The Structural Concepts of Web Dynpro Components

Release 645

SAP®
Copyright

© Copyright 2006 SAP AG. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP AG. The information contained herein may be changed without prior notice.

Some software products marketed by SAP AG and its distributors contain proprietary software components of other software vendors.

Microsoft, Windows, Outlook, and PowerPoint are registered trademarks of Microsoft Corporation.

IBM, DB2, DB2 Universal Database, OS/2, Parallel Sysplex, MVS/ESA, AIX, S/390, AS/400, OS/390, OS/400, iSeries, pSeries, xSeries, zSeries, z/OS, AFP, Intelligent Miner, WebSphere, Netfinity, Tivoli, and Informix are trademarks or registered trademarks of IBM Corporation in the United States and/or other countries.

Oracle is a registered trademark of Oracle Corporation.

UNIX, X/Open, OSF/1, and Motif are registered trademarks of the Open Group.

Citrix, ICA, Program Neighborhood, MetaFrame, WinFrame, VideoFrame, and MultiWin are trademarks or registered trademarks of Citrix Systems, Inc.

HTML, XML, XHTML and W3C are trademarks or registered trademarks of W3C®, World Wide Web Consortium, Massachusetts Institute of Technology.

Java is a registered trademark of Sun Microsystems, Inc.

JavaScript is a registered trademark of Sun Microsystems, Inc., used under license for technology invented and implemented by Netscape.

MaxDB is a trademark of MySQL AB, Sweden.

SAP, R/3, mySAP, mySAP.com, xApps, xApp, SAP NetWeaver, and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world. All other product and service names mentioned are the trademarks of their respective companies. Data contained in this document serves informational purposes only. National product specifications may vary.

These materials are subject to change without notice. These materials are provided by SAP AG and its affiliated companies ("SAP Group") for informational purposes only, without representation or warranty of any kind, and SAP Group shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP Group products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.
**Icons in Body Text**

<table>
<thead>
<tr>
<th>Icon</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>🚸</td>
<td>Caution</td>
</tr>
<tr>
<td>💡</td>
<td>Example</td>
</tr>
<tr>
<td>📄</td>
<td>Note</td>
</tr>
<tr>
<td>📚</td>
<td>Recommendation</td>
</tr>
<tr>
<td>📊</td>
<td>Syntax</td>
</tr>
</tbody>
</table>

Additional icons are used in SAP Library documentation to help you identify different types of information at a glance. For more information, see Help on Help → General Information Classes and Information Classes for Business Information Warehouse on the first page of any version of SAP Library.

**Typographic Conventions**

<table>
<thead>
<tr>
<th>Type Style</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Example text</em></td>
<td>Words or characters quoted from the screen. These include field names, screen titles, pushbuttons labels, menu names, menu paths, and menu options. Cross-references to other documentation.</td>
</tr>
<tr>
<td><em>Example text</em></td>
<td>Emphasized words or phrases in body text, graphic titles, and table titles.</td>
</tr>
<tr>
<td>EXAMPLE TEXT</td>
<td>Technical names of system objects. These include report names, program names, transaction codes, table names, and key concepts of a programming language when they are surrounded by body text, for example, SELECT and INCLUDE.</td>
</tr>
<tr>
<td><em>Example text</em></td>
<td>Output on the screen. This includes file and directory names and their paths, messages, names of variables and parameters, source text, and names of installation, upgrade and database tools.</td>
</tr>
<tr>
<td><em>Example text</em></td>
<td>Exact user entry. These are words or characters that you enter in the system exactly as they appear in the documentation.</td>
</tr>
<tr>
<td>&lt;<em>Example text</em>&gt;</td>
<td>Variable user entry. Angle brackets indicate that you replace these words and characters with appropriate entries to make entries in the system.</td>
</tr>
<tr>
<td>EXAMPLE TEXT</td>
<td>Keys on the keyboard, for example, F2 or ENTER.</td>
</tr>
</tbody>
</table>
# Table of Contents

Copyright ............................................................................................................................... 2
Icons in Body Text ................................................................................................................ 3
Typographic Conventions .................................................................................................... 3
Table of Contents .................................................................................................................. 4
Table of Figures ..................................................................................................................... 6

## Introduction

What is Web Dynpro? ............................................................................................................ 7
What this document does not cover .................................................................................... 7

## Summary of the Structural Concepts of a Web Dynpro Component

The Metamodel Concept ...................................................................................................... 8
Execution of Web Dynpro Applications .............................................................................. 8
Web Dynpro implementation differences between Java and ABAP ................................. 9
Graphical Development Tools ............................................................................................ 9
SAP’s implementation of the MVC paradigm ...................................................................... 10

## Web Dynpro Application

Definition ............................................................................................................................... 12
Lifespan of a Web Dynpro Application ............................................................................... 12

## Web Dynpro Component

Definition ............................................................................................................................... 13
Lifespan of a Web Dynpro Component ............................................................................... 14
The faceless Component ..................................................................................................... 14

## A quick overview ............................................................................................................ 16

## Web Dynpro Controllers

Definition ............................................................................................................................... 17
Controllers having no visual interface ................................................................................ 17
  Structure of custom controllers ....................................................................................... 18
Controllers having a visual interface ................................................................................ 20
  Structure of view controllers .......................................................................................... 20
Specific controller types ..................................................................................................... 24
  Component Controller .................................................................................................... 24
  Custom Controllers ....................................................................................................... 25
  Interface Controller ...................................................................................................... 25
## Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Web Dynpro code generation process</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>A Web Dynpro Application</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>The composition of a Web Dynpro Component</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>Displaying data held in the component controller's context using a combination of context mapping and UI element binding</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>Web Dynpro Controllers with no visual interface</td>
<td>17</td>
</tr>
<tr>
<td>6</td>
<td>Web Dynpro Controllers that have no visual interface</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>Web Dynpro Controllers that have a visual interface</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>Structure of a Web Dynpro View Controller</td>
<td>21</td>
</tr>
<tr>
<td>9</td>
<td>A client side event causes a server side action event handler to be executed</td>
<td>23</td>
</tr>
<tr>
<td>10</td>
<td>A Web Dynpro Window</td>
<td>30</td>
</tr>
<tr>
<td>11</td>
<td>Design time appearance of views embedded within Window</td>
<td>32</td>
</tr>
<tr>
<td>12</td>
<td>The first view assembly is rendered using the default views</td>
<td>33</td>
</tr>
<tr>
<td>13</td>
<td>The next view assembly is rendered using the default views</td>
<td>33</td>
</tr>
<tr>
<td>14</td>
<td>The context is at the heart of every Web Dynpro controller</td>
<td>36</td>
</tr>
<tr>
<td>15</td>
<td>A simple context structure</td>
<td>38</td>
</tr>
<tr>
<td>16</td>
<td>Context Nodes with a maximum cardinality of 1</td>
<td>39</td>
</tr>
<tr>
<td>17</td>
<td>Context Nodes with a maximum cardinality of n</td>
<td>40</td>
</tr>
<tr>
<td>18</td>
<td>A dependent child node</td>
<td>41</td>
</tr>
<tr>
<td>19</td>
<td>Multiple instances of a child node</td>
<td>42</td>
</tr>
<tr>
<td>20</td>
<td>A single instance of a child node</td>
<td>43</td>
</tr>
<tr>
<td>21</td>
<td>A simple example of a recursion node</td>
<td>44</td>
</tr>
<tr>
<td>22</td>
<td>A mapping relationship between a view controller and the component controller</td>
<td>47</td>
</tr>
<tr>
<td>23</td>
<td>UI element properties can be manipulated via the context</td>
<td>48</td>
</tr>
<tr>
<td>24</td>
<td>A window controller allows for dynamic calculation of the start view(s)</td>
<td>50</td>
</tr>
</tbody>
</table>
**Introduction**

**What is Web Dynpro?**

Web Dynpro is the SAP NetWeaver programming model for user interfaces (UIs). It provides a programming framework within which you can achieve:

- Clear separation of business logic and display logic
- An application specification that is both:
  - Client neutral and
  - Backend neutral

The Web Dynpro framework was developed because SAP needed a development framework for web based applications that satisfied all the requirements demanded by its current application software.

