James Wood and Shaan Parvaze

Web Dynpro ABAP®
The Comprehensive Guide
# Contents at a Glance

1. Getting Started ................................................................. 29
2. The Anatomy of WDA Applications ................................. 61
3. Developing Your First WDA Application ......................... 89
4. Controller Development .................................................. 127
5. Working with Contexts ..................................................... 179
6. Windows and Views ......................................................... 237
7. Basic UI Elements ........................................................... 307
8. Advanced UI Elements ..................................................... 347
9. Component-Based Development Concepts ....................... 421
10. User Interaction ............................................................. 467
11. Dynamic Programming .................................................. 519
12. Configuration and Adaptation ......................................... 565
13. Modifications and Enhancements .................................... 593
14. Working with the Floorplan Manager .............................. 607
15. WDA Integration ............................................................ 633
17. Performance Tuning ....................................................... 669

## Development Workbooks

- **W1** Displaying Basic Contact Information for a User .......... 703
- **W2** Creating a Fully Functional User Maintenance Application .... 709
- **W3** Showing User Role Assignments ................................ 719
- **W4** Developing a Custom User Report .............................. 725
- **W5** Implementing Special Features .................................. 735
- **W6** Using Component-Based Development Techniques .......... 743
- **W7** Configuring an FPM Application ................................. 749
- **W8** Integrating WDA with Flash ....................................... 757

## A
- Debugging WDA Applications .......................................... 765

## B
- The Authors ................................................................. 773
Contents

Introduction ..................................................................................................... 21

1 Getting Started .................................................................................. 29
   1.1 Foundations for Web Dynpro ......................................................... 30
      1.1.1 HTTP and the World Wide Web ......................................... 30
      1.1.2 Evolution of Web Programming Models ...................... 33
   1.2 A Model-Based Approach to UI Development ...................... 41
      1.2.1 Understanding the Web Dynpro Framework .................. 42
      1.2.2 The Benefits of a Model-Driven Architecture .............. 44
   1.3 Technical Integration Concepts ................................................. 47
      1.3.1 Legacy Web and UI Technologies ....................................... 47
      1.3.2 Internet Communication Manager ..................................... 50
      1.3.3 Internet Communication Framework .......................... 52
      1.3.4 The Web Dynpro Runtime Environment ..................... 57
      1.3.5 System Requirements ......................................................... 59
   1.4 Summary ...................................................................................... 59

2 The Anatomy of WDA Applications ............................................ 61
   2.1 Component-Based Software Development Concepts .............. 61
      2.1.1 Componentization Overview .............................................. 62
      2.1.2 Characteristics of Software Components ..................... 63
      2.1.3 Understanding the Positioning of Web Dynpro Components .............................................. 64
   2.2 Web Dynpro Components .............................................................. 65
      2.2.1 Architectural Overview ...................................................... 65
      2.2.2 Views ................................................................................. 67
      2.2.3 Windows ........................................................................... 69
      2.2.4 Controllers ......................................................................... 73
      2.2.5 Context and Binding Concepts ..................................... 77
   2.3 Web Dynpro Applications ............................................................... 85
   2.4 Summary ...................................................................................... 88
4.5 Web Dynpro Programming Interface Overview .............................................. 165
  4.5.1 The Web Dynpro Runtime API ........................................................... 165
  4.5.2 Service Classes and Interfaces ............................................................ 167
4.6 Assistance Classes .................................................................................... 168
  4.6.1 Creating an Assistance Class ............................................................... 168
  4.6.2 Maintaining Assistance Classes .......................................................... 168
  4.6.3 What Should Assistance Classes Be Used For? ................................... 170
4.7 Tool Support for Controller Development ................................................. 171
  4.7.1 Web Dynpro Code Wizard .................................................................. 172
  4.7.2 Screen Design Time Conversion .......................................................... 173
  4.7.3 Service Calls ....................................................................................... 176
  4.7.4 Refactoring Assistant ........................................................................ 177
4.8 Summary ................................................................................................... 178

5 Working with Contexts ................................................................................ 179
  5.1 Contexts: Revisited ................................................................................ 179
  5.1.1 What Is a Context? ............................................................................. 180
  5.1.2 Context Data Flow Concepts ............................................................... 180
  5.1.3 The Building Blocks of a Context ....................................................... 182
  5.1.4 Contexts at Runtime ......................................................................... 183
5.2 Context Nodes and Context Attributes: Up Close ..................................... 184
  5.2.1 Properties of Context Nodes ............................................................... 184
  5.2.2 Properties of Context Attributes ........................................................ 187
  5.2.3 Understanding the Lead Selection Concept ....................................... 188
  5.2.4 Understanding the Singleton Property ............................................. 189
5.3 Defining a Context .................................................................................... 191
  5.3.1 Defining Context Nodes ..................................................................... 191
  5.3.2 Defining Context Attributes ............................................................... 195
  5.3.3 Recursion Nodes ................................................................................. 197
5.4 Context Programming ............................................................................... 202
  5.4.1 Getting Started ................................................................................... 202
  5.4.2 Maneuvering within a Context ............................................................. 205
  5.4.3 Filling a Context ................................................................................ 212
  5.4.4 Working with Context Attributes ....................................................... 217
  5.4.5 Performing Housekeeping Duties within a Context ............................ 223
  5.4.6 Working with the Context Change Log ............................................. 225
  5.4.7 Tool Support for Context Programming ........................................... 227
## Contents

### 5.5 Supply Functions ................................................................. 228
- 5.5.1 Creating a Supply Function ........................................... 228
- 5.5.2 Implementing Supply Functions ................................. 229
- 5.5.3 Understanding How Supply Functions Are Invoked ....... 230
- 5.5.4 Supply Functions and Singleton Nodes ...................... 231

### 5.6 Context Mapping and Data Binding: Up Close .................. 232
- 5.6.1 How Context Mapping Works ....................................... 232
- 5.6.2 How Data Binding Works ............................................. 233

### 5.7 Summary ............................................................................. 235

### 6 Windows and Views ............................................................... 237

#### 6.1 Windows ........................................................--------------- 238
- 6.1.1 Windows as a View Container ...................................... 238
- 6.1.2 Interface Views ............................................................ 247
- 6.1.3 Dialog Boxes and Popup Windows ................................. 252

#### 6.2 Views ............................................................................... 259
- 6.2.1 Views: Revisited .......................................................... 259
- 6.2.2 Working with Layout Elements .................................... 262
- 6.2.3 Working with Container Layouts ................................. 272

#### 6.3 Navigation Concepts .......................................................... 281
- 6.3.1 Navigation between Views .......................................... 281
- 6.3.2 Working with Window Plugs ....................................... 289
- 6.3.3 Case Study: Dynamically Selecting the Initial View for a Window ......................................................... 292
- 6.3.4 Case Study: Closing WDA Applications ...................... 297
- 6.3.5 Case Study: Suspend and Resume Calls ...................... 299

#### 6.4 Summary ............................................................................. 306

### 7 Basic UI Elements ................................................................. 307

#### 7.1 Introduction to UI Elements .............................................. 307
- 7.1.1 Basic Concepts ............................................................. 308
- 7.1.2 Runtime Classes ........................................................... 310
- 7.1.3 UI Element Categories ............................................... 312

#### 7.2 UI Elements from the Text Category ............................... 313
- 7.2.1 UI Element Overview .................................................. 313
- 7.2.2 Building Input Forms Using Label and InputField ....... 315
Contents

7.2.3 Displaying Plain Text Using TextView .............................................. 323
7.2.4 Editing Plain Text with TextEdit ................................................... 323
7.2.5 Displaying Formatted Text using FormattedTextView ................. 325
7.2.6 Editing Formatted Text with FormattedTextEdit ......................... 328
7.3 UI Elements from the Action Category .............................................. 329
  7.3.1 UI Element Overview .................................................................. 330
  7.3.2 Basic Concepts ........................................................................... 331
  7.3.3 Working with Menus .................................................................. 332
  7.3.4 Grouping Action Elements with the ToolBar Element ................. 335
7.4 UI Elements from the Selection Category ......................................... 336
  7.4.1 UI Element Overview .................................................................. 337
  7.4.2 Working with Dropdown Lists .................................................... 340
  7.4.3 Working with Radio Buttons ....................................................... 345
7.5 Summary .......................................................................................... 346

8 Advanced UI Elements ................................................................. 347

8.1 UI Elements from the Graphic Category ......................................... 347
  8.1.1 UI Element Overview .................................................................. 348
  8.1.2 Displaying Images with the Image Element ............................... 350
  8.1.3 Working with the ThresholdSlider Element ............................... 355
8.2 UI Elements from the Integration Category ..................................... 358
  8.2.1 UI Element Overview .................................................................. 358
  8.2.2 Uploading and Downloading Files .............................................. 360
  8.2.3 Integrating PDF Forms with the InteractiveForm Element .......... 364
8.3 UI Elements from the Complex Category ........................................ 366
  8.3.1 UI Element Overview .................................................................. 366
  8.3.2 Displaying Tabular Data with the Table Element ....................... 374
  8.3.3 Visualizing Hierarchical Data with the Tree Element ................. 392
  8.3.4 Working with the RoadMap Element .......................................... 404
8.4 Drag and Drop ............................................................................... 409
  8.4.1 Conceptual Overview .................................................................. 409
  8.4.2 The Mechanics of Drag and Drop .............................................. 412
  8.4.3 Case Study: Defining a Drag-and-Drop Scenario ......................... 415
8.5 Summary .......................................................................................... 420
Contents

9 Component-Based Development Concepts ................................................. 421
  9.1 Web Dynpro Component Interfaces: Revisited ................................ 422
    9.1.1 Component Interface Overview ............................................. 423
    9.1.2 Defining the Interface Controller .......................................... 424
    9.1.3 Configuring Interface Views ................................................... 428
    9.1.4 Defining Standalone Web Dynpro Component Interfaces ... 430
  9.2 Component Usage Concepts ........................................................... 434
    9.2.1 Defining Component Usages .............................................. 434
    9.2.2 Instantiating Used Components ......................................... 436
    9.2.3 Accessing Services of the Interface Controller ..................... 439
    9.2.4 Cross-Component Context Mapping .................................. 441
    9.2.5 Defining Cross-Component Navigation Scenarios ............... 446
  9.3 Case Study: Working with Faceless Components ......................... 449
  9.4 Case Study: Working with the SAP List Viewer ......................... 452
    9.4.1 Integrating ALV Content into the View Layout ................... 452
    9.4.2 Accessing the ALV Configuration Model ............................ 455
    9.4.3 Event Handling Concepts ................................................... 458
  9.5 Guidelines for Applying Multi-Component Designs ......................... 462
  9.6 Summary ........................................................................................ 465

10 User Interaction .................................................................................. 467
  10.1 Defining Input Helps ...................................................................... 467
    10.1.1 ABAP Dictionary Search Helps ........................................... 468
    10.1.2 OVS Value Helps .............................................................. 477
    10.1.3 Freely Programmed Input Helps ......................................... 482
    10.1.4 Select Options ................................................................. 487
    10.1.5 Suggested Values for Input Fields ....................................... 497
  10.2 Providing Help Texts ................................................................. 498
    10.2.1 Working with the Explanation UI Element .......................... 498
    10.2.2 Explanation Property for UI Elements .................................. 500
    10.2.3 Integrating Classic F1 Help from the ABAP Dictionary ........ 501
  10.3 Confirmation Dialog Boxes ............................................................ 503
  10.4 Displaying Messages ....................................................................... 505
    10.4.1 Creating Messages with the Message Manager ..................... 506
    10.4.2 Automatic Message Generation .......................................... 513
    10.4.3 Customizing the Message Area ........................................... 515
  10.5 Summary ........................................................................................ 518
11 Dynamic Programming ................................................................. 519
  11.1 Manipulating the Context ......................................................... 519
    11.1.1 Working with the IF_WD_CONTEXT_NODE_INFO Interface .......... 520
    11.1.2 Adding Context Nodes to the Node Hierarchy ......................... 522
    11.1.3 Adding Context Attributes to a Context Node ......................... 527
    11.1.4 Dynamic Context Programming Dos and Don'ts ......................... 528
  11.2 Modifying the View Layout at Runtime ...................................... 528
    11.2.1 Getting Started with the WDDOMODIFYVIEW( ) Hook Method .......... 529
    11.2.2 Navigating the Class Hierarchy for UI Elements ....................... 531
    11.2.3 Adding UI Elements to the UI Element Hierarchy ..................... 533
    11.2.4 Manipulating UI Elements Programmatically ............................. 536
    11.2.5 Declarative Alternatives .................................................. 538
  11.3 Working with Dynamic Component Usages .................................... 539
    11.3.1 Dynamically Creating Component Usages ................................. 539
    11.3.2 Dynamically Embedding Interface Views .................................... 542
    11.3.3 Calling Methods of the Interface Controller ............................ 546
    11.3.4 Registering Event Handler Methods ...................................... 547
    11.3.5 Programming against Component Interfaces .............................. 548
  11.4 Case Study: Creating a Data Browser Tool ................................... 552
    11.4.1 Requirements Overview .................................................... 552
    11.4.2 Generating a Selection Screen Using Selection Options .............. 554
    11.4.3 Dynamic Context Manipulation ............................................ 557
    11.4.4 Generating the Results Screen ............................................ 561
  11.5 Summary .................................................................................. 564

12 Configuration and Adaptation ...................................................... 565
  12.1 The Adaptation Concept .......................................................... 566
    12.1.1 Adaptation Layers .............................................................. 566
    12.1.2 Adaptation Multiplicity and Processing ..................................... 567
  12.2 Component Configuration with the Configuration Editor .................. 568
    12.2.1 Accessing the Configuration Editor ....................................... 568
    12.2.2 Accessing the Customization Editor ...................................... 569
    12.2.3 Creating a Component Configuration ..................................... 570
    12.2.4 Configuring Component-Defined Properties ............................. 571
    12.2.5 Configuring Built-In Properties .......................................... 572
Contents

12.3 Application Configuration ............................................................... 572
  12.3.1 Accessing the Application Configurator .............................. 573
  12.3.2 Creating an Application Configuration ............................ 574
  12.3.3 Assigning Component Configurations ................................. 574
  12.3.4 Specifying Application Parameters .................................... 574
  12.3.5 Assigning Application Configurations ................................. 575

12.4 Personalization and Customization ................................................. 575
  12.4.1 Personalizing Applications .................................................. 576
  12.4.2 Embedding Built-In Configuration Options in a Custom
       Personalization Dialog ....................................................... 576
  12.4.3 Customizing Applications .................................................. 577
  12.4.4 Deactivating Customization and Personalization ................. 578
  12.4.5 Managing Personalization and Customization Data ............ 579

12.5 Component-Defined Adaptation ..................................................... 579
  12.5.1 Configuration Controllers ................................................... 580
  12.5.2 Component-Defined Adaptation Data ................................ 581
  12.5.3 Mandating and Deactivating Component-Defined
       Configuration Data ............................................................. 582

12.6 Parameterization ............................................................................ 583
  12.6.1 Setting Parameters ............................................................. 583
  12.6.2 Processing Parameters ........................................................ 584
  12.6.3 Parameter Security ............................................................. 585

12.7 Accessibility .................................................................................... 585

12.8 Style Sheets .................................................................................... 586
  12.8.1 WDA Integration with Themes ........................................... 587
  12.8.2 Integrating a Custom Theme into WDA .............................. 588
  12.8.3 Setting a Global Theme for WDA ....................................... 588

