139
  Scope Creep 105
  Standard Hook Methods 138, 140
  Stateful or stateless connections 127
  Suspend Request 138
  Text Direction 126
  Text in images 127
  The value of not making a mistake 108
  Tight Coupling 122, 130
  Training 104
  Utility Components 114, 118, 119
  wdDoApplicationStateChange() 138
  wdDoBeforeNavigation() 139
  wdDoExit() 138
  wdDoInit() 138

Web Dynpro Design (cont.)
  wdDoModifyView() 139
  wdDoPostProcessing() 140

Web Dynpro Framework 127, 129, 138, 139, 140


Lifecycle hook methods 70
  Round-trip hook method 70
  Standard Hook Methods 69, 85

Web Dynpro i18n 287
  Web Dynpro i18n concept 288
  Web Dynpro Models DCs 433
  Web Dynpro naming placeholders 499
  Web Dynpro Perspective 419
  Web Dynpro Project 98

Web Service 39
Window 359, 499, 508
  Working with Tree UI elements 311

X

X.509 certificate 473
Xerox PARC 40
XLF 290
XLIFF 289