As an initial starting point, the well established Model View Controller (MVC) design paradigm was used to provide the architectural foundation for Web Dynpro. However, it became apparent that the concepts of MVC would have to be implemented in an SAP specific manner, and include certain extensions in order to create a framework that met all of SAP’s requirements.

This document will lay out both the structural concepts found within a Web Dynpro Component, and how they are have been derived from the core principles found in the MVC design paradigm.

**What this document does not cover**

This document has not been written to describe how a large Web Dynpro application should be structured. Instead, it focuses on how one of the reusable building blocks (a Web Dynpro Component) from which an application is built, is structured.

Therefore, the level of technical detail contained within this document sits in between that found in the high level architectural documents, and the low level coding features documents.
Summary of the Structural Concepts of a Web Dynpro Component

The Metamodel Concept

Since SAP uses both ABAP and Java as languages for the delivery of its application software, any development framework used by SAP must be able to accommodate both the requirements and the idiosyncrasies of these languages. It made little sense to have one design methodology for ABAP based applications and another for Java; therefore, a common structural concept was developed to lie at the heart of all Web Dynpro development. This common structural foundation is known as the “Web Dynpro Metamodel”, and acts a language neutral specification for both the visual appearance and development structure of a Web Dynpro program.

Using the Web Dynpro tools, you are able to design the main structural and visual parts of a business application in a manner that is largely independent of the implementation language. Much of this design takes place using declarative programming techniques. At design time therefore, you may declare:

- The data structures used to hold business data at runtime
- UI element definitions and properties
- The aggregation of UI elements into views
- The aggregation of views into view sets and windows
- Events that perform business tasks in response to user actions
- Navigation links between views in a window
- Reusable units of functionality known as components
- The use of persistent backend data and functionality via Web Services, Enterprise Java Beans or ABAP Remote Function Calls.

Once the metamodel has been defined, a significant proportion of the application coding can be generated by the Web Dynpro toolset. What remains for the application developer to do, is write the coding that responds to user interaction from the front end, and subsequently interacts with some backend system.

Figure 1 shows that the coding generated by the Web Dynpro toolset acts as a container for the subsequently developed custom code. What ever custom code you require must be placed into specific locations known as “hook methods”. The hook methods act as a predefined entry points into your custom code from the Web Dynpro metamodel coding.

---

1 Direct access to Enterprise Java Bean objects via the Remote Method Invocation (RMI) protocol is only possible using Web Dynpro for Java. If such an EJB is to be accessed from a Web Dynpro for ABAP program, it must first be wrapped as a Web Service.
Whatever coding you add into these hook methods will always be preserved, even if the Web Dynpro toolset needs to regenerate the entire application.

There are certain parts of the application metamodel (defined at design time) that can be changed at runtime. For example, if you have defined a view to contain a particular set of UI elements; then this set can be dynamically extended (or even deleted) at runtime if required. In this case, dynamic UI adaptation is achieved by writing your application in such a way that it contains the code necessary to interact with Web Dynpro’s runtime framework. It is not the purpose of this document to describe the details of this coding; however, it is important to understand that such possibilities exist.

**Execution of Web Dynpro Applications**

In order to execute the unit of coding represented by the “Deployable App” in Figure 1 above, a server side environment known as the Web Dynpro framework is needed. Without this runtime environment, it is not possible to execute a Web Dynpro application.

The Web Dynpro framework is supplied as an integral part of a NetWeaver 04s ABAP system, or as a deployed service in a NetWeaver 04s Java Server.

**Web Dynpro implementation differences between Java and ABAP**

Due to the fact that Java and ABAP are very different languages, there will be certain inevitable differences between Web Dynpro applications implemented in either of these languages: where these differences are significant (i.e. where structural or usage differences exist), a comment will be made. If no comment is made, then it can be assumed that the implementation structures are identical in both environments.

**Graphical Development Tools**

Since a large part of the Web Dynpro design process uses declarative programming techniques, this type of programming is best supported using graphical tools. Both the SAP NetWeaver Developer Studio (NWDS) and the ABAP Development Workbench contain a range of graphical Web Dynpro tools that allow you to define the:

- Data flow between the front end and back end
- Layout of the user interface
- Properties of user interface elements
- Navigation from one view to another

**Separation of Presentation and Business Logic**

Much literature has already been devoted to the importance of separating data processing from data presentation. However, this principle is based on a much more fundamental one in which those parts of the program that generate data are separated from those parts of the program that consume data. Whilst not explicitly stated in the original MVC design, this is one of its inherent, underlying principles.

---

2 The Web Dynpro framework service deployed on the SAP NetWeaver Java Server cannot function correctly without other deployed SAP services such as the Java Dictionary, WebServices, and the JCo service. Therefore, Web Dynpro applications are only supported for execution on an SAP Java Server and not a third party Java Server such as JBoss or WebSphere.

3 For instance, a Java Virtual Machine uses a thread-based architecture, whereas ABAP’s runtime environment is process based.

4 This point is mainly relevant for Web Dynpro for Java, since a Web Dynpro for ABAP program is typically executed in the same system in which the “backend” data is present.
Within a Web Dynpro component, SAP has implemented the principle of separating data generation from data consumption in a variety of situations, and on a variety of scales. As a result, a Web Dynpro application is structured in such a way there is much more than just a simple separation of presentation logic from business logic.

On the largest scale, a Web Dynpro application as a whole can be regarded as a consumer of data supplied by the back end system; correspondingly, the backend system as a whole can be considered a supplier of business data and logic.\(^5\)

In this situation, the following options are currently available for connecting Web Dynpro applications to back end systems:

- Calling a remote enabled ABAP function module (E.G. a BAPI) in an SAP system\(^6\)
- Calling an Enterprise Service Model
- Calling a Web service using the SOAP protocol
- Calling an Enterprise Java Bean\(^7\) through the Remote Method Invocation (RMI) protocol

**SAP’s implementation of the MVC paradigm**

All Web Dynpro applications are built using SAP’s implementation of the Model View Controller (MVC) design paradigm. As has already been stated, in its most general terms, this paradigm separates those parts of a program responsible for generating data, from those parts of the program that consume the generated data.

In reality however, you will often find that this division is too simplistic, because certain parts of a program will need to act as both generators and consumers of data. This extra complication does not negate the usefulness of either the data generator/consumer separation concept, or the MVC paradigm; rather it just means that procedures must be put in place that allow for the existence of this duality during implementation.

Within the Web Dynpro toolset, the principle of separating data generation from data consumption has been implemented in a variety of situations, using a variety of techniques. Consequently, some units of a Web Dynpro component are permitted to act only as consumers of data, whereas other units are permitted to act as both generators and consumers of data. In spite of the various technical implementations, the principle of separating data generators from data consumers remains the common foundation. The basic elements of a Web Dynpro application are the following:

- **The model [Page 29]**
  A model object encapsulates the interface to some back end system. Its purpose is to act as a proxy that detaches the Web Dynpro application from the data and functionality found in the remote system.\(^8\) In other words, the Web Dynpro application need not concern itself with the specific communication technology required to interact with the back end system.

- **The view [Page 14]**
  A view is used to define a client neutral visualisation of the business data.

- **The controller [Page 16]**
  In the original MVC paradigm, a controller was responsible for managing the interaction of view(s) and model(s), formatting the model data to be displayed in a view, and calculating which view (or views) was (were) to be displayed next. However, in Web

---

\(^5\) This statement is particularly true for Web Dynpro for Java applications.

\(^6\) This technical implementation of this interface differs depending on whether Web Dynpro for Java or Web Dynpro for ABAP is being used.

\(^7\) The direct use of RMI is only possible using Web Dynpro for Java.

\(^8\) The use of model objects is possible, though somewhat unnecessary in an ABAP environment. However, model objects are used extensively in the Java environment.
Dynpro this concept has been changed so that there are now – broadly speaking – two categories of controller; the differences between which will be described later in this document.
**Web Dynpro Application**

**Definition**

A Web Dynpro application defines a URL that acts as the client’s entry point into the functionality contained within a Web Dynpro Component. That is, a URL is created which points to one of the visual interfaces supplied by a Web Dynpro Component. For those readers familiar with the ABAP coding environment, the following analogy is useful. In the same way that an SAP transaction code is an entry point into a particular screen of a particular ABAP Module Pool, so a Web Dynpro Application is a URL entry point into a particular visual interface available from a Web Dynpro Component.