12.9 Internationalization and Translation ................................................ 588
  12.9.1 Static Texts ........................................................................ 589
  12.9.2 Dynamic Texts ................................................................... 590
  12.9.3 Long Texts ......................................................................... 591

12.10 Summary ........................................................................................ 591

13 Modifications and Enhancements ....................................................... 593
  13.1 WDA Integration with the Enhancement Framework ................. 593
  13.1.1 Enhancement Spots ........................................................... 594
  13.1.2 Enhancement Implementations ............................................ 595
13.1.3 Creating an Enhancement Implementation ......................... 595
13.1.4 Switch Framework Integration ............................................. 597
13.2 Enhancing Web Dynpro Components ............................................. 597
13.2.1 Creating New Views ........................................................... 597
13.2.2 Enhancing Existing Views ................................................... 598
13.2.3 Enhancing Controllers ........................................................ 600
13.2.4 Adding New Windows ....................................................... 602
13.2.5 Enhancing Existing Windows .............................................. 602
13.2.6 Enhancing Components ..................................................... 603
13.3 Explicit Enhancements in Web Dynpro ........................................... 603
13.3.1 Working with BAdIs ........................................................... 603
13.3.2 Working with Enhancement Options .................................. 604
13.4 Adjustments after an Upgrade ....................................................... 605
13.4.1 Types of Web Dynpro Conflicts .......................................... 605
13.4.2 Making Adjustments to Your Enhancements ...................... 605
13.5 Summary ........................................................................................ 606

14 Working with the Floorplan Manager ............................................. 607

14.1 Floorplan Architecture .............................................................. 608
14.1.1 Design of a Floorplan Manager Application ........................ 608
14.1.2 Implementation Architecture of a Floorplan Manager Application ........................................... 609
14.1.3 Floorplan Components ....................................................... 610
14.1.4 User Interface Building Blocks ............................................ 613
14.1.5 Floorplan Configuration Objects ......................................... 614
14.2 Getting Started with Floorplans ...................................................... 615
14.2.1 Implementing a UIBB ......................................................... 615
14.2.2 Creating a Floorplan Application ........................................ 615
14.2.3 Configuring a Floorplan Application ................................... 617
14.3 Working with the Configuration Editor ........................................... 618
14.3.1 Accessing the Configuration Editor ..................................... 619
14.3.2 Features of the Configuration Editor ................................ 620
14.3.3 Assigning a UIBB to the Page ............................................ 621
14.4 Working with GUIBBs .............................................................. 622
14.5 Configuring Wires ................................................................. 623
14.6 Working with Dialog Boxes ....................................................... 624
14.6.1 Creating an FPM Dialog Box .............................................. 624
Contents

14.6.2 Assigning a Dialog Box to a Toolbar Button ............................................... 625
14.6.3 Calling Dialog Boxes Programmatically ..................................................... 626
14.7 Web Dynpro Page Builder ....................................................................... 626
  14.7.1 Creating Pages ........................................................................ 627
  14.7.2 Accessing the Page Builder ......................................................... 628
  14.7.3 Creating Web Dynpro CHIPs ....................................................... 628
  14.7.4 Adding a Side Panel to WDA Applications .................................... 630
14.8 Summary .................................................................................................. 632

15 WDA Integration ......................................................................................... 633
  15.1 RIA Integration with Web Dynpro Islands ............................................. 634
    15.1.1 View Concepts in the Islands Framework ...................................... 634
    15.1.2 Data Transfer Concepts in the Islands Framework ...................... 635
    15.1.3 Eventing Concepts in the Islands Framework ............................. 637
  15.2 Adobe Flash Islands Integration ......................................................... 638
    15.2.1 Preparing Adobe Flash Builder to Work with Web Dynpro .......... 638
    15.2.2 Integrating Flash Islands in Web Dynpro .................................... 639
    15.2.3 Data Transfer ......................................................................... 640
    15.2.4 Event Integration .................................................................... 641
    15.2.5 Theme Integration .................................................................. 641
    15.2.6 Drag and Drop Integration ...................................................... 642
  15.3 Microsoft Silverlight Islands Integration ............................................. 642
    15.3.1 Preparing Microsoft Visual Studio to Work with Web Dynpro ........ 642
    15.3.2 Integrating Silverlight Islands in Web Dynpro ............................ 643
    15.3.3 Data Transfer ........................................................................ 643
  15.4 SAP Interactive Forms by Adobe Integration ..................................... 644
    15.4.1 Creating a New Form from Web Dynpro .................................. 645
    15.4.2 Integrating Existing Forms ...................................................... 646
  15.5 SAP NetWeaver Portal Integration ...................................................... 647
    15.5.1 Preparing SAP NetWeaver Portal to Work with WDA ................ 647
    15.5.2 Creating Web Dynpro ABAP iViews ........................................ 649
    15.5.3 Portal Event Integration ........................................................... 650
    15.5.4 Portal Navigation .................................................................... 650
  15.6 Consuming Web Services in WDA ....................................................... 651
  15.7 Summary .................................................................................................. 651


## 16 Security Concepts ................................................................. 653

16.1 System Logon ........................................................................ 654
  16.1.1 Customizing the Logon Screen for Your Applications ...... 654
  16.1.2 Logon Using SSO Tickets ................................................. 656
  16.1.3 Establishing a Single Sign-On Between Systems Using SSO Tickets .................................................. 657
  16.1.4 Logon Using an Identity Provider (IdP) ......................... 659

16.2 System Logoff and Error Pages ............................................... 660
  16.2.1 Application Logoff Page ............................................... 660
  16.2.2 Application Error Page ................................................. 661
  16.2.3 External Aliases ......................................................... 662

16.3 Authorizations ....................................................................... 663

16.4 Application Data Security ....................................................... 665

16.5 Security of Context Data in a View ......................................... 666

16.6 File Upload Security .............................................................. 666

16.7 Whitelist Infrastructure .......................................................... 667

16.8 Summary ............................................................................. 668

## 17 Performance Tuning ............................................................... 669

17.1 Formulating an Approach ....................................................... 670

17.2 Best Practices and Design Tips ............................................... 672
  17.2.1 Separation of Concerns .................................................... 672
  17.2.2 Context Management ..................................................... 673
  17.2.3 Managing Resources ...................................................... 674
  17.2.4 Frontend Delegation ....................................................... 679
  17.2.5 Compressing HTTP Transmissions ............................... 683

17.3 Performance Analysis Tools .................................................... 684
  17.3.1 Web Dynpro–Specific Monitoring Tools ....................... 685
  17.3.2 ABAP Monitoring Tools ............................................... 691
  17.3.3 HTTP Monitoring Tools ............................................... 695

17.4 Performance Tuning Checklist ............................................... 700

17.5 Summary ............................................................................. 700
## Contents

### Development Workbooks

<table>
<thead>
<tr>
<th>Workbook</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1 Displaying Basic Contact Information for a User</td>
<td>703</td>
</tr>
<tr>
<td>W1.1 Requirements Overview</td>
<td>703</td>
</tr>
<tr>
<td>W1.2 Design Approach</td>
<td>704</td>
</tr>
<tr>
<td>W1.2.1 Modeling the Application Data</td>
<td>704</td>
</tr>
<tr>
<td>W1.2.2 Defining the User Interface</td>
<td>705</td>
</tr>
<tr>
<td>W1.2.3 Event Handling</td>
<td>706</td>
</tr>
<tr>
<td>W1.3 Solution</td>
<td>707</td>
</tr>
<tr>
<td>W2 Creating a Fully Functional User Maintenance Application</td>
<td>709</td>
</tr>
<tr>
<td>W2.1 Requirements Overview</td>
<td>709</td>
</tr>
<tr>
<td>W2.2 Design Approach</td>
<td>713</td>
</tr>
<tr>
<td>W2.2.1 Building an Application Model</td>
<td>713</td>
</tr>
<tr>
<td>W2.2.2 Defining the Context</td>
<td>714</td>
</tr>
<tr>
<td>W2.2.3 Designing the User Interface</td>
<td>715</td>
</tr>
<tr>
<td>W2.2.4 Application Mode Handling</td>
<td>716</td>
</tr>
<tr>
<td>W2.3 Solution</td>
<td>717</td>
</tr>
<tr>
<td>W3 Showing User Role Assignments</td>
<td>719</td>
</tr>
<tr>
<td>W3.1 Requirements Overview</td>
<td>719</td>
</tr>
<tr>
<td>W3.2 Design Approach</td>
<td>719</td>
</tr>
<tr>
<td>W3.2.1 Defining the Context</td>
<td>720</td>
</tr>
<tr>
<td>W3.2.2 Configuring the Table Element</td>
<td>722</td>
</tr>
<tr>
<td>W3.2.3 Turning on Sorting and Filtering</td>
<td>722</td>
</tr>
<tr>
<td>W3.2.4 Defining the TablePopin</td>
<td>723</td>
</tr>
<tr>
<td>W3.3 Solution</td>
<td>723</td>
</tr>
<tr>
<td>W4 Developing a Custom User Report</td>
<td>725</td>
</tr>
<tr>
<td>W4.1 Requirements Overview</td>
<td>725</td>
</tr>
<tr>
<td>W4.2 Design Approach</td>
<td>727</td>
</tr>
<tr>
<td>W4.2.1 Defining the Component Architecture</td>
<td>727</td>
</tr>
<tr>
<td>W4.2.2 Constructing a Reusable Query Model</td>
<td>728</td>
</tr>
<tr>
<td>W4.2.3 Designing the User Interface</td>
<td>731</td>
</tr>
<tr>
<td>W4.2.4 Coordinating the Event Flow</td>
<td>731</td>
</tr>
<tr>
<td>W4.3 Solution</td>
<td>733</td>
</tr>
</tbody>
</table>
Appendices ....................................................................................... 763

A Debugging WDA Applications ....................................................... 765
A.1 Activating the ABAP Debugger Tool ....................................... 765
A.2 Working with the Web Dynpro Debugger Tool ....................... 770
B The Authors .................................................................................. 773

Index ........................................................................................................ 775
If we trace the flow of data through WDA components, we can see that data is extracted from the model layer, filtered in the controller layer, and ultimately displayed in the view layer. At the intersection between these layers lie controller contexts, which broker the transfer of data in both directions. In this chapter, we’ll learn how to work with these contexts in order to facilitate data exchange in WDA components.

5 Working with Contexts

In the early days of web programming, developers spent quite a bit of time coming up with ways to shuffle data back and forth between form fields in the UI and the backend data model. Fortunately, with the advent of modern web development frameworks like Web Dynpro, developers can now rely on built-in data binding facilities to automate this synchronization of data. Here, data binding greatly simplifies the application development process since the framework assumes the responsibility of making sure that data is where it’s supposed to be when we need it.

Within the Web Dynpro framework, the glue that binds UI elements with the backend data model is the context. Therefore, in order to master the Web Dynpro programming model, we must learn how to work with controller contexts. In this chapter, we’ll spend quite a bit of time looking at contexts from three different angles: conceptual overview, design time, and runtime. Collectively, these various perspectives will provide us with the insight we need to model and exchange various kinds of data using contexts.

5.1 Contexts: Revisited

Developers who are new to Web Dynpro often have a difficult time wrapping their heads around the notion of a context. This is perhaps due to the fact that the term context takes on different meanings depending on when and where it’s
being used. For instance, at design time, a context is a hierarchical data structure that is declaratively defined using graphical editor tools. However, at runtime, a context more closely resembles a complex structure or object collection that is manipulated programmatically using object-oriented APIs.

Recognizing that all this may still seem a bit abstract, we’ll spend some time in this section revisiting some of the context-related concepts explored in Chapter 2. Along the way, we’ll try to fill in some gaps now that you’re more familiar with surrounding elements of the Web Dynpro framework. Alas, with a streamlined framework like Web Dynpro, it’s difficult to comprehend one concept without having at least a cursory understanding of several others. Fortunately, by the close of this chapter, this circuitous journey through the core of the Web Dynpro framework will be almost complete.

5.1.1 What Is a Context?

The textbook definition of a context in Web Dynpro literature typically reads something like this: “A context is a hierarchical data storage area contained within a controller.” While this definition accurately describes what a context is, it fails to illustrate what contexts are used for. So, to take things in a different direction, we’ll define contexts as the interface between the UI and the backend data model. As such, contexts provide a convenient mechanism for passing data back and forth between the view and model layers.

5.1.2 Context Data Flow Concepts

Before we delve into the mechanics of context design and development, it’s useful to take a moment to reflect once again on how contexts are used to facilitate data transfer between the different layers of a WDA component. Therefore, in this section, let’s trace through this data flow by looking at an example. Here, we’ll follow the flow of data from the point it’s extracted from the model to the point it’s displayed in the UI. Then, we’ll see how changes to data in the UI are propagated back along the same path to the model layer.

As a basis for our analysis, let’s consider the data flow diagram in Figure 5.1. As you can see, this flow is initiated in a global controller method via a series of API calls that copy data from the model layer into the controller context. This data is then propagated down to the view layer via a technique called context mapping.
Finally, the data from the mapped view context is displayed within the UI via *data binding* relationships established between UI elements in the view layout and selected context nodes and attributes. As users modify data within the UI, the Web Dynpro framework takes care of transmitting those changes back upstream to the global controller. Then, the global controller can take that data and use it to synchronize the model via API calls as necessary.

![Diagram of data flow](image)

**Figure 5.1**  Context Data Flow Diagram

Throughout the remainder of this chapter, we'll see how each leg of this data transfer sequence is implemented.
5.1.3 The Building Blocks of a Context

As we stated earlier, contexts are hierarchical data structures made up of context nodes and context attributes. Looking at Figure 5.2, we can see that context hierarchies are organized as follows:

- At the base of the hierarchy, there’s a specialized context node called the context root. This node is always the first in any context and cannot be changed or removed.

- Underneath the context root node, we can model various types of complex data objects (e.g., sales orders, materials, and so on) using a series of nested child context nodes. Each context node has a defined cardinality that determines how elements the node can maintain at runtime.

- Child context nodes are defined in terms of context attributes and additional child context nodes—more on the latter in a moment. As you can see in Figure 5.2, context attributes are analogous to leaves in a tree data structure. In other words, they’re used to represent atomic values such as strings, dates, or numbers. So, when modeling a sales order, for example, context attributes would be used to represent the order number, order creation date, and so on.

- Since each context node can define one or more child nodes, a context hierarchy can be nested arbitrarily deep. So, for example, with the SalesOrders context node depicted in Figure 5.2, we might have a three-tier hierarchy consisting of SalesOrders, OrderItems, and ScheduleLines nodes.
5.1.4 Contexts at Runtime

Contexts take on a different dimension at runtime, expanding as necessary to provide adequate storage for data displayed within the UI. In this regard, context nodes have certain characteristics in common with internal tables:

- Just as internal tables have a \textit{line type} (which is defined in terms of a structure type), the entities of a context node (i.e., context attributes and child nodes) are aggregated into a unit called a context element. At runtime, a context element is analogous to a record in an internal table.
- Depending on its defined cardinality, a context node may contain zero or more context elements at runtime.
- As is the case with internal tables, it’s possible to access individual elements within the collection by index. Furthermore, it’s also possible to append, remove, and modify context nodes individually or en masse.