![Web Dynpro Application Diagram]

**Figure 2: A Web Dynpro Application**

The Web Dynpro component is the unit of functionality that acts as the application’s entry point. Any component\(^9\) can act as an application entry point, and which ever component has been nominated for this task is known as the root component. The root component should be thought of as being responsible for controlling all further processing that takes place within the application.

A Web Dynpro application *cannot* exist without there first being at least one Web Dynpro component. In Figure 2 above, the application is shown to contain three components and a model object.

**Lifespan of a Web Dynpro Application**

The lifespan of a Web Dynpro application is determined by, and equal to, the lifespan of the application’s root component.

\(^9\) Any component that is, that possesses a visual interface.
Web Dynpro Component

Definition

One of the main advantages of the original MVC design was its focus on the reusability of models, views, and controllers as individual coding units. However, Web Dynpro is focused on more than just the technical reuse of coding entities. Instead, Web Dynpro has been designed to be the foundation for implementing business solutions. Therefore, one of the key elements in its design was the need to provide a reusable unit of code that corresponded to an atomic step within a business process, rather than trying to build business solutions from reusable units of low level code that, in themselves, were not directly related to the business process.

In other words, the Web Dynpro component is a reusable unit of code at the business process level, rather than the technical coding level.

The resulting change in the nature of code reuse produces a shift in the developer’s focus of attention during coding. No longer are they concerned so much with the reuse of technical coding units; instead, the design of a Web Dynpro component focuses on the reuse of atomic units of business processing.

A component can be thought of as a set of controllers, views, and model usage declarations that have been aggregated for the specific purpose of reuse.

Always think of the Web Dynpro component as both:

- Your unit of development and
- Your unit of reuse

A component is therefore, the minimum unit of coding required when developing an executable Web Dynpro application [Page 12].

Figure 3: The composition of a Web Dynpro Component
As can be seen in Figure 3, a Web Dynpro component is not a single coding entity; rather, it is an aggregation of various controllers, views, windows, and model usages which, when taken together, form the reusable unit of business processing known as a component.

There are several key features that must be clearly understood concerning the structure of a Web Dynpro component:

1. It contains coding entities that are hidden from the outside world, and it also contains coding entities that are visible to the outside world. These two types of coding entity are divided from each other by the horizontal dotted line across the top of Figure 3.

2. It contains coding entities that handle the visual representation of data, and coding entities that handle the processing of data. These two types of coding entity are divided from each other by the vertical dotted line down the middle of Figure 3.

3. Within the scope of a component, the various controllers are designed as independent, yet interrelated programs. Therefore, if one controller wishes to gain access to the data and functionality within another controller (in the same component), a usage declaration must first be made: hence the “Usage Declaration” arrows between the various controllers in Figure 3.

4. Model objects are defined outside the scope of any particular Web Dynpro component. Once a model object has been created, any component can access its functionality by means of a usage declaration.

5. Web Dynpro components can use the data and functionality found within other Web Dynpro components. In order to achieve this, a usage declaration must be made in order to establish a “Parent-Child” relationship between the two components.

**Lifespan of a Web Dynpro Component**

There are two distinct lifespan scenarios for Web Dynpro components:

1. **Lifespan of the application’s root component**
   The component chosen to act as the application's entry point is known as the root component (see Figure 2). When a user invokes the associated URL, the Web Dynpro framework creates an instance of the application’s root component.

   This component instance will persist until such time as the user formally terminates the application, or closes their client (e.g. the browser), enters a new URL, or remains inactive for the configured time out period.

2. **Lifespan of a child component**
   Any Web Dynpro component may act as the child of any other Web Dynpro component. In such cases, the lifespan of the child component may either be controlled automatically by the Web Dynpro framework, or it may be controlled by coding written by the application developer in the parent component.

**The faceless Component**

Should you require it, it is possible to create a Web Dynpro component that has no visual interface. In other words, it is a component with zero views and zero windows. Such a component is known as a “faceless” component and is useful when a complex unit of functionality requiring no direct user interaction needs to be encapsulated.

A good example of a faceless component is the creation of something called a model component. This is not actually a specific Web Dynpro component type; rather it is a standard Web Dynpro component that has been written specifically for the task of interacting with a model object.
Often, a model object could have large or complex interface. Since model objects are completely reusable, it does not make sense to have to reimplement the functionality required to interact with the model object every time it is reused. Therefore, SAP recommends that when a model object’s interface requires any type of additional processing or simplification, that the model object be encapsulated within a component in order to be able to reuse the associated interface coding. Very often (though not exclusively) a model component will have no visual interface, thus making it a faceless component.

The model component then becomes the unit of reuse rather than the model object.
A quick overview

By referring to the diagram in Figure 3, you can see that a Web Dynpro component is not just a single program. Instead, it is composed of several programs that interact with each other to fulfill the required task. Before we look at the details of how all these different units of code function within the component, it is worth showing the end result first.

On the right of Figure 4 below is a table that will be seen on the client. The data displayed in this table is obtained from a data storage area in the view controller known as the context. The view controller, in turn, obtains its data from the component controller’s context.

The beauty of the Web Dynpro programming model is that the relationships between the different controllers are created declaratively. It is not necessary to write any code to transfer data from one controller to another. The only code that does need to be written is that which populates the context of the component controller; everything else happens automatically.

So, using nothing more than a declared chain of relationships, it is possible to display data held in the context of a component controller without having to write a single line of code to transport the data anywhere.

Figure 4 above shows that the data held in the context of the component controller (on the left) can be made available to the context of the view controller, which in turn, makes the data available to the various UI elements.

Thus, data will automatically appear on the screen. The only manual coding necessary is that required to place the data into the component controller’s context. The Web Dynpro framework handles everything else for you based on your design time declarations.

We will now take a closer look at the controllers found within a Web Dynpro component.
Web Dynpro Controllers

Definition

Controllers are the active parts of a Web Dynpro component. In the design of Web Dynpro controllers, SAP has made a significant modification to the original MVC concept of a controller.

In broad terms, SAP has defined two categories of Web Dynpro controller. The difference between them is simply this: A controller either

- Has a visual interface, or
- Does not have a visual interface.

SAP has introduced this difference in order to maintain a strict separation between those parts of the business application that display data (typically data consumers), and those parts of the business application that process data (typically data generators).

Controllers having no visual interface

The red ovals in Figure 5 below show the controllers in a Web Dynpro component that have no visual interface. As is to be expected, all these controllers are located entirely on the “Programmatic Entities” side of the diagram.

Controllers that have no visual interface are generically known as custom controllers. At a basic level, the component controller and the interface controller can be thought of as special cases of custom controllers. This statement is a somewhat simplistic, but at an introductory level it serves well to illustrate the general behaviour.
Structure of custom controllers

All custom controllers have the following structure:

![Diagram of Web Dynpro Controllers structure]

All controllers lacking a visual interface are primarily responsible for generating data that will ultimately be supplied to a view controller for display.

In order to gain access to the data and functionality found in other Web Dynpro coding entities, usage declarations must be made. These declarations are represented by the three boxes down the left of Figure 6 entitled “Other WD Controllers”, “Other WD Components”, and “Models”.

**Required Controllers**

All Web Dynpro controllers are independent programs in their own right; therefore, if access is required to data or functionality found in another controller (within the same component), then a specific usage declaration must be made. Only after this declaration has been made is it possible to access the data and functionality found in the other controller. Data is shared between controllers by means of a mechanism known as “context mapping”.

There are two important points to be made about the usage declarations between custom controllers. A custom controller is only permitted to:

- Declare the use of another custom controller.
- Declare the use another custom controller within the same component.

It is not permitted to declare the use of a view controller as a required controller as this would violate the principle that view controllers are only permitted to act as consumers of data, not generators.

**Model Usage**

Model objects are created independently from Web Dynpro components. However, once a model object has been declared for use (at the Web Dynpro component level), all controllers within the component have implicit access to the data and functionality found within it.
Be careful here! It is technically possible for a view controller to interact directly with a model object. But would this violate any known principles of Web Dynpro design? The answer is most definitely “Yes!”

A view controller is **not** responsible for generating the data it displays; therefore, whilst it is technically possible, a view controller should not interact directly with a model object.\(^{10}\)

### Component Usage

In order to maximise code reuse, atomic units of business functionality should be packaged into components. Then, in order to make use of these reusable units of code, one component can declare the use of another component. This forms a Parent-Child relationship between the two components, with the child’s interface controller being the point of interaction.