Though the internal table metaphor is useful for illustrating the relationship between context nodes and context elements, perhaps a more accurate analogy is to think of context nodes as being like \textit{classes} from the object-oriented programming (OOP) paradigm. In OOP, classes are used to model some kind of real-world phenomenon at design time using a series of \textit{attributes}. At runtime, class definitions are used as a template for creating an \textit{object instance}. Naturally, these object instances are created in the image of their template class. Therefore, they inherit all the attributes from their defining class. Similarly, you can think of context elements as being \textit{instances} of their defining context node.

To put all these relationships in perspective, let’s take a look at what the sample context hierarchy from Figure 5.2 might look like at runtime. Figure 5.3 illustrates this relationship. Here, we can see how the SalesOrders context node contains multiple context elements: one per sales order. Each SalesOrders context element gets its own copy of the OrderNo and CreationDate context attributes, as well as its own copy of the OrderItems child node. The OrderItems node in turn contains its own context elements: one for each of the line items within the corresponding parent sales order. We could continue to traverse our way through the tree to look at item schedule lines and so forth, but you get the idea.

Looking at the context nodes depicted in Figure 5.3, you can see that context nodes effectively take on the role of \textit{container} from a runtime perspective. In other words, they are placeholders from which we can traverse the context...
hierarchy and access individual context elements and attributes. This will become clearer as we learn how to work programmatically with these constructs in Section 5.4.

5.2 Context Nodes and Context Attributes: Up Close

Now that you have a feel for the positioning of context nodes and context attributes within the overall context hierarchy, let’s narrow our focus a little bit and take a look at some of the properties of these elements. As we’ll soon see, these properties go a long way toward defining the structure and behavior of a context at runtime.

5.2.1 Properties of Context Nodes

Whenever we define a context node, there are quite a few properties that we must configure. We’ll see how this configuration is carried out in the Context Editor tool in Section 5.3.1. However, in the meantime, we need to spend some time reflecting on the semantic meaning of these properties. Table 5.1 describes each of these properties in detail.

Figure 5.3 A Sample Context at Runtime
### Table 5.1 Properties of Context Nodes

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Node Name</strong></td>
<td>This property defines the context node’s <em>name</em>. The name property is used extensively within the context API to access nodes and traverse the context hierarchy.</td>
</tr>
<tr>
<td><strong>Interface Node</strong></td>
<td>This Boolean property determines whether or not the context node is part of the <em>component interface</em>. As such, it can only be selected for context nodes created within the component controller.</td>
</tr>
<tr>
<td><strong>Input Element (Ext.)</strong></td>
<td>If the <em>Interface Node</em> property is selected for a context node, this Boolean property can be used to determine whether or not the node can be used in external mapping scenarios. We’ll explore these scenarios when we look at component-based development concepts in Chapter 9.</td>
</tr>
<tr>
<td><strong>Dictionary Structure</strong></td>
<td>Frequently, we’ll want to define context nodes in terms of pre-existing ABAP Dictionary structures. In these situations, we can use the <em>Dictionary Structure</em> property to make such an assignment for a context node.</td>
</tr>
<tr>
<td><strong>Cardinality</strong></td>
<td>This property is used to define the <em>cardinality</em> of a context node. In other words, this property determines how many instances of a context node (i.e., context elements) may exist at runtime. The possible values that we can choose from here are as follows:</td>
</tr>
<tr>
<td></td>
<td>- 0..1</td>
</tr>
<tr>
<td></td>
<td>- 1..1</td>
</tr>
<tr>
<td></td>
<td>- 0..n</td>
</tr>
<tr>
<td></td>
<td>- 1..n</td>
</tr>
<tr>
<td></td>
<td>In effect, the cardinality property specifies a range for context elements within a node collection, with the minimum number of elements in the collection on the left-hand side of the ellipsis and the maximum number of elements on the right-hand side. So, for example, if we were to choose a cardinality of 0..n, there could be an unlimited number of context elements within the context node at runtime, but the collection could also be empty. Conversely, if we were to choose a cardinality of 1..n, there must be at least one context element within the collection at all times, but there could be many more.</td>
</tr>
</tbody>
</table>
Ultimately, the elements within a context node collection are intended to be displayed within UI elements at the view layer. Here, users may pick and choose particular elements within the collection for further processing.

To control the number of elements that may be selected at any given time, we can utilize the Selection property. This property has the same domain of values the Cardinality property has. So, if we want to declare that at most one element can be selected at any given time, we might specify a value of 0..1. Similarly, if we want to allow for multiple selection, we might specify a value of 0..n.

One thing to keep in mind as you configure the Selection property is that the value you select must coincide with the Cardinality property. In other words, if you’ve defined the cardinality of a context node as 0..1, it doesn’t make sense to configure the Selection property with a value of 1..n. Of course, if you accidentally make a mistake, the syntax check will let you know.

Provided that a node collection is not empty at runtime, the system will designate a particular element within the collection as the lead selection. As we’ll learn in Section 5.4, this designation simplifies the way context elements are accessed at runtime.

Though it’s possible to specify the lead selection directly via the context APIs, it’s oftentimes much more convenient and effective to let the framework take care of it. You can do this by selecting the Init. Lead Selection property (which is selected by default whenever we create a new context node). We’ll consider the lead selection concept in further detail in Section 5.2.3.

This property is used to determine whether or not a context node is defined as a singleton. We’ll describe this property in further detail in Section 5.2.4.

With this property, we can select a supply function method which will supply a context node with data on demand at runtime. We’ll see how supply functions work in Section 5.5.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>Ultimately, the elements within a context node collection are intended to be displayed within UI elements at the view layer. Here, users may pick and choose particular elements within the collection for further processing. To control the number of elements that may be selected at any given time, we can utilize the Selection property. This property has the same domain of values the Cardinality property has. So, if we want to declare that at most one element can be selected at any given time, we might specify a value of 0..1. Similarly, if we want to allow for multiple selection, we might specify a value of 0..n. One thing to keep in mind as you configure the Selection property is that the value you select must coincide with the Cardinality property. In other words, if you’ve defined the cardinality of a context node as 0..1, it doesn’t make sense to configure the Selection property with a value of 1..n. Of course, if you accidentally make a mistake, the syntax check will let you know.</td>
</tr>
<tr>
<td>Init. Lead Selection</td>
<td>Provided that a node collection is not empty at runtime, the system will designate a particular element within the collection as the lead selection. As we’ll learn in Section 5.4, this designation simplifies the way context elements are accessed at runtime. Though it’s possible to specify the lead selection directly via the context APIs, it’s oftentimes much more convenient and effective to let the framework take care of it. You can do this by selecting the Init. Lead Selection property (which is selected by default whenever we create a new context node). We’ll consider the lead selection concept in further detail in Section 5.2.3.</td>
</tr>
<tr>
<td>Singleton</td>
<td>This property is used to determine whether or not a context node is defined as a singleton. We’ll describe this property in further detail in Section 5.2.4.</td>
</tr>
<tr>
<td>Supply Function</td>
<td>With this property, we can select a supply function method which will supply a context node with data on demand at runtime. We’ll see how supply functions work in Section 5.5.</td>
</tr>
</tbody>
</table>

Table 5.1 Properties of Context Nodes (Cont.)
5.2.2 Properties of Context Attributes

Much like context nodes, context attributes also require the maintenance of a fair number of properties. These properties are described in Table 5.2.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute Name</td>
<td>This property defines the context attribute’s name. The name property is used within the context API to access and modify attribute values.</td>
</tr>
<tr>
<td>Type Assignment</td>
<td>This property determines whether or not the attribute refers to an elementary type such as an integer or string, or a reference type such as an object instance. Thus, the values we have to choose from include: TYPE, TYPE REF TO. In practice, you’ll find that context attributes are rarely defined using reference types because these types don’t lend themselves towards data binding in the UI. In time, this may change, though, so it’s nice to have the option there just in case.</td>
</tr>
<tr>
<td>Type</td>
<td>This property defines the type of the context attribute. Here, we can choose between built-in types or ABAP Dictionary types as desired. However, as we progress through this book, we’ll see several benefits that can be gained by using ABAP Dictionary types.</td>
</tr>
<tr>
<td>Read-Only</td>
<td>This property determines whether or not a context attribute is write-protected. When this Boolean property is set, attribute value cannot be modified.</td>
</tr>
<tr>
<td>Default Value</td>
<td>If desired, this property can be used to set a default value for the context attribute. This property can come in handy when we’re developing input forms that need to contain default input values.</td>
</tr>
<tr>
<td>Input Help Mode</td>
<td>Depending on the nature of the data contained within a context attribute, it may make sense to provide an input help to assist the user in data entry within the UI.</td>
</tr>
</tbody>
</table>

Table 5.2 Properties of Context Attributes
In addition to the static properties outlined in Table 5.2, we should also point out that context attributes maintain a handful of dynamic properties at runtime. As we’ll see in Section 5.4.4, these dynamic properties can prove quite useful for adjusting the characteristics of certain types of UI elements on the fly.

### 5.2.3 Understanding the Lead Selection Concept

In Section 5.2.1, we briefly touched on the lead selection concept as it relates to the definition of context nodes. However, as this concept plays a significant role in determining how certain context operations are interpreted at runtime, it deserves a closer look.

Lead selections come into play whenever we have context nodes with a cardinality of $0..n$ or $1..n$. Here, whenever the node collection contains more than one context element at runtime, it is important to be able to identify which context element is currently being processed. The Web Dynpro framework identifies such context elements using the lead selection designator.

For the most part, you can think of the lead selection designator as a convenience mechanism for simplifying node access expressions. For example, as we’ll see in

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| Input Help Mode (Cont.)| As we’ll learn in Chapter 10, the Web Dynpro framework provides several different options for delivering input helps. Therefore, the Input Help Mode property can be used to determine the method of input help selection. The list of options you have to choose from for this property are as follows:  
  - Automatic  
  - Deactivated  
  - Dictionary Search Help  
  - Object Value Selector  
  - Freely Programmed |

(Input Help) Once an input help mode is selected, we can plug in a specific input help using this property. Note that the name of this property will vary based on the selection made in the Input Help Mode property.

Table 5.2 Properties of Context Attributes (Cont.)
Section 5.4, the context API uses the lead selection to implicitly refer to the currently selected context element within a given context node. We’ll also see this property at work in Chapters 7 and 8 when we look at complex UI elements used for the purposes of selection.

**Setting the Lead Selection**

As we learned in Section 5.2.1, the *Init. Lead Selection* property of a context node determines whether or not a multi-element node collection contains a pre-selected lead selection element. If the *Init. Lead Selection* property is selected, the first element within the node collection will be given the lead selection designer by default. Otherwise, the property is undefined.

In either case, the lead selection property for a context node can subsequently be adjusted in one of two ways:

- It can be set explicitly using methods provided via the context API.
- It may be adjusted via selection within a UI selection element that is bound to the corresponding context node.

**5.2.4 Understanding the Singleton Property**

As described in Section 5.2.1, it’s possible to define a context node as a *singleton*. Of course, if you’re not familiar with the concept of a singleton, a brief description is in order. The singleton design pattern is derived from the field of mathematics. There, the term *singleton* is used to describe a set with exactly one element. In their classic software engineering text, *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma, et al. demonstrated how this concept could be applied to the creation of objects (Addison-Wesley, 1995). Here, the singleton pattern is employed to ensure that only a single object instance may exist for a class at runtime.

If we think of context nodes as a type of class, it follows that the *Singleton* property for context nodes is used to ensure that only a single instance of a context node may exist at runtime. To understand how this works, let’s look at how the Singleton property affects the way a child node such as *OrderItems* is loaded into our sample *SalesOrders* context at runtime. As you can see in Figure 5.4, without the Singleton property set, multiple *OrderItems* nodes will be created at runtime: one for each sales order element in the *SalesOrders* node collection. However, if
we were to mark the Singleton property on the OrderItems node, only one instance of the OrderItems node would exist at any given point in time. Here, the only OrderItems node created is the one that corresponds with the lead selection element (see Figure 5.5). Of course, if the lead selection in the SalesOrders node collection were to change, then the OrderItems node would need to be refreshed to reflect the item records associated with the selected sales order. As we’ll learn in Section 5.5, this refresh process can be performed automatically by supply functions.

Figure 5.4 Understanding the Singleton Property (Part 1)

Figure 5.5 Understanding the Singleton Property (Part 2)
This distinction between singleton and non-singleton nodes is subtle, but often-times crucial to the overall design of a context. For instance, with our sales orders example, imagine the overhead associated with loading all the line items for every sales order a user might select for processing. For a couple of orders, it's not such a big deal. On the other hand, if the user were to load in hundreds or thousands of orders, we could have some serious memory utilization problems on our hands. By selecting the Singleton property for the `OrderItems` node, we can avoid such overhead since we'll only ever have a single `OrderItems` node in context at one time. Naturally, this can lead to huge performance benefits from a runtime perspective.

One final thing we should point out is that the performance benefits described in this section really only apply to context nodes that aren't direct descendents of the root node. If you think about it, this makes sense because there is only ever a single instance of the root node. So, even though direct descendant nodes of the root node are technically classified as singletons, there are no real performance benefits to be gained in this particular case.

### 5.3 Defining a Context

Having seen how contexts are organized and configured from a conceptual point of view, let's now turn our attention toward more practical matters and see how contexts are maintained using the graphical editor tools. As we'll see, this process is pretty straightforward once you understand how all the various pieces fit together.

#### 5.3.1 Defining Context Nodes

Context nodes are defined and maintained using the Context Editor tool, which can be accessed via the Context tab of the Controller Editor. Here, we can define new context nodes by performing the following steps:

1. In order to create a new context node, we must locate an existing node within the context that will serve as the node's parent. Initially, the default root node will be all we have to choose from here. Of course, as additional nodes are added to the mix, those nodes will become parent candidates as well. In any case, once we select the target parent node, we can create a new child node by
right-clicking on the parent node and selecting the Create • Node menu option as shown in Figure 5.6.

![Figure 5.6 Creating a Context Node (Part 1)](image1)

2. This will open the Create Nodes dialog box shown in Figure 5.7. Here, we can configure the individual properties of the context node as we like. (See Table 5.1 for a detailed description for each of the individual properties.)

![Figure 5.7 Creating a Context Node (Part 2)](image2)

3. At this point, we can save our changes by clicking on the button with the green checkmark on it. However, if the context node we’re creating is modeled in terms of a pre-existing ABAP Dictionary structure (e.g., the BAPIADDR3 structure depicted in Figure 5.7), then we also have the option of going ahead and creating context attributes based on selected component fields of that structure. We can do this by clicking on the Add Attributes from Structure button (see Figure 5.7).
4. In the Select Components of Structure dialog box shown in Figure 5.8, we can pick and choose among the various structure components that will be used to model the context attributes for the selected context node. If we wish to select all the component fields, we can do so by clicking the Select All button in the toolbar.

![Figure 5.8 Creating a Context Node (Part 3)](image)

5. Finally, once we’re satisfied with our selections, we can finish the node creation process by clicking on the button with the green checkmark on it.

After the new context node is defined, we can maintain its properties after the fact in the properties editor shown in Figure 5.9. Here, as needed, we can click on the button adjacent to the Dictionary Structure property to reopen the Select Components of Structure dialog box shown in Figure 5.8. Within this dialog box, we can add or remove context attributes by selecting or de-selecting component fields within the structure. Alternatively, we can access this editor screen by...
right-clicking on the context node and selecting the Create Using the Wizard • Attributes from Components of Structure menu option.