As soon as the component acting as the parent, declares the use of the child component, the parent component controller has access to all the data and functionality found in the child component’s interface controller.

### Implementation

The implementation of any Web Dynpro controller will always contain a standard framework of generated code, within which are sections of code written by the developer at design time.

#### Standard Hook Methods

The standard hook methods act as a static interface between the Web Dynpro framework and the coding written by the application developer. These hook methods may not be renamed or modified in any way; however, the Web Dynpro developer will frequently need to insert their own coding into these methods.

The Web Dynpro developer is not responsible for explicitly calling any of the hook methods; that job is performed by the Web Dynpro framework. They will be called automatically by the Web Dynpro framework at predefined points in time such as component initialisation, and just prior to the controller’s end-of-life (to name but a few).

#### Instance Methods

The Web Dynpro developer may declare the existence of any instance methods they feel necessary for the completion of their business task. Such methods will never be called directly by the Web Dynpro framework, but the developer may insert calls to their instance methods from coding they have added within one of the standard hook methods.

#### Events

Events are declared runtime objects that represent standard server side triggers for invoking subsequent business processing. An instance method may be declared to act as a handler for such an event.

#### Context

Irrespective of its type, every Web Dynpro controller has a data storage area known as the context. This is a hierarchical repository within which all runtime data used by the controller is stored. The metadata used to define the structure of the context is typically defined at design time, but can also be defined dynamically at runtime.

As soon as the controller instance reaches the end of its lifecycle, then the all the data in its context is lost. It is incumbent upon the Web Dynpro developer to write the necessary coding to store all required context data before the controller reaches this point.

---

\(^{10}\) There is one legitimate exception to this rule. In Web Dynpro for Java, a certain type of pop-up help selector can be written called an “Object Value Selector”. In this situation, it is permissible for a view controller to interact directly with a model object, and thus generate its own data.
The structure and behaviour of the context will be discussed in greater detail at a later point in this document.

**Controllers having a visual interface**

The red ovals in Figure 7 below show the controllers in a Web Dynpro Component that have a visual interface. These controllers span both the “Visual Entities” and “Programmatic Entities” sides of the diagram.

![Figure 7: Web Dynpro Controllers that have a visual interface](image)

Having view controllers defined in the manner, Web Dynpro programs are able not only to separate business logic (contained in the component and custom controllers) from presentation logic (contained in view controllers), but a further level of abstraction can be implemented by separating the coding that manipulates the presentation data (the view controller) from the UI elements used to display the data (the view layout).

The split between the view controller and the view layout has been shown only to highlight this specific feature of a Web Dynpro component. In reality, there is a one-to-one relationship between a view controller and its corresponding view layout, and the two are *inseparable*.

**Structure of view controllers**

All controllers that have a visual interface conform to the following structure.
All controllers that have a visual interface are responsible for interaction with the user. This covers all aspects of data presentation, and calling action handlers in response to user input.

Notice also that the controller structures shown Figure 6 and Figure 7 have a large number of similarities.

There are two types of functionality that should not be implemented in a view controller!

1. Although it is technically possible to do so, you should not implement coding in a view controller that causes it to generate its own data. E.G. You should not put coding into a view controller that interacts directly with a model object. If you do, then you have violated the principle that view controllers should act only as consumers of data, not generators.  

   For those readers writing Web Dynpro for ABAP programs, you must be especially careful to implement this principle. It is easy to violate, but will have significant consequences on the future maintainability of your coding!

2. The context of a view controller cannot be used to hold persistent data. Once a view controller is no longer a member of the current View Assembly (See "The View Assembly Concept" on page 31), then the view controller instance and all its data is lost.

In order to gain access to the data and functionality found in other Web Dynpro coding entities, usage declarations must be made. These declarations are represented by the three boxes down the left of Figure 6 entitled “Other WD Controllers”, “Other WD Components”, and “Models”.

---

As stated earlier, there is only one situation in which a view controller may legitimately interact with a model object. This is when a type an Object Value Selector (OVS) is being written. A discussion of OVS implementation however, is beyond the scope of this document.
Required Controllers

As per the “Required Controllers” for a custom controller, a view controller may declare the use another custom controller within the same component in order gain access to its data and functionality.

A view controller is not permitted to declare the use of another view controller as a required controller, since this would violate the principle that view controllers are only consumers of data, not generators.

Model Usage

Although it is technically possible, it is considered very poor Web Dynpro design for a view controller to interact directly with a model object. Should you find yourself needing to write this type of code then it is very likely that you have failed to implement the principles of Web Dynpro component design correctly.

Component Usage

As per custom controllers, a view controller can inherit the use of a child component. Usually however, a view controller will not need to interact directly with a child component instance. However, since the possibility exists for a child component to perform some processing related only to data presentation, such interaction is technically possible.

Implementation

The implementation of any Web Dynpro controller will contain a standard framework of generated code, plus any sections of code written by the developer at design time.

Standard Hook Methods

The standard hook methods act as a static interface between the Web Dynpro framework and the coding written by the application developer. These hook methods may not be renamed or modified in any way; however, the Web Dynpro developer will frequently need to insert their own coding into these methods.

The Web Dynpro developer is not responsible for explicitly calling any of the hook methods; that job is performed by the Web Dynpro framework. They will be called automatically by the Web Dynpro framework at predefined points in time such as component initialisation, and just prior to the controller’s end-of-life (to name but a few).

Instance Methods

The Web Dynpro developer may declare the existence of any instance methods they feel necessary for the completion of their business task. Such methods will never be called directly by the Web Dynpro framework, but the developer may insert calls to their instance methods from coding they have added to one of the standard hook methods.

Actions

Actions are a view controller specific mechanism for responding to events raised on the client device as a result of user interaction.\(^\text{12}\)

Once an action has been declared, it can then be associated with any number of UI element events. In this manner, it is possible to write a single, generic action event handler method that can respond to a variety of events raised by multiple UI elements.\(^\text{13}\)

For instance, you have placed a Button UI element on the view layout, and you wish the server to respond when the button is pushed. The required steps are as follows:

\(^{12}\) It is not appropriate (or even possible) to expect a view controller Action to respond to a server side event.

\(^{13}\) In the case that generic action event handlers are to be written, extra parameter information will need to be supplied to the action handler method in order for it to distinguish which UI element event invoked it.
1. Create an action object in the view controller. Unless you say otherwise, this will cause an action handler method to be created.

2. Place a Button UI element on the view layout. The Button UI element is able to raise a client side event.

3. Associate the UI element event with the action.

As a result of performing these configuration steps, when the Button UI element raises its client side event, a round trip to the server will take place and the associated server side action handler method will be invoked. See Figure 9.

**Figure 9: A client side event causes a server side action event handler to be executed**

**Navigation Plugs**

Navigation from one view to another is performed using navigation plugs. A plug can either be a navigation entry point, known as an “inbound” plug; or it can be a navigation exit point, known as an “outbound” plug.

Inbound and outbound plugs are declared on a per view basis. A plug belongs to a specific view controller and in order for navigation to take place, corresponding inbound and outbound plugs must be associated with each other at design time.14

Once this design time association has been made, the developer should implement the coding in the action event handler method that decides whether navigation is to take place or not (E.G. only perform navigation if all the data received from the user valid).

Since a design time association has already been made between an outbound plug and one or more inbound plugs, it is sufficient for the developer simply to call the method generated by the declaration of the outbound plug. The Web Dynpro framework then automatically handles the execution of the corresponding inbound plugs in the target views.

**Context**

Irrespective of its type, every Web Dynpro controller has a data storage area known as the context. This is a hierarchical repository in which all runtime data used by the controller is stored. The metadata used to define the structure of the context is typically defined at design time, but can also be defined dynamically at runtime.

As soon as the controller instance reaches the end of its lifecycle, then all the data in its context is lost. It is incumbent upon the Web Dynpro developer to write the necessary coding to store all required context data before the controller reaches this point.

---

14 Normally, most steps performed at design time can also be performed at runtime through the Web Dynpro framework API. However, the association of an inbound plug with an inbound is one of the few exceptions to this rule.
The structure and behaviour of the context will be discussed in greater detail later in this document.

**Layout**

The arrangement of User Interface (UI) elements used to present information on the client device is known as the view layout. There is a one-to-one relationship between the view controller and its corresponding layout. It is not permitted for one view controller to supply data to multiple view layouts, neither is it permitted for one view layout to obtain data from multiple view controllers.

If you refer back to the section on SAP’s implementation of the MVC paradigm, it should be apparent now that UI elements act primarily as consumers of data from their corresponding view controller. However, certain UI elements (such as **InputFields** and **DropDownLists**) can also act as data generators (I.E. the user enters data through these UI elements that is then supplied to the view controller coding). Here, we see an implementation of the principle of separating data generators from data consumers – and allowing for that fact that two way data flow is also required.

**UI Elements**

UI elements are specified in a manner that is client agnostic. This means that the application developer need have no concern for the technical implementation differences of a UI element between various client devices. The only client specific concern a Web Dynpro developer need have surrounds the amount of physical space available on the client's screen. All other technical factors are handled automatically by the Web Dynpro framework at runtime.

All UI elements have a set of properties, many of which can accept hard coded values. However, when the data to be displayed on the screen is held in the context of the view controller, a simple declarative relationship can be made between the appropriate UI element property and the attribute within the view controller's context that holds the required data. This is relationship known as data binding, and is the standard mechanism for presenting data to the user.

Using this relationship, the Web Dynpro developer need only write coding that populates the context attribute. Once the data is present in the context, the Web Dynpro framework ensures that the value of the UI element property is obtained from the associated context attribute. Consequently, a Web Dynpro program can often be written that does not contain any lines of code concerned with rendering the UI!

Another direct consequence of this fact is that the developer need not obtain a reference to the UI element object in order to find out what value the user entered. The UI element object is merely a tool for presenting the data found in the context. So if you want to know what the user typed in, don’t look in the UI element object, look in the context!

**Specific controller types**

**Component Controller**

The component controller is the master controller for a Web Dynpro component. As soon as you declare the existence of a Web Dynpro component, the component controller is automatically created. You cannot have a Web Dynpro component without a corresponding component controller.

This controller can be thought of as being hierarchically superior to all other controllers in the component. Consequently, the lifespan of the component is always equal to the lifespan of the component controller.

When the Web Dynpro framework receives a URL that corresponds to a Web Dynpro application, the component controller of the root component is instantiated. Here is an example of where the Web Dynpro framework automatically interacts with coding written by the Web Dynpro developer.
Since the component controller has no visual interface, it is incapable of directly presenting any information to the user. This task however, is of no concern to the component controller. The component controller should contain only the coding necessary to control the interaction with:

- All other controllers
- Child component instances
- Model objects

You should regard the component controller as the central point of control for the behaviour of the Web Dynpro component.

**Custom Controllers**

Custom controllers are similar to the component controller in that they too have no visual interface. However, the existence of custom controllers is controlled entirely by the Web Dynpro developer at design time. Declaring the existence of a Web Dynpro component causes the component controller to be created automatically, but this is not the case with custom controllers. Custom controllers are only created by an explicit design time declaration.

**When should a custom controller be used?**

Custom controllers should be created only when their presence will simplify the overall structure of the component. The principles to follow for deciding when a custom controller is needed are controlled by:

- **Reuse**
  Create a custom controller if you have identified a reuse case for a specific unit of code required by several other controllers within the component.

- **Task separation**
  To avoid placing too much coding in the component controller (and thereby potentially increasing its initialisation time), place coding dedicated to a specific task into a separate custom controller

**Lifespan of custom controllers**

The lifespan of a custom controller is determined by a parameter setting made during the design time declaration. It can be either “Framework Controlled” or “On demand”.

If you choose “Framework Controlled”, then the Web Dynpro framework will instantiate the custom controller when the component is instantiated.

If however, you choose “On demand”, then the Web Dynpro developer must write the coding necessary to instantiate the custom controller.

Each child component usage is instantiated with a unique name that must be defined at design time. During the lifespan of the parent component, a child component may only ever be instantiated once under a given name; however, should it be necessary, you may declare multiple usages of the same child component as long as you specify different usage names.

**Interface Controller**

It has already been stated that one Web Dynpro component may declare the use of another Web Dynpro component to form a “Parent-Child” usage relationship. In such cases, the only point of interaction between the parent and child components is the functionality found in the child’s interface controller.

Referring back to Figure 3, there is a usage declaration arrow pointing from the component controller to a child component. Notice that the arrow joins the child component at the top right-hand corner. This is the location of the child’s interface controller.

At no time does the Web Dynpro framework interact with a component’s interface controller. It is only ever used when a Parent-Child relationship exists between two Web Dynpro
components. Therefore the methods declared in an interface controller will only ever be called by coding written by the application developer.

⚠️ Java/ABAP difference

The Java implementation of Web Dynpro has been available since late 2003. In versions of Web Dynpro for Java based on this architecture (NetWeaver 04 and NetWeaver 04s), the interface controller is a distinct Java class that is instantiated whenever an instance of the child component is created.

This architecture will be deprecated in the next major release of NetWeaver for Web Dynpro for Java. The NetWeaver 04s release of Web Dynpro for ABAP already contains the new architecture.

Therefore, when creating a component interface using Web Dynpro for Java in NetWeaver 04 or NetWeaver 04s, SAP strongly recommends that no application coding is written directly into the methods of the interface controller. Instead, the public methods of the interface controller should simply delegate their functionality to a method of the same name in the component controller. This will greatly reduce the conversion task required when upgrading to the next major release of NetWeaver.

This warning does not apply to Web Dynpro for ABAP in NetWeaver 04s. Since the new architecture was known at the point when Web Dynpro for ABAP was being built, it was considered pointless to implement a structure for Web Dynpro ABAP in NetWeaver 04s that would immediately be changed in the next release.

Therefore, Web Dynpro for Java in the next major release of NetWeaver and Web Dynpro for ABAP in NetWeaver 04s have the following architectural change to that documented here:

The public interface of a Web Dynpro component has been implemented within the component controller itself. In these versions of Web Dynpro, when either a method, or an event or a context node is declared, there is an additional “interface” checkbox; which, if checked, will expose the object through the component’s public interface.

View Controller

A view controller is the controller within a Web Dynpro component that handles both the presentation of business data, and processing of responses to user input.

⚠️ A view controller is not responsible for the generation of business data.

Remember the fundamental principles of MVC design? There is a strict division of labour between those parts of the program that generate data, and those parts that consume data. With respect to the other controllers in a component, a view controller is always considered to be a consumer of data; therefore it can never be nominated as a “Required Controller” by any other Web Dynpro controller.

View Layout

A view layout holds the description of the appearance and behaviour of a rectangular area of the user interface. This description is typically defined at design time, and developers need not concern themselves with the specific technology implemented by the client device. This greatly simplifies the Web Dynpro developer’s job because they no longer need to spend those many tedious hours trying to make some graphical widget work in three different browser versions.

The positioning of UI elements within a Web Dynpro view layout is typically performed using design time declarations. No consideration need be given as to whether the client understands a certain version of JavaScript, or whether a certain HTML tag is supported. The Web Dynpro framework automatically handles all these considerations for you at runtime.

In respect of the view layout, the only thing the Web Dynpro developer need be concerned about is the amount of screen space available on the client device. A hand-held device will have only a small amount of screen real-estate available, whereas a browser running on a desktop computer will have a much larger amount of screen real-estate available. This will
affect the quantity of information you can present to the user in one screen, but from a Web Dynpro programming perspective, it has no bearing whatsoever on the application coding.

**UI Elements**

All view layouts need UI elements in order for information to be made visible on the screen. UI elements are abstract representations of graphical elements found across the range of client devices supported by Web Dynpro.

When you declare that a certain UI element will be used on the screen, you are **not** writing any actual HTML or JavaScript: the reason being that the view layout is client independent and it is not until runtime that the Web Dynpro framework actually decides what specific encoding is required to render the screen.

**Interface View Controller**

The visual interface of a Web Dynpro component is supplied via the Interface View. This behaves much like a UI element in so much as it can be embedded into the view layout of a parent view as a single, reusable entity.

Once the interface view of the child component has been embedded into a view layout belonging to the parent component, the parent component then has programmatic access to the visual interface of the child component through the child’s interface view controller.