![Figure 5.9 Maintaining Context Node Properties](image1)

Before we wrap up our discussion on context node maintenance, we should also point out that the Context Editor offers an alternate tabular view that can be used to edit context nodes and attributes en masse. To toggle between these different views, simply click on the Switch Context Editor View button in the toolbar. Figure 5.10 illustrates the tabular view.

![Figure 5.10 Switching the Context Editor View](image2)
5.3.2 Defining Context Attributes

In the previous section, we observed how the context node creation wizard could be used to automatically create context attributes in reference to component fields of an ABAP Dictionary structure. Though this feature is handy, there will be times when we need to create context attributes by hand. In these situations, we can perform the following steps:

1. First, we'll need to select the context node that will contain the context attribute.

2. Then, once we've selected the target node, we can right-click on the target node and select the Create • Attribute menu option as shown in Figure 5.11.

3. In the CREATE ATTRIBUTE dialog box shown in Figure 5.12, we can configure the attribute properties as needed. Refer to Section 5.2.2, for more details about the properties of context attributes.

4. Finally, we can confirm our changes by clicking on the button with the green checkmark on it.
Once the context attribute is created, we can maintain its properties in the editor screen shown in Figure 5.13. This also goes for auto-generated context attributes created in reference to component fields of ABAP Dictionary structures.
5.3.3 Recursion Nodes

Most of the time, whenever we're defining a context, the depth of the node hierarchy will be known up front. For instance, when modeling a sales order, we know that the node hierarchy will be three levels deep:

- SalesOrders
  - OrderItems
    - ScheduleLines

However, what happens if we don't know the depth of the node hierarchy up front? Here, a classic example is a node hierarchy used to model a file system. In this scenario, the node hierarchy could be arbitrarily deep depending on the number of subdirectories contained within a given file system.

To model such arbitrary node hierarchies, we must enlist the aid of recursion nodes. From a conceptual perspective, a recursion node is a special kind of context node whose structure is recursively defined in terms of some superordinate context node. In effect, recursion nodes are design time placeholders that can be used to model complex node hierarchies whose depth is unknown at design time. In this section, we'll learn how work with recursion nodes.

What Is Recursion?

Recursion is the process of repeating items in a self-similar way. A classic example used to demonstrate recursion is the factorial function from the field of mathematics. Since it's probably been a while since your last high school math class, the factorial function calculates the product of all the positive integers less than or equal to a number \( n \). So, for instance, if \( n = 3 \), then \( \text{factorial}(3) = 3 \times 2 \times 1 = 6 \).
While it’s possible to define the factorial function using looping structures, it’s much more elegant to define the solution using a divide-and-conquer approach in which the function is defined in terms of itself. For example, in the ABAP Objects class listing below, you can see how the \texttt{factorial()} method repeatedly calls itself using the argument \(n - 1\). The recursion sequence bottoms out whenever \(n\) becomes less than or equal to 1, at which point the program call stack unwinds and the factorial values calculated at the lower levels bubble their way back up the call stack.

\begin{verbatim}
CLASS lcl_recursion DEFINITION.
  PUBLIC SECTION.
    CLASS-METHODS:
      factorial IMPORTING n TYPE i
        RETURNING VALUE(re_factorial) TYPE i.
    ENDCLASS.

  CLASS lcl_recursion IMPLEMENTATION.
    METHOD factorial.
      IF n LE 1.
        re_factorial = 1.
      ELSE.
        re_factorial = n * factorial(n - 1).
      ENDIF.
    ENDMETHOD.
  ENDCLASS.
\end{verbatim}

To put the nature of this recursive solution into perspective, let’s trace through a call to the \texttt{factorial()} method. Here, if we were to pass in an argument of 3, the final product would be calculated as follows:

\[
\text{factorial}(3) = 3 \times \text{factorial}(2 - 1) = 3 \times 2 \times 1 \checkmark
\]

\[
2 \times \text{factorial}(1 - 1) = 2 \times 1
\]

\[
1
\]

Once we reach the bottom of the recursion stack, we can begin plugging in the calculated values for each recursion step. So, for example, the expression \texttt{factorial}(2 - 1) evaluates to 1, \texttt{factorial}(3 - 1) evaluates to 2, and so on until we have \(3 \times 2 \times 1 = 6\).

Regardless of whether or not we’re using recursion to define a function or a complex context node hierarchy, there are a couple of important observations we can take away from the implementation of the recursive \texttt{factorial()} solution:
5.3 Defining Recursion Nodes

In order to understand how recursion nodes work, it’s useful to see how they’re defined. That way, we can begin to visualize how recursive node hierarchies are realized at runtime. To guide us through this process, let’s try to model the file system node hierarchy described earlier.

To begin, we’ll first define a regular context node called $\text{DIRECTORY}$ as shown in Figure 5.14. This context node has a cardinality of $0..n$ and defines a single context attribute called $\text{FILENAME}$. Thus, at runtime, we can use the $\text{DIRECTORY}$ node to enumerate all the files contained within a single directory.

![Figure 5.14. Modeling a File System in the Context](image)

As you can see in Figure 5.15, our $\text{DIRECTORY}$ node is rather limited in that it only allows us to model the files contained within a single directory. Here, the

- First, notice that the $\text{factorial()}$ method made no assumptions about the depth of the recursion. So, whether we pass in an argument of 2 or 100,000, $\text{factorial()}$ just goes about its business as usual.

- In order to avoid an infinite loop, we must define a base case that can be solved without recursion. In the case of the $\text{factorial()}$ method, the base case occurs whenever $n$ is less than or equal to 1.

Though recursive solutions are not always intuitive at first glance, they’re very effective for solving certain types of complex problems. This will become apparent as we search for ways to model certain types of complex context node hierarchies.
DIRECTORY node collection contains a separate element per file, with the file name being represented by the FILENAME context attribute.

In order to be able to model subdirectories, we need to add a recursion node called SUBDIRECTORY as a child of the DIRECTORY node. We can create this node by right-clicking on the DIRECTORY node and selecting the CREATE • RECURSION NODE menu option. This will open the CREATE RECURSION NODE dialog box shown in Figure 5.16. As you can see, recursion node definitions consist of a node name and a reference node that’s used to describe the layout of the recursion node(s) at runtime. We can confirm our selection by clicking on the button with the green checkmark on it. After the recursion node is created, we can view its definition by clicking on the folder with the circular blue arrows on it (see Figure 5.17).

Figure 5.15 Visualizing the Directory Context Node at Runtime

Figure 5.16 Creating a Recursion Node
Recursion Nodes at Runtime

Getting back to our file system example, let’s once again take a look at our `DIRECTORY` node at runtime now that we’ve added the `SUBDIRECTORY` recursion node. As you can see in Figure 5.18, the addition of the `SUBDIRECTORY` node now allows us to represent subdirectories within a given directory. Now, we have two different types of context elements within a `DIRECTORY` (or `SUBDIRECTORY`) node collection:

- If the context element represents a file, the `FILENAME` attribute contains the file name and the `SUBDIRECTORY` node collection is empty.
- If the context element represents a directory, the `FILENAME` attribute contains the directory name and the `SUBDIRECTORY` node collection will contain additional elements (and subelements) to represent the underlying directory structure.

We can then extend this structure out indefinitely to represent arbitrarily nested file structures as needed. Of course, not all of this happens by magic; we still have to programmatically create `SUBDIRECTORY` elements to represent the various subdirectories. We’ll see how this is accomplished in Section 5.4.
Restrictions for Using Recursion Nodes

When working with recursion nodes, there are a couple of restrictions we need to be aware of:

- First of all, we cannot use the root node of a context as the repeated node for a recursion. When you think about it, this makes sense because it would lead to circular references.
- Secondly, we cannot map a recursive node from one context to another (via context mapping). In other words, in our file system example, we could not map the `SUBDIRECTORY` node by itself. However, this isn’t to say that we couldn’t map the superordinate `DIRECTORY` node. The point is that recursion nodes cannot exist in isolation.

5.4 Context Programming

Up until now, we have been primarily focused on the design-time characteristics of contexts. With this foundation in place, we’re now ready to begin looking at contexts from a runtime perspective. Therefore, in this section, we’ll explore the use of the provided context API and see how contexts are built and manipulated using good old-fashioned ABAP code.

5.4.1 Getting Started

Behind the scenes, the Web Dynpro framework is responsible for coming up with in-memory representations of the context node hierarchies we define within the graphical Context Editor tool. As you might expect, defining such an in-memory representation is no trivial matter, especially when you begin to think about all the different properties that can be configured for individual context nodes and context attributes within a node hierarchy.

In order to shield developers from the underlying complexity of context implementation details, SAP has provided a simplified API that allows us to traverse through and manipulate contexts in a straightforward and intuitive manner. Figure 5.19 contains a UML class diagram, which shows the primary interfaces we’ll interact with when utilizing this API.
As you can see, these interfaces embody basic elements of a context: context nodes are represented via interface IF_WD_CONTEXT_NODE, context elements are represented via interface IF_WD_CONTEXT_ELEMENT, and so on.

In order to understand the positioning of the context API, let’s take a closer look at how contexts are constructed behind the scenes. Whenever a controller is instantiated at runtime, the Web Dynpro framework assumes the responsibility of building an in-memory representation of a context. This representation is based on the design time metadata contained within a controller’s context definition. The implementation of this representation, while proprietary, is realized in the form of objects, which implement the interfaces shown in Figure 5.19.
In order to tap into this auto-generated context hierarchy, we must obtain a reference to the context root node. Fortunately, as it turns out, this reference is provided to us automatically via the default WD_CONTEXT attribute, which is included with every controller within a WDA component. As you can see in Figure 5.20, this attribute is defined as an object reference, which points to an object that implements the IF_WD_CONTEXT_NODE interface.

Once we have access to the context root node, we can use the methods of the IF_WD_CONTEXT_NODE interface to begin traversing through the context hierarchy, adding and removing context elements, and so on. In the upcoming sections, we'll see how all this works by considering some practical programming use cases. To guide us through these exercises, we'll work with the SALES_ORDERS context hierarchy shown in Figure 5.21. Here, each of the nested context nodes (SALES_ORDERS, ORDER_ITEMS, and SCHEDULE_LINES) have a cardinality of 0..n. As we progress through these examples, we'll refer back to this sample context periodically to see how its properties influence the approaches we'll take to manipulate the context at runtime.

Note
You can also see how this context is defined by looking at component ZWDC_CONTEXT_DEMO within the chapter examples provided with the code bundle for this book.
5.4.2 Maneuvering within a Context

At the end of the day, context programming is mostly about performing CRUD operations on the context node collections contained within a context hierarchy. Of course, in order to perform these operations, we must figure out a way to get our hands on the individual context nodes and context elements contained within a context hierarchy. Therefore, in this section, we’ll spend some time looking at ways to traverse through the context nodes and context elements contained within a context.

Accessing Child Nodes Directly

As we learned in the previous section, each controller within a WDA component contains a default attribute called `WD_CONTEXT`, which points to the root node of the controller’s context. We can use the methods of the `WD_CONTEXT` attribute to traverse the node hierarchy in several different ways.

One of the most common approaches for navigating from one context node to another is to use the `GET_CHILD_NODE()` method provided via the `IF_WD_CONTEXT_NODE` interface. Listing 5.1 demonstrates how this method works. Here, we’re using the method to traverse two levels down in the sample context depicted in Figure 5.21:

- First, we use the method to traverse from the context root node to the `SALES_ORDERS` node.
- Then, we call the method on the provided sales orders object reference to traverse down to the child `ORDER_ITEMS` node.

```plaintext
METHOD wddoinit.
  DATA lo_sales_orders_node TYPE REF TO if_wd_context_node.
  DATA lo_item_node TYPE REF TO if_wd_context_node.
  lo_sales_orders_node =
      wd_context->get_child_node( 'SALES_ORDERS' ).
  lo_item_node =
      lo_sales_orders_node->get_child_node( name = 'ORDER_ITEMS',
                                                index = 5 ).
ENDMETHOD.
```

**Listing 5.1** Accessing a Child Node Using the `GET_CHILD_NODE()` Method
Looking at the signature of the `GET_CHILD_NODE()` method illustrated in Figure 5.22, you can see that the method can receive two importing parameters:

- **NAME**
  As you would expect, the `NAME` parameter refers to the name of the node being accessed (e.g., `SALES_ORDERS` or `ORDER_ITEMS`).

- **INDEX**
  The `INDEX` parameter is an optional parameter that can be used to specify which context element within the node element collection of the referenced context node we want to perform the lookup on. For example, in Listing 5.1, notice that we passed a value of 5 for the `INDEX` parameter in the call to look up the `ORDER_ITEMS` node. This call can be interpreted as a lookup for order items for the fifth sales order element in the referenced `SALES_ORDERS` node's element collection. If the `INDEX` parameter isn't specified, the method will use the lead selection element by default (see Figure 5.22).

The `GET_CHILD_NODE()` method is useful in situations when we need to access a specific child node. However, sometimes we may need to look up all the child nodes for a given context node at once. In these situations, we can use the `GET_CHILD_NODES()` method of the `IF_WD_CONTEXT_NODE` interface. This method will return a table of context node references we can manipulate in a loop as per usual.
Finally, if we already happen to have our hands on a context element, we can access a child node for that element directly using the `GET_CHILD_NODE()` method defined in the `IF_WD_CONTEXT_ELEMENT` interface. The primary difference here is that this version of the `GET_CHILD_NODE()` method doesn't provide an `INDEX` parameter since the parent context element is implied from the context element object reference. We'll learn how to obtain references to context elements shortly.

### Best Practices: Avoiding the Use of String Literals

For demonstration purposes, we used the string literal `SALES_ORDERS` to access the `SALES_ORDERS` child node in the code excerpt in Listing 5.1. However, from this point forward, we'll be utilizing constants from the controller interface for this purpose. You might ask, where do these constants come from? They're defined automatically by the system whenever we define context nodes within a controller context. So, if we define a context node called `SALES_ORDERS`, a constant called `WDCTX_SALES_ORDERS` will be created behind the scenes within the controller interface (which can be accessed via the `WD_THIS` attribute). We can then plug in this constant within the call to `GET_CHILD_NODE()` like this:

```abap
lo_sales_orders_node =
wd_context->get_child_node( wd_this=>wdctx_sales_orders ).
```

What's the difference? Well, by using constants, we allow the ABAP syntax check to verify that we're accessing a proper node name. Otherwise, there's nothing to stop us from accidentally entering a typo and having the lookup fail at runtime.