---

### Java/ABAP difference

The Java implementation of Web Dynpro has been available since late 2003. In versions of Web Dynpro for Java based on this architecture (NetWeaver 04 and NetWeaver 04s), the interface controller is a distinct Java class that is instantiated whenever an instance of the child component is created.

This architecture will be deprecated in the next major release of NetWeaver for Web Dynpro for Java. The NetWeaver 04s release of Web Dynpro for ABAP already contains the new architecture.

Therefore, when creating a component interface using Web Dynpro for Java in NetWeaver 04 or NetWeaver 04s, SAP strongly recommends that no application coding is written directly into the methods of the interface controller. Instead, the public methods of the interface controller should simply delegate their functionality to a method of the same name in the component controller. This will greatly reduce the conversion task required when upgrading to the next major release of NetWeaver.

This warning **does not** apply to Web Dynpro for ABAP in NetWeaver 04s. Since the new architecture was known at the point when Web Dynpro for ABAP was being built, it was considered pointless to implement a structure for Web Dynpro ABAP in NetWeaver 04s that would immediately be changed in the next release.

Therefore, Web Dynpro for Java in the next major release of NetWeaver and Web Dynpro for ABAP in NetWeaver 04s have the following architectural change to that documented here:

The interface view controller of a component has been replaced by the concept of a Window controller. The implementation of Window controllers will probably not be required for components that are not defined as root components.

---

### Adding your own coding to a Web Dynpro Controller

All Web Dynpro components and their constituent controllers are created as a result of design time declarations. Once these declarations have been made, the Web Dynpro developer is free to add their own coding at predefined locations.

All Web Dynpro controllers have a well defined Application Programming Interface (API). This API comprises a static part that can always be assumed to be present, but it also has a variable part that is generated in response to design time declarations. This gives Web Dynpro programs a highly flexible, and developer friendly API.
Whenever you wish to implement your own functionality within a Web Dynpro controller, it can only be done at predefined locations within the source code. These locations are identified with special comment markers. Any attempt to enter source code outside these comment markers will result in the loss of that code. If any custom coding that has been entered outside the designated areas, somehow manages to survive the regeneration process, then it places the deployed application outside of SAP’s standard support agreement.

You can enter source code to implement functionality in the following areas:

- Standard lifecycle hook methods that control the initialisation and end-of-life of controller instances.
- Standard round-trip hook methods that perform controller specific tasks.
- Custom written event handlers [Page 16] that respond to specific user actions on the client.
- Custom controllers that encapsulate some reusable unit of functionality.
- Rebuilding data held in the controller’s context.
- Etc…
Web Dynpro Model

Caveat

Due to the fact that Web Dynpro for Java programs executed by the SAP Java Server typically need to access remote systems in order to obtain business data and functionality, model objects are required in order to encapsulate the communication functionality.

This however, is not the case for a Web Dynpro for ABAP program. Calls to RFC enabled function modules or Web Services appear simply as particular ABAP statements within your controller coding. The SAP system itself performs all the necessary communication with external systems. It is superfluous therefore, for an ABAP based Web Dynpro program to make use an extra abstraction layer (a model object) between its own coding and the required business data and functionality. Consequently, the concept of distinct model objects described in this document is only relevant for Java based Web Dynpro applications.

The only similarity between the Java and ABAP implementations of Web Dynpro in respect of model objects, is that it is possible (and often beneficial) to write model components in both environments. As stated above, the purpose of a model component is firstly to simplify, and then to reuse, complex functionality required to access business data.

Definition

A model object encapsulates access to business data and functionality that typically resides in some remote backend system. Since a Java based Web Dynpro application will be executed in a different runtime environment from the backend business functionality, it will be highly dependent upon the use of model objects to act as proxies between the Java Web Dynpro environment and the remote system.

In the Java environment, a model object is used to encapsulate access to:

- Remote enabled function modules (RFMs) in SAP systems
- Enterprise Services Models
- Web services
- Enterprise Java Beans

From the Java perspective, a model object hides the technical communication layer needed to access data from a remote system, and simply provides an interface with a set of methods sufficient for obtaining the required data. Only in certain specialised cases will the Web Dynpro developer need to be concerned with the details of the underlying communication layer.

In the ABAP environment, the encapsulation of functionality into distinct model components is possible, but not necessarily required. However, access is still possible to:

- Any function module in the local SAP system
- Any other ABAP based Web Dynpro component in the local SAP system
- Remote enabled function modules in other SAP systems
- Enterprise Services Models in SAP systems (local or remote).
- Web services
- Enterprise Java Beans

15 If, and only if, the Enterprise Java Beans have first been wrapped as Web Services.
The Window Concept

Caveat

The following section applies mainly to Web Dynpro for Java. Web Dynpro for ABAP components have a new controller type called a Window controller that is documented in a separate appendix.

⚠️ The concepts of a window controller and a window plug have not been implemented in Web Dynpro for Java in SAP NetWeaver 04 or 04s.

Overview

A window represents the sum total of all views making up an application's visual interface.

As can be seen from Figure 10, a Web Dynpro component has two interfaces – one visual and one programmatic. The visual interface is supplied through the interface view controller. From the diagram, you will also notice that there is a one-to-one relationship between an interface view controller and a corresponding window.

The task of creating the first window for any given component is performed automatically when the component is first declared. Thereafter, the existence of new windows can be declared as required.

From a development perspective, the window defines two things:

- The superset of all possible views that could be displayed for one particular visual interface of the component.
- The navigation paths that exist between those views.
Once a view has been created, it will not be visible on the front end device until it has first been embedded into a window. As soon as a view is declared to be a member of any particular window, it can be made visible on the front end client device.

There is no particular requirement to use a view set; however, in Web Dynpro for Java, this possibility exists if needed.

**The View Set Concept**

**Caveat**

⚠️ The view set concept is not implemented in Web Dynpro for ABAP.

If you wish to build your user interface from multiple views, then this achieved by adding view container UI elements to a parent view, and then embedding views within these containers. This technique is equally applicable for Web Dynpro for Java.

**What is a View Set?**

A view set is a visual framework that subdivides the window into predefined areas. Each subdivision of a view set is known as a view area, and multiple views can be embedded into a single View Area.

The following preconfigured view sets are available:

<table>
<thead>
<tr>
<th>T layout</th>
<th>T layout 90°</th>
<th>T layout 180°</th>
<th>T layout 270°</th>
<th>Grid layout</th>
<th>Tab strip</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="https://example.com/layout1.png" alt="Image" /></td>
<td><img src="https://example.com/layout2.png" alt="Image" /></td>
<td><img src="https://example.com/layout3.png" alt="Image" /></td>
<td><img src="https://example.com/layout4.png" alt="Image" /></td>
<td><img src="https://example.com/layout5.png" alt="Image" /></td>
<td><img src="https://example.com/layout6.png" alt="Image" /></td>
</tr>
</tbody>
</table>

Each subdivision within the view set layout is known as a view area.

**Empty View**

There is a special type of view known as the empty view. This view requires no manual implementation, neither is it possible to interact with it in any way other than invoking its default inbound plug – `showEmptyView`.

If you require one particular area of a view set to be empty, then you should embed the empty view into the view area. You can then treat this view just like any other view you have written, except that calling its inbound plug will cause the corresponding view area to be blanked out.

If a view set has had no views manually embedded into one of its view areas, then the empty view will be substituted automatically.

**Reusing views in Web Dynpro for ABAP**

So if the view set concept is not implemented in Web Dynpro for ABAP, what options are there for reusing views? In both Web Dynpro for ABAP and Java, there is a specific UI Element called the ViewContainer. This UI element, when added to a view layout, acts as a container for any other view. ViewContainers can be arranged in large variety of ways in order to achieve the desired layout on the screen.

The views that can be embedded into a ViewContainer UI element are the following:

- Any view from the current component
- Any visual interface from a child Web Dynpro component
- An empty view (supplied automatically by the Web Dynpro runtime)
The View Assembly Concept

As stated above, a window defines the superset of all possible views that a Web Dynpro application could require whilst running a particular component. The number of views visible at any one time however, will typically be only a subset of the number of views embedded within the window.

The subset of views rendered at any one time is known as the View Assembly. User interaction followed by subsequent navigation processing will frequently cause this subset of views to change with every server round-trip.