### Accessing Descendant Nodes Using Paths

Though it's possible to use the `GET_CHILD_NODE()` method repeatedly to access the descendant nodes of a given context node, it can be much more efficient to bypass these steps and access descendant nodes directly using path lookup methods. To understand how these path methods work, consider the code excerpt in Listing 5.2. As you can see, we're using the `PATH_GET_NODE()` method to access the `SCHEDULE_LINES` node from our sample hierarchy. Here, you can evaluate the path expression `SALES_ORDERS.ORDER_ITEMS.SCHEDULE_LINES` in the same way you'd evaluate a path used to access a file in the file system of your local computer: `C:\Folder1\Folder2\MyFile.txt`. 

```abap```
When building path expressions, we must be careful to take node cardinalities into account. For example, the path expression contained in Listing 5.2 uses the lead selections for the `SALES_ORDERS` and `ORDER_ITEMS` nodes to obtain the targeted `SCHEDULE_LINES` node reference. If we want to access the schedule lines of the second line item of the third sales order, we would have to build a path expression like the one in Listing 5.3. In this case, the target index is specified after the context node using the syntax `<context node>.<index>`.

Most of the time, we’ll probably know the node path we want to query ahead of time. However, if we need to derive the node path dynamically in relation to a particular node, we can use the `GET_PATH()` method against that node to determine the current node path and then build our path expression from there. In addition, we can also use the `PATH_GET_ELEMENT()` and `PATH_GET_ATTRIBUTE()` methods to navigate to context elements and context attributes, respectively.

**Accessing Context Elements**

Once we get our hands on a context node reference, we can use the methods of the `IF_WD_CONTEXT_NODE` interface to access context elements contained within the context node's element collection. Table 5.3 describes each of these methods in turn.
### Table 5.3 Methods for Accessing Context Elements within a Node Collection

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| get_element | This method uses index-based semantics to access a particular context element within the referencing context node’s element collection. If a specific index value is not provided via the optional INDEX parameter, the method will return the context element designated as the lead selection. Otherwise, the method will attempt to return the context element at the specified position. So, for example, if we want to access the fifth sales order element within the SALES_ORDERS node collection, we could do so like this:  
**DATA:** lo_sales_orders  
**TYPE** REF TO if_wd_context_node,  
lo_sales_order  
**TYPE** REF TO if_wd_context_element.  
lo_sales_order =  
lo_sales_orders->get_element(5).  
**IF NOT** lo_sales_order IS INITIAL.  
*Do stuff...*  
**ENDIF.**  
As you can see in the code excerpt above, it’s always a good idea to perform a null check on the resultant context element because it may not always be mapped. For example, if we were to pass an index value of 11 to the GET_ELEMENT() method for a context node whose node collection contained only 10 context elements, the operation would be undefined. In this case, the GET_ELEMENT() method would return the null value instead of an instance of IF_WD_CONTEXT_ELEMENT. Naturally, this leads to null reference exceptions at runtime. |
| get_elements | If we want to obtain a list of all the context elements contained within a node collection, we can utilize the GET_ELEMENTS() method. This method returns a table of IF_WD_CONTEXT_ELEMENT references, which represent each of the elements in the collection.  
If we like, we can use the provided FROM and TO parameters to whittle down the list to a subset of values like this:  
**DATA:** lo_sales_orders  
**TYPE** REF TO if_wd_context_node,  
lt_order_elems  
**TYPE** wdr_context_element_set. |
Working with Contexts

In addition to the methods provided for accessing context elements contained within a node’s collection, the `IF_WD_CONTEXT_NODE` interface provides a method that allows us to access the parent element of the selected node: `GET_PARENTELEMENT()`. This method can be handy for dynamic programming situations when we need to walk up the node hierarchy one step at a time.

### Working with the Lead Selection Property

In Section 5.2.3, we observed the ways in which the Web Dynpro framework uses the lead selection property to identify which context element is currently being processed within a node collection. In practice, the lead selection provides us with a very convenient way to address context elements within a node hierarchy without having to worry about keeping up with indices. Of course, in order for all this to work, we must be careful to make sure the lead selection is kept up to date. For these tasks, we can use the methods outlined in Table 5.4.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_elements (Cont.)</td>
<td>FIELD-SYMBOLS: <code>&lt;lfs_order&gt;</code> LIKE LINE OF lt_order elems.</td>
</tr>
<tr>
<td></td>
<td><code>lt_order elems = lo_sales_orders-&gt;get_elements( from = 2 to = 4 ).</code></td>
</tr>
<tr>
<td></td>
<td>LOOP AT lt_order elems ASSIGNING <code>&lt;lfs_order&gt;</code>.</td>
</tr>
<tr>
<td></td>
<td>&quot;Process the order element...&quot;</td>
</tr>
<tr>
<td>get_element_count</td>
<td>As the method name suggests, this method provides us with a count of the number of context elements contained within a node collection. This method can come in handy when accessing context elements within a loop. Here, the resultant value can be used as an upper bound on the loop to make sure that calls to <code>GET_ELEMENT()</code> are made with valid index values.</td>
</tr>
</tbody>
</table>

Table 5.3 Methods for Accessing Context Elements within a Node Collection (Cont.)
<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_lead_selection</td>
<td>This method returns the context element at the lead selection. Behavior-wise, it’s semantically equivalent to calling the GET_ELEMENT() method without passing in a specific index value.</td>
</tr>
<tr>
<td>get_lead_selection_index</td>
<td>This method returns the index of the lead selection element as an integer. As such, it can be useful for determining where the lead selection is with respect to the rest of the elements within the node collection.</td>
</tr>
<tr>
<td>move_first</td>
<td>This method marks the first context element in the node collection as the lead selection.</td>
</tr>
<tr>
<td>move_last</td>
<td>This method marks the last context element in the node collection as the lead selection.</td>
</tr>
<tr>
<td>move_next</td>
<td>This method moves the lead selection indicator to the next element in the node collection. So, if the lead selection index is currently at 2, a call to MOVE_NEXT() will move the lead selection index to the element at position 3. If the last element within the collection is currently designated as the lead selection, the method will return without making any changes to the lead selection.</td>
</tr>
<tr>
<td>move_previous</td>
<td>This method is the analog of MOVE_NEXT() in that it moves the lead selection pointer to the previous element in the context node collection. If the first element in the list is marked as the lead selection, the method will return without making any changes to the lead selection.</td>
</tr>
<tr>
<td>move_to</td>
<td>This method allows us to move the lead selection pointer to the element at the specified index, assuming it exists.</td>
</tr>
<tr>
<td>set_lead_selection</td>
<td>Unlike the other methods we’ve seen that have used index-based semantics, this method allows us to pass in the actual context element we wish to designate as the lead selection.</td>
</tr>
</tbody>
</table>

Table 5.4 Methods for Querying and Setting the Lead Selection
5.4.3 Filling a Context

Now that we’ve seen how to maneuver within a context, let’s spend some time exploring ways to fill a context. In this section, we’ll consider some of the methods we have at our disposal for carrying out these tasks.

Creating New Context Elements

For the most part, the context representation generated by the Web Dynpro framework is empty by default. So, in order to begin loading data into a context, we must create new context elements to fill in the context nodes contained within the node hierarchy. However, since the only representation of a context element we have to work with is the `IF_WD_CONTEXT_ELEMENT` interface, we cannot instantiate new context elements using the `CREATE OBJECT` statement. Instead, we must rely on a series of factory methods provided within the `IF_WD_CONTEXT_NODE` interface. In this section, we’ll take a look one of these methods in particular: the `CREATE_ELEMENT()` method.
As the name suggests, the \texttt{CREATE\_ELEMENT()} method creates a brand new context element. Listing 5.4 demonstrates how this works.

\begin{verbatim}
DATA: lo_sales_orders TYPE REF TO if_wd_context_node,
     lo_sales_order TYPE REF TO if_wd_context_element.
lo_sales_orders = ld_context->get_child_node( wd_this=>wdctx_sales_orders ).
lo_sales_order = lo_sales_orders->create_element( ).
\end{verbatim}

\textbf{Listing 5.4} Creating a Context Element Using \texttt{CREATE\_ELEMENT()}

By default, the context attributes for the generated context element returned from the call to \texttt{CREATE\_ELEMENT()} will be set to the initial values of their corresponding data types. We can pre-initialize these values using the \texttt{STATIC\_ATTRIBUTE\_VALUES} parameter as demonstrated in Listing 5.5. Here, notice that we're using the \texttt{wd_this->Element\_sales\_orders} structure type to prefill the context attributes. This structure type was generated for us automatically within the controller interface when we created the \texttt{SALES\_ORDERS} node in the Context Editor and provides a convenient way for us to specify context attribute values en masse. Of course, we're also free to define our own structure type if we prefer. In either case, the Web Dynpro framework will map the component fields of the structure in much the same way component fields are copied from one structure to another using the \texttt{MOVE\_CORRESPONDING} statement in ABAP.

\begin{verbatim}
DATA: lo_sales_orders TYPE REF TO if_wd_context_node,
     lo_sales_order TYPE REF TO if_wd_context_element,
     ls_sales_order TYPE wd_this->Element_sales_orders.
   "Initialize the attributes of the new sales order:
   ls_sales_order-order_no = '1234567890'.
   ls_sales_order-creation_date = sy-datum.
   ls_sales_order-customer_no = '3456789012'.
lo_sales_orders = ld_context->get_child_node( wd_this=>wdctx_sales_orders ).
lo_sales_order = lo_sales_orders->create_element( ls_sales_order ).
\end{verbatim}

\textbf{Listing 5.5} Creating a Pre-Initialized Context Element
Adding Context Elements to a Node Collection

In the previous section, we learned how the CREATE_ELEMENT() method could be used to create new context elements. By default, these newly created context elements exist in isolation. In other words, they aren't yet part of the element collection of the context node whose reference was used to create them.

In order to add the context element to the node's element collection, we must bind the element to the context node. This bind operation is similar to the APPEND statement used to add new lines to an internal table.

The IF_WD_CONTEXT_NODE interface provides four different methods for binding context elements to a node collection. The first method we'll look at is the BIND_ELEMENT() method. This method's three importing parameters influence its behavior at runtime:

- **NEW_ITEM**
  This parameter contains the context element that will be added to the node collection.

- **SET_INITIAL_ELEMENTS**
  This optional Boolean parameter determines whether or not we want to refresh the node collection before appending the selected context element.

- **INDEX**
  This optional integer parameter can be used to specify the precise index into which we want to insert the selected context element. If this parameter is not specified, the context element will be added to the end of the node collection.

Assuming we want to accept the default behavior for the BIND_ELEMENT() method, we can extend our element creation example from Listing 5.5 to append the generated sales orders element to the SALES_ORDERS node collection as demonstrated in Listing 5.6.

```plaintext
DATA: lo_sales_orders TYPE REF TO if_wd_context_node,
     lo_sales_order TYPE REF TO if_wd_context_element,
     ls_sales_order TYPE wd_this->element_sales_orders.

"Initialize the attributes of the new sales order:
...

lo_sales_orders =
  ld_context->get_child_node( wd_this=>wdctx_sales_orders ).
```
lo_sales_order =
    lo_sales_orders->create_element( ls_sales_order ).
lo_sales_orders->bind_element( lo_sales_order ).

Listing 5.6 Adding a Context Element to a Node Collection

To add a series of context elements to a node collection in one fell swoop, we can use the `BIND_ELEMENTS()` method. This method behaves in much the same way that the `BIND_ELEMENT()` method does. The primary difference is in that `BIND_ELEMENTS()` allows us to pass a table of context elements in its `NEW_ITEMS` parameter.

As we’ve seen, the `CREATE_ELEMENT()` and `BIND_ELEMENT()` methods are fairly straightforward to work with. However, if we find ourselves having to build a complex node hierarchy, this repeated call sequence can get tedious in a hurry. Fortunately, the `IF_WD_CONTEXT_NODE` interface provides a couple of shortcut methods to simplify this process.

The first shortcut method we’ll look at is the `BIND_STRUCTURE()` method. This method consolidates the calls to `CREATE_ELEMENT()` and `BIND_ELEMENT()` into single method call as demonstrated in Listing 5.7. This call has the same effect as the previous two calls, providing us with the same parameters that `BIND_ELEMENT()` uses to determine how the element is added to the node collection.

Listing 5.7 Creating and Appending Elements Using `BIND_STRUCTURE()`

In addition to the `BIND_STRUCTURE()` method, the `IF_WD_CONTEXT_NODE` interface provides the `BIND_TABLE()` method for appending context elements en masse. In effect, a call to this method is equivalent to building a series of context elements
using `CREATE_ELEMENT()`, storing them in an internal table, and then adding them to the node collection via the `BIND_ELEMENTS()` method.

Listing 5.8 shows how the `BIND_TABLE()` method works. As you can see, the majority of this code is business as usual in ABAP: we’re simply building an internal table. The type of this internal table was defined for us automatically within the controller interface when we created the `SALES_ORDERS` node. So, rather than reinventing the wheel, we simply define the local table variable `LT_SALES_ORDERS` using the `Elements_Sales_Orders` table type. Finally, once the table is built, we can simply pass it to the `BIND_TABLE()` method in order to create the context elements. If you look at the signature of the `BIND_TABLE()` method, you’ll see that it provides us with the same parameter choices as `BIND_ELEMENTS()`, allowing us to influence how the elements are appended at runtime.

```abap
DATA: lo_sales_orders TYPE REF TO if_wd_context_node,
  lt_sales_orders TYPE wd_this->Elements_sales_orders.
FIELD-SYMBOLS:
  <lfs_sales_order> LIKE LINE OF lt_sales_orders.

"Create a couple of sales orders elements:
APPEND INITIAL LINE TO lt_sales_orders
  ASSIGNING <lfs_sales_order>.
  <lfs_sales_order>-order_no = '1234567890'.
  <lfs_sales_order>-creation_date = sy-datum.
  <lfs_sales_order>-customer_no = '3456789012'.

APPEND INITIAL LINE TO lt_sales_orders
  ASSIGNING <lfs_sales_order>.
  <lfs_sales_order>-order_no = '234567890'.
  <lfs_sales_order>-creation_date = sy-datum.
  <lfs_sales_order>-customer_no = '3456789012'.

lo_sales_orders =
  ld_context->get_child_node( wd_this=>wdctx_sales_orders ).
lo_sales_orders->bind_table( lt_sales_orders ).
```

**Listing 5.8** Adding Elements to a Node Collection Using `BIND_TABLE()`
5.4.4 Working with Context Attributes

Since we learned how to create context elements in the previous sections, let's now turn our attention to the maintenance of the context attributes contained within these context elements. For the most part, you'll find these operations to be quite straightforward. After all, there's only so much you can do to a set of name-value pairs. However, as we'll soon see, context attributes also maintain a handful of dynamic properties that can be used to influence the behavior of certain types of UI elements.

To query or set the values of context attributes, we must use the methods described in Table 5.5. As you might expect, these methods are provided via the IF_WD_CONTEXT_ELEMENT interface. Thus, you can think of the attribute access operations from the perspective of the referencing context element.

**Note**

For convenience, you'll also find versions of the same methods provided with interface IF_WD_CONTEXT_NODE. However, these methods are really just wrapper methods in that they delegate the request to a selected context element behind the scenes (either via lead selection or explicit index selection).

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_attribute</td>
<td>This method can be used to obtain the value of a context attribute. The code excerpt below demonstrates what this call looks like. Here, assume the object reference LO_SALES_ORDER points to some context element. As you can see, the method call simply takes in a context attribute name and returns the attribute's value.</td>
</tr>
<tr>
<td></td>
<td>DATA: lv_order_date TYPE datum.</td>
</tr>
<tr>
<td></td>
<td>lo_sales_order-&gt;get_attribute(</td>
</tr>
<tr>
<td></td>
<td>EXPORTING name = <code>CREATION_DATE</code></td>
</tr>
<tr>
<td></td>
<td>IMPORTING value = lv_order_date ).</td>
</tr>
</tbody>
</table>

Table 5.5 Methods for Working with Context Attributes
### Methods for Working with Context Attributes (Cont.)

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| **get_attribute_ref**| This method is similar to `GET_ATTRIBUTE()` in that it allows us to access the value of a given context attribute. However, in this case, the attribute value is returned using reference semantics. This subtle variation allows us to avoid a potentially expensive memory copy operation in certain situations. The code excerpt below demonstrates how the `GET_ATTRIBUTE_REF()` method works. If you’re unfamiliar with data references, we recommend you check out the ABAP help documentation in the section entitled “Data References.”  
DATA: lr_order_date TYPE REF TO datum.  
DATA lv_order_date TYPE datum.  

lr_order_date ?=  
lo_sales_order->get_attribute_ref(  
`CREATION_DATE` ).  

*We can access the value by dereferencing the reference variable like this:*  
lv_order_date = lr_order_date->*. |
| **set_attribute**    | This method is the analog of the `GET_ATTRIBUTE()` method, allowing us to map a value onto a context attribute by name. The code excerpt below demonstrates how this works.  
DATA: lv_order_date TYPE datum.  
lv_order_date = sy-datum + 7.  

lo_sales_order->set_attribute(  
EXPORTING  
name = `CREATION_DATE`  
value = lv_order_date ). |
| **get_static_attributes** | This method is used to obtain the values of all the statically defined context attributes contained within a context element in one fell swoop. |
get_static_attributes (Cont.)

The values are copied over to a structured data type, whose component fields mirror the context attributes contained within the referencing context element as demonstrated in the code excerpt below:

```
DATA:
  lo_order TYPE REF TO if_wd_context_element,
  ls_order TYPE wd_this->element_sales_orders.

lo_order->get_static_attributes(
  IMPORTING
  static_attributes = ls_order).
```

get_static_attributes_ref

Much like the GET_ATTRIBUTE_REF() method considered earlier, this method uses reference semantics to pass back a reference to the statically defined attributes of a context element. As was the case with GET_ATTRIBUTE_REF(), this method comes in handy whenever we need to avoid expensive memory copy operations for large context elements.

set_static_attributes

This method is the analog of GET_STATIC_ATTRIBUTES() in that it allows you to initialize the values of the statically defined attributes of a context element in one fell swoop using a structured data object. The code excerpt below demonstrates how this method is used.

```
DATA:
  lo_order TYPE REF TO if_wd_context_element,
  ls_order TYPE wd_this->element_sales_orders.

  "Adjust the static attributes of an order:
  ls_order-order_no = '1234567890'.
  ls_order-creation_date = sy-datum.
  ls_order-customer_no = '3456789012'.

lo_order->set_static_attributes(
  ls_order).
```

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_static_attributes</td>
<td>The values are copied over to a structured data type, whose component fields mirror the</td>
</tr>
<tr>
<td></td>
<td>context attributes contained within the referencing context element as demonstrated in the</td>
</tr>
<tr>
<td></td>
<td>code excerpt below:</td>
</tr>
<tr>
<td>get_static_attributes_ref</td>
<td>Much like the GET_ATTRIBUTE_REF() method considered earlier, this method uses reference</td>
</tr>
<tr>
<td></td>
<td>semantics to pass back a reference to the statically defined attributes of a context</td>
</tr>
<tr>
<td></td>
<td>element. As was the case with GET_ATTRIBUTE_REF(), this method comes in handy whenever we</td>
</tr>
<tr>
<td></td>
<td>need to avoid expensive memory copy operations for large context elements.</td>
</tr>
<tr>
<td>set_static_attributes</td>
<td>This method is the analog of GET_STATIC_ATTRIBUTES() in that it allows you to initialize the</td>
</tr>
<tr>
<td></td>
<td>values of the statically defined attributes of a context element in one fell swoop using a</td>
</tr>
<tr>
<td></td>
<td>structured data object. The code excerpt below demonstrates how this method is used.</td>
</tr>
</tbody>
</table>

Table 5.5 Methods for Working with Context Attributes (Cont.)
Runtime Properties of Context Attributes

At the end of the day, context attributes are intended to be used as a means for transferring data back and forth between UI elements in the view layer and the backend data model. Here, the properties of UI elements are bound to context attributes in such a way that data is synchronized automatically.

Normally, when we talk about data binding with UI elements, we’re talking about binding the value of a context attribute to a property of a UI element used for displaying data. For example, to display the value of a context attribute using an `InputField` UI element, we’d bind the `value` property of the `InputField` element with the context attribute. However, it’s also possible to use data binding to define the values of other UI element properties, such as the `enabled` property, `readOnly` property, and so on.

Recognizing the benefits of using data binding to determine the values of certain common UI element properties, SAP elected to extend context attributes to include additional runtime properties that could be used for this purpose. The benefit here is that we don’t have to incur the overhead of defining additional context attributes to specify these UI element properties. So, for instance, we could use a single context attribute to define not only the `value` property of an
InputField element, but also its enabled property, readOnly property, and so on. This leads to much better performance at runtime.

Table 5.6 describes the set of runtime properties available for a given context attribute. Each of these properties is Boolean in nature and can be maintained using methods provided via the IF_WD_CONTEXT_ELEMENT interface as described in Table 5.7.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enabled</td>
<td>This property can be used to determine whether or not a UI element is enabled within the view.</td>
</tr>
<tr>
<td>readOnly</td>
<td>This property can be used to adjust the readOnly property of UI elements.</td>
</tr>
<tr>
<td>state</td>
<td>This property can be used to determine whether or not a UI element is considered to be a required field in the view layout.</td>
</tr>
<tr>
<td>visible</td>
<td>This property can be used to determine whether or not a UI element is visible within the view layout.</td>
</tr>
</tbody>
</table>

Table 5.6 Runtime Properties of Context Attributes

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_attribute_property</td>
<td>This method is used to set a single runtime property of a context attribute. The individual runtime properties are enumerated via the E_PROPERTY constant defined within the IF_WD_CONTEXT_ELEMENT interface. The code excerpt below demonstrates how this method call works.</td>
</tr>
</tbody>
</table>

DATA:
lo_order TYPE REF TO IF_WD_CONTEXT_ELEMENT.

lo_order->set_attribute_property(
  EXPORTING
  attribute_name = 'ORDER_NO'
  property       = lo_order->e_property-read_only
  value          = abap_true ).

Table 5.7 Maintaining the Runtime Properties of a Context Attribute
Once the runtime properties for a context attribute are in place, we can use them to define context bindings for UI elements in a view using the familiar Define Context Binding dialog box, shown in Figure 5.23. However, in this case, we’ll want to choose the target runtime property in the Property list underneath the Bind to the Property of the Selected Attribute radio button as shown in Figure 5.23.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_attribute_props_for_elem</td>
<td>This method allows us to set the runtime properties for a set of context attributes en masse. The attribute properties are specified within the PROPERTIES table parameter provided in the method signature.</td>
</tr>
<tr>
<td>get_attribute_properties</td>
<td>This method allows us to query the status of each of the runtime properties of a context attribute. Here, we simply pass in the attribute’s name in the ATTRIBUTE_NAME parameter and the method will return a structure containing the current property values.</td>
</tr>
<tr>
<td>get_attribute_props_for_elem</td>
<td>This method performs a wholesale lookup of attribute runtime properties for the referencing context element.</td>
</tr>
</tbody>
</table>

Table 5.7 Maintaining the Runtime Properties of a Context Attribute (Cont.)

Figure 5.23 Defining Context Bindings Using Runtime Properties
If you look closely at Figure 5.23, you can see that there’s also an Invert Attribute/Property (“NOT”) checkbox that can be used to invert the value of a runtime property as needed. So, for example, if the readOnly property for a context attribute has a value of abap_false, the value abap_true will be used for the purposes of data binding.

5.4.5 Performing Housekeeping Duties within a Context

During the course of a WDA application session, we may need to move context elements around or even remove them from a context node collection altogether. In this section, we’ll see how these tasks are carried out using methods of the IF_WD_CONTEXT_NODE interface.

Removing Context Elements from a Node Collection

To remove a context element from a node collection, we can use the REMOVE_ELEMENT() method. Listing 5.9 shows how this method can be used to remove the lead selection element from a node collection. We can determine whether or not the operation was successful using the resultant Boolean value from the method call.

```
DATA: lo_sales_orders_node TYPE REF TO if_wd_context_node,
     lo_sales_order_elem TYPE REF TO if_wd_context_element,
     lv_removed TYPE abap_bool.

"Remove the lead selection element from the collection:
lo_sales_orders_node = wd_context->get_child_node( 
     name = wd_this->wdctx_sales_orders ).
lo_sales_order_elem = lo_sales_orders_node->get_element( ).

"The variable LV_REMOVED will tell us whether or not the 
"operation was successful.
 lv_removed =
     lo_sales_orders_node->remove_element( 
     lo_sales_orders_elem ).
```

Listing 5.9 Removing Elements from a Node Collection

Once a context element is removed from the node collection, the context elements to the right of the removed element will be shifted to the left by one
position. So, for example, if we have a node collection containing ten elements, and we remove the element at index five, the elements at indexes six through ten will shift left such that the element formerly at index six is now at index five, and so on.

**Moving Context Elements within a Node Collection**

Occasionally, we may want to adjust the order of context elements within a node collection. For instance, if we’re using a node collection to represent the contents of a Table UI element, we may need to provide methods for sorting the elements in the node collection to correspond with sorting criteria specified within the view layer. For this task, we can enlist the aid of the `MOVE_ELEMENT()` method.

As the name suggests, the `MOVE_ELEMENT()` method allows us to move a context element from one index to another. Listing 5.10 demonstrates how this works. Here, we’re using the `FROM` and `TO` parameters to determine the source and target indexes for the move operation.

```java
DATA: lo_sales_orders_node TYPE REF TO if_wd_context_node.

"Move the element at position 4 to position 8:
lo_sales_orders_node = wd_context->get_child_node( 
    name = wd_this->wdctx_sales_orders ).
lo_sales_orders_node->move_element( from = 4 to = 8 ).

Listing 5.10  Moving Elements Around Using MOVE_ELEMENT()
```

When working with the `MOVE_ELEMENT()` method, be aware that the underlying method implementation will silently adjust erroneous index values before performing the move operation. For example, if we were to pass a `FROM` value of –1 or a `TO` value greater than the size of the node collection, the method would move the context element at position 1 to the end of the list. Sometimes, this may be what you want, but it’s nevertheless dangerous to leave this to chance. Therefore, when sorting node collections, we recommend you double-check your index-derivation logic and use the `GET_ELEMENT_COUNT()` method to make sure your indexes are in bounds.
5.4.6 Working with the Context Change Log

During the course of the lifecycle of a context, there will be many changes made to context nodes. At the outset, nodes collections will be filled in a controlled manner using API methods as demonstrated throughout the course of this section. From that point forward, however, changes may be initiated from a couple of different places:

- From users entering data in UI elements bound to context elements or context attributes contained within the node’s element collection
- From event handler methods in controllers that are designed to add, change, or remove context elements on demand

Most of the time, any changes made to a context will need to be synchronized with the model layer. Though it’s possible to perform these updates using a “kill-and-fill” approach in which updates are performed indiscriminately \textit{en masse}, it can be much more efficient to update only the values that actually changed. Of course, the problem with the latter approach is trying to figure out exactly what was changed. In particular, we need to know what was changed by users at the view layer. Fortunately, the Web Dynpro framework provides us with a context change log that can be used to track changes made by users on the frontend. In this section, we’ll see how to put this change log to use.

Flipping the Switch on the Context Change Log

Strictly speaking, the context change log is not associated with any particular context node. Instead, it’s used to track changes made by users for all of the context nodes contained within the overarching controller context. Therefore, instead of being aligned with an instance of \texttt{IF\_WD\_CONTEXT\_NODE}, the context change log is accessed via methods provided within the \texttt{IF\_WD\_CONTEXT} interface.

To turn on the context change log, we can use the \texttt{ENABLE\_CONTEXT\_CHANGE\_LOG()} method demonstrated in Listing 5.11. Similarly, we can later turn off the context change log by calling the \texttt{DISABLE\_CONTEXT\_CHANGE\_LOG()} method, which is also demonstrated in Listing 5.11.

*Obtain a reference to the overarching controller context:

\begin{verbatim}
  DATA: lo_context TYPE REF TO if_wd_context.
  lo_context = wd_context->get_context( ).
\end{verbatim}
"Turn on the context change log:
lo_context->enable_context_change_log( ).

...

"Turn off the context change log:
lo_context->disable_context_change_log( ).

Listing 5.11 Turning on the Context Change Log

Accessing the Change Log

Once the context change log is turned on, it will begin silently tracking changes behind the scenes. At any point along the way, we can view the contents of the change log by invoking the `GET_CONTEXT_CHANGE_LOG()` method of the `IF_WD_CONTEXT` interface. This method will return a table with a line type based on the `WDR_CONTEXT_CHANGE` structure illustrated in Figure 5.24. As you can see, this table provides us with access to the type of change, the corresponding context node/element/attribute, and the old/new values.

<table>
<thead>
<tr>
<th>Structure</th>
<th>WDR_CONTEXT_CHANGE</th>
<th>Active</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Description</td>
<td>Web Dynpro: Change to Context Entry</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Components</th>
<th>Entry help/check</th>
<th>Currency/quantity fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 / 9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Figure 5.24](image)

Figure 5.24 Evaluating Changes in the Context Change Log

If you're interested in seeing how all this works in an actual application, check out the `DEMO_CONTEXT_CHANGES` component in the `SWDP_DEMO` package. You can also find further details concerning the use of the context change log in the documentation for interface `IF_WD_CONTEXT`.

226
5.4.7 Tool Support for Context Programming

Hopefully by now you should feel comfortable enough with the context API to be able to perform basic CRUD operations within a context. However, if you forget how to perform a particular operation along the way, you can lean on the Web Dynpro code wizard to help jog your memory. As we learned in Chapter 4, we can access the code wizard within a controller method by clicking on the Web Dynpro Code Wizard button in the toolbar (see Figure 5.25).

![Figure 5.25 Accessing the Web Dynpro Code Wizard (Part 1)](image1)

After you initiate the wizard, you’ll be presented with the familiar Web Dynpro Statement Structure dialog box shown in Figure 5.26. Here, on the Context tab, you can use the input help for the Node/Attribute input field to select the target context node or attribute you want to access. Then, you can select from the available radio buttons to define the type of operation you want to perform. You can also select the As Table Operation checkbox to perform bulk operations.

![Figure 5.26 Accessing the Web Dynpro Code Wizard (Part 2)](image2)
As is the case with most code wizards, the generated code will most likely require some tweaking in order for everything to work correctly. Still, more often than not, this is much better than having to write the code from scratch.

5.5 Supply Functions

So far, our context programming examples have assumed that data is loaded into a context via regular controller methods (i.e., hook methods or instance methods). However, as we learned in Chapter 4, controllers also offer a specialized method type for this purpose: supply functions.

As the name suggests, supply functions are used to supply a context node with data on demand. As such, their implementations are relatively straightforward. In the upcoming sections, we'll look at how supply functions are used to streamline the context node initialization process.

5.5.1 Creating a Supply Function

Despite the use of the term function, supply functions are in fact controller methods. Therefore, they’re created in much the same way that regular controller methods are created. Here, we have two options for creating a supply function:

1. We can manually create a supply function by navigating to the METHODS tab of the Controller Editor and filling in the method details in the provided input table, as shown in Figure 5.27. Then, once the supply function is created, we can associate it with a context node in the Context Editor, as shown in Figure 5.28.
2. Alternatively, we can let the system automatically create the method for us by filling in the desired supply function name for the selected context node in the Context Editor screen, as shown in Figure 5.28. In this case, the supply function will be created silently behind the scenes if it doesn’t exist already.

![Figure 5.28 Creating a Supply Function (Part 2)](image)

### 5.5.2 Implementing Supply Functions

Regardless of the approach taken to create a supply function, we’ll end up with an empty method that defines two importing reference parameters:

- **NODE**
  
  This parameter refers to the node whose element collection is being filled by the supply function. Naturally, this parameter is defined using the interface type `IF_WD_CONTEXT_NODE`.

- **PARENT_ELEMENT**
  
  This parameter provides a reference to the parent element of the context node. It’s defined using the interface type `IF_WD_CONTEXT_ELEMENT`.

With these two parameters in hand, we can implement a supply function by filling in the element collection for the selected context node using methods of the `IF_WD_CONTEXT_NODE` interface. Listing 5.12 demonstrates how this works for a
supply function called SUPPLY_ORDER_ITEMS, which is used to fill in the ORDER_ ITEMS context node. As you can see, this implementation is mostly business as usual from a context programming perspective.