The view assembly represents those views seen by the user on their client device after the completion of a particular server round trip.

How does the Web Dynpro framework decide which particular views make up the current view assembly? There are two answers to this question:

1. When an application is executed for the first time, only those views which have their default flag set to true will belong to the first view assembly.
2. Thereafter, user navigation will occur and the view assembly will be composed of those views that have been newly instantiated (on account of their inbound plugs being fired), and those views that persist from the previous view assembly (because no outbound navigation took place from them).

![Figure 11: Design time appearance of views embedded within Window](image)

You will notice in Figure 11 above that the window has been defined to contain two view sets, each of which has multiple views embedded within it. Some view areas even have multiple views embedded within them. This is a perfectly legitimate window definition.

Notice that three of the embedded views have a darker blue background compared to the others. This change of background colour indicates that these are the default views. These views will be rendered in the first view assembly without any previous view determination processing being required. The resulting first view assembly therefore looks like the Figure 12 below:
Figure 12: The first view assembly is rendered using the default views

Now as a result of the first view assembly being rendered, the user can see information on their client device, and interaction is now possible. As a result of user interaction, navigation will most likely occur, and this could cause the next view assembly to look like Figure 13 below.

Figure 13: The next view assembly after navigation has occurred

Compare Figure 12 and Figure 13. Notice that user navigation has caused the two empty views in Figure 12 to be replaced by the “Customer List” and “Product Details”.

⚠️ A view controller instance (and therefore the data it contains) will persist only for as long as it is a member of the current view assembly. As soon as a view controller instance is no longer part of the current view assembly, it is thrown away and all the data it contains is lost!

Therefore, Web Dynpro view controllers should never be used to store data that is required to persist beyond the view’s membership of the current view assembly. Persistent data should always be stored in the component controller, or possibly a custom controller, since such controllers persist for the lifespan of the entire component.
Why would a component need multiple windows?

A good example of where a Web Dynpro component would need multiple windows is where a single business application needs to be accessible on variety of client devices. For example, a particular application needs to be written that can be executed from both desktop based browsers and handheld devices.

In order to avoid having to write the same business logic twice, you can write a single Web Dynpro component but within it, you define two sets of views. The first set of views has been laid out with a desktop browser in mind (i.e. there will be a lower number of views because a larger quantity of data can be presented on each view). The second set of views however, is laid out with a handheld device in mind (i.e. the restricted space on the handheld device will mean that more views will be needed in order to present the same quantity of information).

The two sets of views are then grouped together into different windows; one for the desktop based browser, and the other for the handheld device. Couple this design together with the principle that view controllers are not responsible for generating the data they display, and you should quickly be able to see that all the business logic need only be written once and placed within the component controller and custom controllers. The view controllers then simply display (consume) the data supplied to them by the non-visual controllers.

The last step is to define two different Web Dynpro applications. Both applications will use the same Web Dynpro component, but since two windows have been defined, there will be two Interface view controllers – one for each window. These interface view controllers are then used to define the visual interface of each application.

A second example for a component with more than one window is the use of popup windows. A popup window will always be implemented by a separate window which may be defined in the same component, but processed as an independent window.
Plugs and Navigation Links

If you wish one view to be removed from the current view assembly, and replaced by another view, then you must define navigation plugs in each view and connect them together with a navigation link.

Navigation plugs are view specific and come in two varieties: inbound and outbound.

Outbound Plugs

When an outbound plug is fired, a “request for navigation” message is sent to the Web Dynpro framework. The Web Dynpro framework takes this request and places it in a stack that will only be processed after all the views in the current view assembly have responded to user events.

Inbound Plugs

An inbound plug is a method in the target view that is able to respond to the outbound plug’s “request for navigation”. Once all the views in the current view assembly have finished their processing, the Web Dynpro Framework turns its attention to the requests on the navigation stack.

In order for each “request for navigation” to be processed successfully, you must have already declared an association between an outbound plug and an inbound plug. This association is known as a Navigation Link.\(^\text{16}\)

Within the scope if window, any view having its default parameter set to true, does not need to be called using a developer defined inbound plug in order for it to be seen when the screen is first rendered.

Navigation Links

To navigate from one view to another, each outbound plug must be associated with an inbound plug in the target view. This association is known as a navigation link, and can only be defined at design time.

You are not restricted to having a one-to-one relationship between outbound and inbound plugs. It is possible to have one outbound plug associated with many inbound plugs. Similarly, it is possible to have many outbound plugs associated with one inbound plug.

By declaring a navigation link you create a fixed association between two views in a window. Once you fire the outbound plug, the “request-for-navigation” is unconditionally placed on the navigation stack. After this point in time, the only options available to you concerning navigation processing are:

- Process all the requests in the navigation stack, or
- Abort the processing of the entire navigation stack

There is no possibility to selectively process navigation requests once they have been added to the navigation stack. You either process the whole stack, or you abort the navigation processing entirely. Therefore, when writing navigation coding, care should be exercised before deciding to fire an outbound plug.

\(^{16}\) If you fail to create a navigation link between an outbound and an inbound plug, then firing such an outbound plug will cause a fatal error in the Web Dynpro framework.
### The Context

#### Definition

Every Web Dynpro controller has exactly one hierarchical data storage structure known as a context. The data held in the context exists only for the lifespan of the controller. Once the controller instance has been terminated, all data held within its context is lost.

![Figure 14: The context is at the heart of every Web Dynpro controller](image)

Before any data can be stored in the context at runtime, metadata must first be created that describes the runtime data. The creation of context metadata is often done at design time, but can also be done dynamically at runtime.

#### Data visibility

Although each Web Dynpro controller cannot function in isolation from the Web Dynpro component within which it has been defined, each controller is nonetheless an independent program. Consequently, controllers do not normally have access to each other’s data or functionality.

#### Sharing data between different controllers

In order for two controllers to be able to share data with each other, one controller must expose its context through its public interface. However, it has already been stated that a view controller is only ever permitted to act as a data consumer, and never as a data source; therefore, the context of a view controller is only ever part of its private interface, whereas the context of a custom controller is always part of both its public and private interfaces.

Information held in the context of a custom controller can be made accessible to the context of another controller (view or custom) by a technique known as context mapping. Using this technique, two or more controllers can access the same runtime data.
Context mapping is the primary mechanism for sharing data between controllers within a single component.

It is not permitted for a view controller to share its context data. Since a view controller is not responsible for generating the data it displays, it should therefore never need to act as a data source in a context mapping relationship.

Context mapping is discussed in greater detail in the section Context Mapping on page 46.

**Context structure**

All controller contexts are constructed from a hierarchical arrangement of entities known as nodes and attributes. A context always has a parent node known as the *root node*.

For Java programmers, the context should be thought of as a type-safe collection that may in turn contain other type-safe collections.

For ABAP programmers, the context should be of as in internal table into which multiple levels of deep structures can be nested.

**Context Nodes**

A context node is the main abstraction class used for runtime data storage within the Web Dynpro framework. Context nodes are arranged hierarchically and may have attributes or other nodes as children.

All the child entities of a node are aggregated into a unit known as an element. A node is a collection of such elements in the same way that a table is a collection of rows.

The maximum and minimum number of elements permitted in a node collection is controlled by a pair of value collectively known as the node’s cardinality.

All nodes contain an element collection, even if the maximum number of elements within the collection is limited to one.

⚠️ All node names must be unique within the scope of a controller's context.

**Context Attributes**

An attribute is a hierarchical entity within the context that is not permitted to have children. A context attribute cannot exist without being the child of some parent node — be it the context root node itself, or some other node.¹⁷

All context attributes must have a declared data type. The data type may be one of the built-in Web Dynpro data types, or it may be a simple type defined in the Dictionary, or any other valid object type.

⚠️ All attribute names must be unique within the scope of their parent node.

**The context root node**

The first node in any context is the root node. It is a standard node just like any other node in the context; however, its properties have been hard coded. The context root’s element collection may contain one, and only one element, and its singleton property has been hard coded to true.

¹⁷ Please avoid creating attributes directly under the context root node. This is considered bad programming style.
The context root node may not be deleted neither can its properties modified in any way!