```
method SUPPLY_ORDER_ITEMS.
  "Method-Local Data Declarations:
  DATA: lt_order_items TYPE wd_this->Elements_order_items.
  FIELD-SYMBOLS:
    <lfs_order_item> LIKE LINE OF lt_order_items.

  "Add the sales order items to the node collection;
  "Note: This data would be normally obtained from
  "the model. We can use context attributes from the
  "PARENT_ELEMENT parameter as a key for performing
  "the lookup:
  APPEND INITIAL LINE TO lt_order_items
    ASSIGNING <lfs_order_item>.
    <lfs_order_item>-item_no = '12345'.
    <lfs_order_item>-product_no = '1234567890'.
    <lfs_order_item>-quantity = 10.

  node->bind_table(
    new_items = lt_order_items
    set_initial_elements = abap_true ).
endmethod.
```

Listing 5.12 A Supply Function Example

### 5.5.3 Understanding How Supply Functions Are Invoked

In general, supply functions are invoked automatically by the Web Dynpro framework to fill a context node whenever one of the following conditions is satisfied:

- The context node is initial or not yet filled.
- The context node has been invalidated at a previous step.

While the first condition makes intuitive sense, the second condition requires some additional description.

During the lifecycle of a context node, there may be times when the contents context node becomes dirty. This could be as a result of a change in the model layer,
upstream node changes, etc. Regardless of the root cause, we need to make sure the node gets refreshed so that data remains in sync. Here, we could manually clean up the node using context API methods, or we could defer the initialization to supply functions. In the latter case, we must invalidate the context node in order for the Web Dynpro framework to determine that it needs to be refreshed.

Context nodes can be invalidated in two ways:

- The Web Dynpro framework will automatically invalidate a singleton node whenever the lead selection in the parent node changes.
- You can also manually invalidate context nodes using the `INVALIDATE()` method provided with the `IF_WD_CONTEXT_NODE` interface.

Once a context node is invalidated, the Web Dynpro framework will take care of invoking the node’s corresponding supply function behind the scenes. We’ll examine how this works for singleton nodes in the next section.

### 5.5.4 Supply Functions and Singleton Nodes

One of the most common use cases for supply functions is to automate the synchronization of singleton child nodes. To understand how this works, let’s take a closer look at how the `ORDER_ITEMS` singleton node is filled using the `SUPPLY_ORDER_ITEMS()` supply function, as shown in Listing 5.12.

If you recall from Section 5.2.4, only a single instance of a singleton node will be created at runtime. This single node instance corresponds with the lead selection element of the parent node. So, in the case of our `SALES_ORDERS` context, we want to use the `SUPPLY_ORDER_ITEMS()` supply function to populate the `ORDER_ITEMS` node collection each time the lead selection changes in the parent `SALES_ORDERS` node.

Fortunately, the Web Dynpro framework will initiate the synchronization process automatically by invalidating the `ORDER_ITEMS` node each time the lead selection for the `SALES_ORDERS` node changes. Then, it will invoke the `SUPPLY_ORDER_ITEMS()` method, passing to it a reference to the `ORDER_ITEMS` singleton node and the lead selection element from the parent `SALES_ORDERS` node. The supply function can then use these references to fill in the `ORDER_ITEMS` node collection based on the selected `SALES_ORDERS` context element, as demonstrated in Listing 5.12.
5.6 Context Mapping and Data Binding: Up Close

Before we conclude our discussion on contexts, we should take a moment to revisit the concepts of context mapping and data binding in light of what we’ve learned in this chapter. That way, you’ll have a clearer picture of what’s going on behind the scenes.

5.6.1 How Context Mapping Works

As we observed in Section 5.1.2, the data within a WDA component flows down-stream from the model layer to the view layer as follows:

1. First, data is extracted from the model layer via a service call performed in a global controller method.
2. Then, the extracted data is loaded into the global controller’s context using the context API, as demonstrated in Section 5.4.
3. Once the data is loaded into the global controller context, it will be made visible to downstream views via context mapping relationships.
4. Finally, the data will become visible in UI elements through data binding relationships.

From a conceptual point of view, there’s an invisible line that separates steps one and two from steps three and four. This line represents the virtual barrier that exists between the controller and view layers. Here, the controller layer produces data by loading model data into a global controller context, and the view layer then consumes this data using declaratively defined mapping relationships. The question is: how does this implicit data transfer occur?

As we’ve seen, context mapping relationships are declaratively defined within a controller by dragging and dropping a context node from a used controller into the controller’s context. Once the relationship is defined, a copy of the mapped context node will be created within the referencing controller’s context (see Figure 5.29). As evidenced by the little arrow icon underneath it, this copied node effectively points to the node it’s mapping from. In other words, you can think of this copied node as an alias that can be used to address the mapped node at runtime.
5.6

If we think about the implied meaning of terms like pointer and alias, we can begin to grasp the nature of context mapping relationships from a runtime perspective. Though there are two copies of the mapped node at design time, there is in fact only one copy of the context node data at runtime: the one maintained within the used controller. Thus, any references to the copied node indirectly refer to the mapped node in the used controller. There are a couple of major benefits to this approach:

1. First of all, it improves performance by avoiding unnecessary memory copy steps between controllers.
2. Secondly, it further reinforces the principle of strict separation of producers and consumers from the MVC paradigm by ensuring that data is transferred between controller and view layers in a controlled and automated manner.

The great thing about all this from a developer’s perspective is that the Web Dynpro framework takes care of managing these references for us behind the scenes. Therefore, not a single line of ABAP code is required to shuffle the data back and forth between controllers.

5.6.2 How Data Binding Works

Once data makes its way into a view controller context, it can be made available to selected UI elements within the view layout by declaratively defining data binding relationships. As you can see in Figure 5.30, these data bindings formally associate a UI element property with a context node or context attribute. This
A bi-directional relationship provides the Web Dynpro framework with the meta-data it needs to determine how to map values back and forth between a UI element property and the corresponding context node or context attribute it’s bound to.

At runtime, data transfer is carried out automatically by the Web Dynpro framework in two distinct places:

- At the beginning of the request/response cycle, data input by the user is transported from the various UI elements to their corresponding context nodes and context attributes.
- At the end of the request/response cycle, data is copied from context nodes and context attributes into the UI elements during the client rendering phase so that it can be displayed on the screen.
Though all this is shielded from developers, rest assured that this data transfer does not occur by magic. Behind the scenes, the Web Dynpro framework is utilizing the context API to transfer data to and from the context. On the other side of this exchange, the framework queries/sets the properties of UI elements using their corresponding runtime classes. Here, for example, to copy data to or from an InputField UI element, the framework would utilize the standard runtime class CL_WD_INPUT_FIELD. This class provides getter and setter methods such as GET_VALUE() and SET_VALUE(), which allow the framework to access the value property of an InputField element programmatically. We’ll get a taste for some of this when we look at dynamic programming concepts in Chapter 11.

5.7 Summary

In this chapter, we explored the Web Dynpro context from several different angles. We began by analyzing the structure of contexts from both a conceptual and technical point of view. From there, we observed how these context hierarchies are declaratively defined using the Context Editor tool. Next, we surveyed the interfaces and methods that make up the context programming API, demonstrating how this API could be used to manipulate a context at runtime. Finally, we concluded our discussion by reviewing context mapping and data-binding concepts.

While context programming is certainly not one of the flashier topics related to WDA programming, it’s right up there among the most important ones. So, suffice it to say that we’ll see the concepts described in this chapter again and again as we progress through the rest of this book.

In the next chapter, we’ll shift gears and begin looking at UI development concepts. Specifically, we’ll take a closer look at windows and views and see how they’re used to arrange the UI layout within the view layer.
Index

A

ABAP
  Code Inspector, 673
  Debugger, 123
  Runtime Analysis, 673, 691
  unit, 673
ABAP Repository, 92
  package concept, 93
ABAP statements, SELECT-OPTIONS, 487
ABAP Workbench, 91
  Repository browser, 91
  screen layout, 91
Accessibility, 585
  accessibilityDescription property, 585
  tooltip property, 585
Actions, 140
  creating, 142
Active component Framework (ACF), 358
Active Server Pages (ASPs), 36
Adaptation, 566
  customization vs. configuration, 570
  layers, 566
  processing, 567
Adjustments, 605
  conflicts, 605
Adobe Flash Islands, 638
  data transfer, 640
  events, 641
  themes, 641
Adobe Flex, 638
AJAX, 40
ALV
  integration, 452
  SALV_WD_TABLE component, 453
  SAP List Viewer, 64
ALV configuration model, 455
Application configuration, 572, 574
  Application Configurator, 573
  assigning application configurations, 575
  assigning component configurations, 574
  objects, 572
Application parameters, 574, 583
  processing, 584
AS ABAP, 50
  architecture, 50
  trial download, 59
Assistance classes, 168
  creating, 168
  developing, 168
  usage, 170
  WD_ASSIST attribute, 171
Asynchronous JavaScript (AJAX), 40, 669
Authorizations, 663

B

BApis, 603
Best design practices, separation of concerns, 672
BreadCrumb Element, description, 367
BSPs, 36
  HTML, 37
  Scriptlet tags, 36
Business Address Services (BAS), 421
Business Server Pages (see BSPs)

C

Cascading Style Sheets (see CSS)
CGI, 34
  CGI, Perl script, 34
Change and Transport System (CTS), 353
Class
  browser, 532
  CL_SALV_WD_COLUMN, 456
  CL_SALV_WD_CONFIG_TABLE, 456
  CL_WD_COMPONENT_ASSISTANCE, 168
  CL_WD_CUSTOM_EVENT, 137
  CL_WD_DYNAMIC_TOOL, 513, 563
  CL_WD_FORMATTED_TEXT, 326
  CL_WD_INPUT_FIELD, 310, 467
  CL_WD_LAYOUT_DATA, 534
  CL_WD_NOTIFICATION_SERVICE, 349
  CL_WD_RUNTIME_SERVICES, 363
Index

Class (Cont.)
  CL_WD_TABLE, 382
  CL_WD_UIELEMENT_CONTAINER, 530
Classic Dynpro, example, 48
Common Gateway Interface (CGI), 34
Component, 422
Component configuration, 568, 570
    built-in properties, 572
    component configuration objects, 568
    component-defined properties, 571
Component controller, 74, 144
  hook methods, 147
Component interface, IWD_VALUE_HELP, 483
Component usages
  cloning, 539
  defining, 434
  groups, 541
Component-defined adaptation, 579
  data, 581
  disabling, 582
  primary attributes, 581
Componentization, 62
Composite design pattern, 251
Configuration, 567
  Configuration Editor, 568
Configuration controller, 580
Confirmation dialog boxes, 503
  creating, 503
Container layouts, 272
  FlowLayout, 273
  FormLayout, 280
  GridLayout, 278
  MatrixLayout, 276
  RowLayout, 275
Context attributes, 184
  defining, 195
  properties, 187
  runtime properties, 220
  static attributes vs. dynamic attributes, 220
Context change log, 225
  interface IF_WD_CONTEXT, 226
Context Editor, 197
Context mapping, 232
  across components, 441
  defining, 83
Context mapping, 232 (Cont.)
  overview, 232
Context nodes, 184
  cardinality, 185
  defining, 191
  interface nodes, 441
  lead selection concept, 188
  properties, 184
  singleton property, 189
Context programming
  accessing child nodes, 205
  accessing context elements, 208
  best practices, 207, 673
  creating new context elements, 212
  path-based access, 207
  removing elements from a node collection, 223
  setting the lead selection, 210
  sorting a node collection, 224
  supply functions, 228
  tool support, 227
  working with context attributes, 217
  working with node collections, 214
  working with the Context Change Log, 225
Contexts, 77
  attributes, 77
  Context Editor, 78
  data flow overview, 83
  elements, 78
  example, 77
  hierarchy organization, 77
  mapping, 81
  node cardinalities, 78
  nodes, 77
  overview, 77
  root, 78
  usage, 80
Controller contexts, 179
  at runtime, 183
  attributes, 182
  basic elements, 182
  data flow, 180
  defined, 179
  defining, 191
  nodes, 182
  programming concepts, 202
  root node, 182
Index

Controller contexts, 179 (Cont.)
  WD_CONTEXT attribute, 204
Controller events, 134
  configuration, 136
  event process flow, 134
  parameter passing, 138
  usage cases, 139
  usage example, 137
  usage restrictions, 140
Controller interface, 145
  relationship to component controller, 145
Controller methods
  example, 102
  syntax restrictions, 131
Controller types, 144
  Component controller, 144
Controller usages, 157
  accessing programmatically, 159
  creating, 158
  restrictions, 159
Controllers, 73
  component controller, 74
  contexts, 77
  Controller Editor, 75
  custom controllers, 74
  interface controller, 74
  usage relationships, 82
  view controllers, 75
  window controllers, 75
  with a visual interface, 75
  without a visual interface, 74
Cross-component context mapping
  external, 443
  simple, 441
Cross-component navigation scenarios, 446
CSS, 43, 680
Custom controllers, 74, 149
  creating, 149
  hook methods, 151
Customization, 567, 577
  Customization Editor, 569
  decorative UI elements, 577
  disabling, 578

D
  Data binding, 80, 232
    overview, 80, 233
  Data security
    application, 665
    context, 666
    file upload, 666
  Declarative programming, 41
  Delta rendering, 681
    functionality, 682
  Document Object Model (DOM), 690
  DOM analysis, 690
  Drag and drop, 409
    conceptual overview, 409
    example, 415
    supported SAP NetWeaver versions, 409
    supported UI element types, 409
  DropDownByIndex element, index binding, 340
  DropDownByKey element, key binding, 340
  Dynamic component usages, 539
    creating, 539
  Dynamic context manipulation
    adding context attributes, 527
    adding context nodes, 522
    API methods, 520
    best practices, 528
  Dynamic programming, 519
    calling methods of a used component, 546
    registering event handler methods, 547
    view manipulation, 528
  Dynamic view manipulation
    declarative alternatives, 538
    manipulating UI elements, 536
    navigating the class hierarchy, 531
    adding new UI elements, 533
  Dynamically embedding interface views, 542
  Dynpro component interfaces, implementation, 433
  Dynpro technology, 47

E
  Enhancement Framework, 593
  Enhancement implementations, 595
Index

Enhancement spots, 594
Enhancing Web Dynpro components, 597
  controllers, 600
  views, 597, 598
  windows, 602
Error pages, 661
Explanation element, demonstration, 498
Explicit enhancements, 594
  in Web Dynpro, 603
External aliases, 662

Faceless components, 449
FileDownload element, example, 362
FileUpload element, example, 361
Floorplan Manager, 430, 607
application configuration, 617
application creation, 615
Application Creation Tool, 615
application hierarchy browser, 617
architecture, 608, 609
Configuration Editor, 618, 620
configuration objects, 614
dialog boxes, 624
genetic user Graphical User Interface Building Blocks (GUIBBs), 622
Generic User Interface Building Blocks (GUIBBs), 613
Guided Activity Floorplan (GAF), 611
Object Instance Floorplan (OIF), 611
Overview Page Floorplan (OVP), 612
Quick Activity Floorplan (QAF), 612
toolbar buttons, 625
types of floorplans, 610
User Interface Building Blocks (UIBBs), 615, 621
wiring, 623
Freely programmed input helps, 482
integration, 485
IWD_VALUE_HELP component interface, 483
Fully Qualified Domain Names (FQDN), 657

H
Help texts, 498
Hook methods, 128
HTTP, 30
cookies, 38
DELETE method, 32
GET method, 32
HEAD method, 32
message format, 32
POST method, 32
PUT method, 32
request flow, 30
request method, 32
server roundtrip, 40
session management, 37
stateless protocol, 32
HTTP compression, 683
HTTP trace tools
  HTTPWatch, 699
  WireShark, 699
HTTPS, 656
Hypertext, 33
Hypertext Markup Language (HTML), 33
Hypertext Transport Protocol (see HTTP)

ICF, 52
common service nodes, 56
IF_HTTP_EXTENSION interface, 55
IF_HTTP_REQUEST interface, 58
IF_HTTP_RESPONSE interface, 58
monitoring tools, 695
process flow of an HTTP request, 52
service nodes, 53
ICM, 50
Image element
integration concepts, 350
integration with MIME Repository, 351
Imperative programming, 41
Implicit enhancements, 594
Inbound plugs, 282
  event handler methods, 284
Input helps, 467
InputField element
  integrating classic F1 help, 501
  suggested values, 497
Instance methods, 130
InteractiveForm element, example, 364
Interface
  IF_MR_API, 398
  IF_SALV_WD_COLUMN_SETTINGS, 456
  IF_SALV_WD_EXPORT_SETTINGS, 456
  IF_SALV_WD_FIELD_SETTINGS, 456
  IF_SALV_WD_FUNCTION_SETTINGS, 456
  IF_SALV_WD_STD_FUNCTIONS, 456
  IF_SALV_WD_TABLE_SETTINGS, 456
  IF_WD_ACTION, 166
  IF_WD_APPLICATION, 166
  IF_WD_COMPONENT, 166
  IF_WD_COMPONENT_USAGE, 437, 539, 540, 546
  IF_WD_COMPONENT_USAGE_GROUP, 541
  IF_WD_CONTEXT, 226
  IF_WD_CONTEXT_ELEMENT, 203, 217
  IF_WD_CONTEXT_NODE, 203, 225
  IF_WD_CONTEXT_NODE_INFO, 343, 520
  IF_WD_CONTROLLER, 166, 425, 506
  IF_WD_MESSAGE_AREA, 515
  IF_WD_MESSAGE_MANAGER, 166, 506
  IF_WD_OVS, 477
  IF_WD_VALUE_HELP_LISTENER, 484
  IF_WD_VIEW, 166, 529
  IF_WD_VIEW_CONTROLLER, 544
  IF_WD_VIEW_ELEMENT, 531
  IF_WD_WINDOW, 166, 254, 448
  IF_WD_WINDOW_MANAGER, 166, 255, 446, 503
  IWCI_SALV_WD_TABLE, 456
Interface controller, 66, 74, 423
  context nodes, 425
  corresponding ABAP Objects interface, 427
  defining, 424
  editor view, 426
  events, 425
  methods, 425
Interface nodes, 441
Interface views, 247, 423
  configuration, 428
  defined, 247
Interface views, 247, 423 (Cont.)
  embedding in windows, 446
  link to WDA applications, 249
  navigation plugs, 428
  opening as popup windows, 448
Internationalization, 588
  dynamic texts, 590
  long texts, 591
  Online Text Repository (OTR), 589
  static texts, 589
Internet Communication Framework (see ICF)
Internet Communication Manager (see ICM)
Internet Graphics Service (IGS), 348
Internet Transaction Server (ITS), 49
Islands framework, 634
  data transfer, 635
  events, 637
  FlashIsland element, 634, 638, 639
  GACDataSource control, 636
  GACEvent control, 637
  GACEventParameter control, 637
  GACProperty control, 636
  SilverlightIsland element, 634, 642, 643
  views, 634
ITS
  basic architecture, 49
  limitations, 50
  positioning, 49

J
Java Runtime Environment (JRE), 348, 358
Java Servlets, 35
JavaServer pages (JSPs), 36

K
Knowledge provider (KPro), 359

L
Layout elements
  Accordion, 267
  Group, 269
  MultiPane, 270
  PageHeader, 265
Index

Layout elements (Cont.)
  Panel, 269
  RowRepeater, 270
  TabStrip, 267
  TransparentContainer, 263
  Tray, 269
Logoff pages, 660

M

Memory Inspector, 694
  Transaction S_MEMORY_INSPECTOR, 694
Menus
  MenuActionItem, 334
  MenuCheckBox, 334
  MenuRadioButton, 334
  MenuSeparator, 334
Message Manager, 506
  API methods, 507
  creating messages, 506
Messages, 505
  automatic message generation, 513
  issuing from assistance classes, 511
  linking with UI elements, 508
Microsoft Office, 360
Microsoft Silverlight, 642
Microsoft Silverlight Islands, 642
  data transfer, 643
  events, 644
MIME Repository, 352
  description, 353
Model View Controller (see MVC)
Modifications, 594
Multi-component designs, guidelines for usage, 462
MVC, 37
  architecture diagram, 39
  design pattern, 38

N

Navigation
  firing outbound plugs, 287
  plugs, 281
Navigation links, 71, 281
  creating, 284

Navigation links, 71, 281 (Cont.)
  Graphical Editor Tool, 286
  illustration, 71
  inbound plugs, 71
  outbound plugs, 71
  process flow, 72
Navigation plugs, 281
  inbound plugs, 282
  outbound plugs, 282
  parameterization, 283
  window plugs, 289
Nesting analysis, 689
Network element, JNet Applet, 349

O

Online Text Repository (OTR), 499
Outbound plugs, 282
OVS value helps, 477
  defining, 479
  event Lifecycle, 477
  overview, 477
  WDR_OVS component, 477
OVS, object value selection, 477

P

Package concept, 93
  application, 94
  development packages, 94
  main packages, 94
  SAP application hierarchy, 96
  structure packages, 94
Page Builder, 626
  CHIP imports, 629
  CHIP exports, 629
  CHIPS, 628
  Configuration Editor, 628
  pages, 627
  side panels, 630
Parameterization, 583
Performance analysis, Transaction ST05, 694
Performance monitor, 687
Performance tuning, 669
  approach, 670
  best practices, 672
Index

Performance tuning, 669 (Cont.)
  frontend delegation, 679
performance analysis tools, 684
  resource utilization, 674
Personal value lists, 475
  maintenance, 475
Personalization, 567, 576
  analysis tool, 579
  disabling, 578
Polymorphism, 551
Popup windows, 448
  event coordination, 256
  lifecycle, 256
  opening, 253
  usage recommendations, 259

R

Radio buttons, 345
Recursion, 197
Recursion nodes, 197
  at runtime, 201
    defining, 199
Resume plugs, defining, 301
Rich Internet Applications (RIA), 634
RoadMap element
  configuration, 404
    demonstration, 404

S

SAP application hierarchy, 96
SAP GUI, 47
  DIAG protocol, 48
    for HTML, 49
SAP Interactive Forms by Adobe, 360, 364, 644
SAP List Viewer
  ALV, 64, 452
    event handling concepts, 458
      integration, 452
  SALV_WD_TABLE component, 453
SAP NetWeaver Developer Studio, NWDS, 46
SAP NetWeaver Portal, 647
  events, 650
  iViews, 649
SAP NetWeaver Portal, 647 (Cont.)
  navigation, 650
  system objects, 647
Search helps
  ABAP dictionary, 468
    attributes, 471
    collective search helps, 470
    elementary search helps, 470
  features in WDA applications, 474
    integration with Web Dynpro, 473
    introduction, 468
    personal value list, 475
Select options, 487
  description, 487
    integration with Web Dynpro, 490
    range table, 488
  WD_SELECT_OPTIONS_20 component, 491
  WDR_SELECT_OPTIONS component, 490
Server pages, 36
  tag libraries, 37
Service calls, 176
Service classes, 167
  CL_WD_COMPONENT_ASSISTANCE, 167
  CL_WD_CONTEXT_SERVICES, 167
  CL_WD_DYNAMIC_TOOL, 167
  CL_WD_FORMATTED_TEXT, 326
  CL_WD_RUNTIME_SERVICES, 167, 363
  CL_WD_TRACE_TOOL, 167
  CL_WD_UTILITIES, 167
  CL_WEB_DYNPRO, 167
Software components, 62
  characteristics, 63
  cohesion, 63
  component interfaces, 63
  encapsulation, 64
    Web Dynpro, 64
SSL, 656
Style sheets, 586
Supply functions, 228
  creating, 228
  process flow, 230
  relationship with singleton nodes, 231
Suspend plugs, defining, 301
Switch Framework, 597
System logon, 654
  identity providers (IdP), 659
Index

System logon, 654 (Cont.)
screens, 654
SSO tickets, 656, 657

T

Table element
configuration, 374
defining selection behavior, 378
demonstration, 374
sorting and filtering, 381	
table popins, 386
Table popins, 386
configuration, 387
Themes, 587
custom themes, 588
ThresholdSlider element, example, 356
Tree element
demonstration, 392
event handling, 402

U

UI element hierarchy, RootUIElementContainer, 243
UI elements
Accordion, 267
AcfExecute, 358, 359
AcfUpDownload, 359
action category, 312
aggregations, 309
basic concepts, 308
basic elements, 307
BIAplicationFrame, 359
BreadCrumb, 367
BusinessGraphics, 348
Button, 330
ButtonChoice, 330
ButtonRow, 335
Caption, 314
categories, 260, 312
Checkbox, 337
CheckboxGroup, 337
Client Table (CTable), 373
Complex Category, 312
DateNavigator, 369

UI elements (Cont.)
DragSourceInfo, 412, 413
DropDownByIndex, 338, 340
DropDownByKey, 338, 340
DropTarget, 414
DropTargetInfo, 413
Events, 308
Explanation, 314, 498
explanation property, 500
FileDownload, 359
FileUpload, 359
FormattedTextEdit, 314, 328
FormattedTextView, 314, 325
Gantt, 348
GeoMap, 348
graphic category, 312
Group, 269
IconButton, 330
IFrame, 359, 360
Image, 349, 350
InputField, 314, 316
integration category, 313
interactive elements, 260
InteractiveForm, 360, 364
introduction, 307
ItemListBox, 338, 410
label, 314, 318
layout category, 312
layout elements, 260, 262
legend, 370
linkage to actions, 331
LinkChoice, 330
LinkToAction, 330, 457
LinkToURL, 330
MenuBar, 333
menus, 332, 333
MessageArea, 515
MultiPane, 270
Network, 349
Overview, 68
PageHeader, 265
Panel, 269
PhaseIndicator, 368
ProgressIndicator, 349
properties, 69, 261, 308
RadioButton, 339

782
UI elements (Cont.)
RadioButtonGroupByIndex, 339, 345
RadioButtonGroupByKey, 339, 345
RoadMap, 368, 404
RowRepeater, 270
runtime class hierarchy, 531
runtime classes, 310
SectionHeader, 314
selection category, 312
Shuttle, 370
Table, 371
TablePopin, 388
TabStrip, 267
text category, 312
TextEdit, 314, 323
TextView, 315, 323
ThresholdSlider, 350, 355
TimedTrigger, 331, 349
ToggleButton, 339
ToggleLink, 339
ToolBar, 335
TransparentContainer, 263
Tray, 269
Tree, 373, 392
TreeItemType, 392
TreeNodeType, 392
TriStateCheckBox, 339
ValueComparison, 350
ViewContainerUIElement, 241
Unified Rendering Light Speed (see URLS)
Uniform Resource Locator (URL), 30
URLS, 45, 680
delta rendering, 681
enabling, 680
Used components
instantiating, 436
listening for events, 439
Views, 259
defined, 259
definition, 67
empty views, 245
lifespan control, 676
UI element hierarchy, 68
WDDOMODIFYVIEW0 hook method, 529
WDA, 29
Adobe Flex, 45
defined, 29
foundations, 30
HTML5, 45
model-driven architecture, 44
multi channel access, 45
programming model, 41
system requirements, 59
WDA applications, 85, 86
creating, 89
definition, 86
ICF service nodes, 86
properties, 87
structural overview, 61
testing, 121
URL format, 86
WDA components
architectural overview, 65
component interface, 66
controllers, 73
creating, 97
interface controller, 66
relationship to MVC, 66
views, 67
windows, 69
WDA programming model, 41
illustrated, 43
Web Application Server, 35
Web Dynpro ABAP (see WDA)
Web Dynpro Code Wizard, 172
calling interface controller methods, 439
creating messages, 511
generating popup windows, 449
instantiating used components, 438
screen design time conversion, 173

V

View controllers, 152
actions, 140
hook methods, 153
View Designer, 261
View Editor, 261
Container Form Wizard, 321
Index

Web Dynpro component interface
  defined, 422, 431
  interface controller, 423
  interface views, 423
  standalone interface, 430
Web Dynpro controllers
  attributes, 132
  element overview, 127
  event handler methods, 132
  events, 134
  methods, 128
  supply function methods, 132
Web Dynpro Debugger, 124
Web Dynpro Explorer, 42, 67, 68, 91
  accessing, 91
  Controller Editor, 127
  Refactoring Assistant, 177
  tool support, 171
Web Dynpro framework, 42
  MVC, 43
Web Dynpro Islands, 683
Web Dynpro metamodel, 42
Web Dynpro Notification Service, 349
Web Dynpro phase model, 160
  actions and events, 163
  illustrated, 161
Web Dynpro runtime API, 165
Web Dynpro runtime environment, 43, 57
  CL_WDR_MAIN_TASK, 57
Web Dynpro trace tool, 685
Web Dynpro, component overview, 65
Web programming model, 33
  CGI, 34
  Java Servlets, 35
  server pages, 36
  static HTML, 33
  web frameworks, 37
Web Services, 651
Whitelist infrastructure, 667
Window controllers, 154
  hook methods, 155
Window plugs, 289
  closing WDA applications with exit plugs, 297
  conditional view selection, 292
  defining inbound plugs, 291
  defining outbound plugs, 290
  inbound plug types, 291
  outbound plug types, 290
  suspend and resume calls, 299
Windows, 238
  as view containers, 238
  default view property, 238
  defined, 69
  embedding views, 240
  interface views, 247
  popup, 70, 253
  relationship with views, 238
  Window Editor, 70, 238
Workload Monitor, 694
  Transaction STAD, 694
World Wide Web, 30