**Context Terminology**

The nodes or attributes that have the context root as their immediate parent are referred to as being *independent*. This is because their existence at runtime is *independent* of any coding written by the Web Dynpro developer. That is, all independent nodes or attributes will be instantiated automatically at runtime.

You cannot modify an independent node’s singleton property; it will always be set to true.

However, nodes or attributes that have some other node as their parent are referred to as being *dependent*. This is because their existence is *dependent* upon coding written by the Web Dynpro developer. At runtime, it cannot necessarily be assumed that a particular dependent node instance exists. The Web Dynpro developer will need to write the appropriate code to instantiate such nodes.

**A Basic Context Structure**

As can be seen from Figure 15, a node has been created called `SalesOrders`. This node has the context root node as its immediate parent, making it an independent node. Underneath the node `SalesOrders`, four context attributes have been created called `OrderNo`, `SalesDate`, `SalesRep`, and `LongText`.

The yellow container with the dotted border represents the node collection itself, and the four attributes define the element structure that will be seen at runtime. From an ABAP perspective, the node `SalesOrders` can be thought of as an internal table and the four attributes can be thought of as table columns.

---

18 Although it is technically possible to do so, it is not considered good Web Dynpro coding practice to create context attributes directly under the context root. SAP recommends that an independent node be used instead to contain the attribute(s).
Building the Context

At design time, you create the metadata structure within which the runtime data will live. At first glance, Figure 15 would give the impression that the context structure is a flat, two-dimensional tree, much like the display of directories and files shown in the Windows™ Explorer. However, all context nodes are collections, so there could potentially be multiple instances of each child node and attribute within a node’s collection at runtime.

The configuration of a node’s properties at design time critically affects the behaviour of the node at runtime. Failure to understand how a node’s design time properties can affect its runtime behaviour can lead to the false impression that data is missing, or has somehow become lost, from the context node.

Node Cardinality

As has been stated previously, the number of elements in a node collection can be controlled using the property known as \textit{cardinality}. This property is actually the combination of two values that, taken together, define the maximum and minimum number of elements the node collection may contain at runtime. The two values are the:

- Cardinality Minimum: 0 or 1 (known as the mandatory flag)
- Cardinality Maximum: 1 or n (known as the multiple flag)

There are four permutations for these values (specified as \texttt{<min>..<max>}):

- \texttt{0..1} Zero or one element permitted
- \texttt{0..n} Zero or more elements permitted
- \texttt{1..1} One and only one element permitted
- \texttt{1..n} One or more elements permitted

The following diagrams illustrate how the value of the cardinality property changes the runtime behaviour of the node.

![Diagram of Context Root Node with Cardinality 1..1](image)

![Diagram of Independent Node with Cardinality 0..1](image)

**Figure 16: Context Nodes with a maximum cardinality of 1**

For those nodes that have their cardinality minimum set to 1, then the node collection will be instantiated such that it already contains a single, empty element. This element is known as the \textit{default element}. 

Web Dynpro Architecture Version 645

14.03.2006
For those nodes however, that have their cardinality minimum set to 0, then the node will be instantiated with an empty collection. The Web Dynpro developer must write the code first to create and then insert, the first element into the collection.

![Context Root Node](image)

**Context Root Node**

- \( c=1..1, s=true \)
- Default Element

Now that the cardinality maximum has been set to \( n \), the node collections may contain any number of elements starting from the value specified by the cardinality minimum.

### Violating the Cardinality Constraints

A node’s cardinality defines the maximum and minimum number of elements its collection may contain at runtime. It is most important that you do not write coding that attempts to violate these constraints. If you do, then a runtime exception will be raised that will probably abort your entire Web Dynpro application. Your Web Dynpro coding should anticipate and then avoid the following situations:

- Trying to delete the default element from a node whose minimum cardinality is 1\(^{19} \)
- Trying to add a second element to a node whose maximum cardinality is 1

### Creating Dependent Nodes

When creating a dependent node, you should consider carefully what runtime behaviour you require. Consider the context structure in Figure 18. There is now a new context node called `LineItems`. At runtime, this node will be a collection of the attributes `Description`, `ItemNo`, `Price` and `Quantity`. However, this node does not have the context root as its immediate parent. Instead it has the node `SalesOrders` as its immediate parent.

---

\(^{19}\) It does not matter explicitly which element is left in the node collection, as long as at least one element remains.
As has already been stated, a node whose parent is some node other than the context root is called a “dependent” node for exactly the reason that its existence is dependent upon the coding you write. The SalesOrders context node will be a collection at runtime, and each element in the collection will be an instance of not only the attributes OrderNo, SalesDate, SalesRep, and LongText, but also of the entire child node LineItems which itself, is a collection.

At this point, it becomes critical to understand the effect of another context node property: the singleton property.

**The Singleton Property**

The singleton property of a context node describes the relationship between a child node and its immediate parent. The above description of the runtime structure of the nodes shown in Figure 18 would logically lead you to represent the number of instances of node LineItems as shown in Figure 19 below. For the sake of clarity, only some of the attributes of each context node have been shown in the diagram.

---

20 The term “singleton” is not used here with the same scope as is found within Java terminology.
We know that the context node *SalesOrders* has a child node called *LineItems*, and that the *LineItems* node is a child node with its own element collection. Therefore, you would expect that for every element in the node *SalesOrders*, there will be a corresponding instance of the child node *LineItems*. This is the situation shown above in Figure 19.

However, certain performance issues will be created if this type of structure is allowed to grow to any hierarchical depth. You would find that the number of instances of child nodes could easily start to grow exponentially, which in turn would have serious consequences for your application’s memory consumption.

To solve these potential performance issues, you can set the *singleton* property on every dependent context. This is a Boolean property that critically affects the relationship between a dependent node and its parent node.

**Setting the Singleton property to false**

The diagram in Figure 19 shows the runtime structure if the Singleton property of node *LineItems* is set to false. Now, for every element in the parent node collection (*SalesOrders* in this case), there will be a distinct instance of the child node *LineItems*.

The most important thing to understand here is that each instance of node *LineItems* is related to the respective *element* in the parent node collection. Notice that the arrows pointing to each of the *LineItems* node collections originate from the *element* in the parent node.

21 And in Web Dynpro for Java, there could also be potential performance issues related to garbage collection execution times.
Therefore, if there are \( n \) elements in the parent node, then there will be \( n \) distinct instances of a non-singleton child node.

**Setting the Singleton property to true**

SAP recommends that unless you specifically require multiple instances of child nodes (and there are several legitimate situations in which you will), that the singleton property be left at its default value of true. This has a significant impact on the runtime behaviour of the child node instance, as shown in Figure 20.

![Figure 20: A single instance of a child node](image)

If the node `LineItems` now has its singleton property set to true (which is the default), then irrespective of how many elements are present in the `SalesOrders` parent collection; there will only ever be one instance of the child node `LineItems`.

The most important thing to understand here is that the singleton node is now related to the entire parent node collection, and not any specific element within that collection. Notice that the arrow pointing to the `LineItems` node collection now originates from the entire `SalesOrders` node collection and not from any particular element within the collection.

Having singleton child nodes greatly reduces the amount of memory required for a business application, and means that the Web Dynpro application does not waste time performing unnecessary processing on business data the user has not explicitly requested.

**Lead Selection**

At runtime, any node having multiple cardinality (i.e. \( 0..n \) or \( 1..n \)) will probably have more than one element in its collection, and it is customary to process such elements in a loop. During this iterative access to the node collection, it will be necessary to identify which element is currently being processed.
The Web Dynpro framework automatically keeps track of which element in a node collection is currently being processed and refers to it as the element “at the Lead Selection”. This explains why some elements in Figure 19 and Figure 20 have a darker background colour. This is used to identify the element at the lead selection.

It now follows that every time the element at the lead selection in the parent collection changes, the contents of a singleton child node (in this case LineItems) must be resynchronized according to the details found in the new parent element. Consequently, the Web Dynpro developer must write a special method to perform this resynchronization.

**Supply functions**

In order to resynchronize singleton child nodes, the Web Dynpro developer must write a supply function. This method will be associated with the singleton child node, and its purpose is to rebuild the node collection every time the lead selection in the parent element changes. The Web Dynpro framework will then automatically call your supply function when both of the following two conditions are satisfied:

- The lead selection in the parent node has changed.
- An attempt is made to read the contents of the singleton child node.

**Lazy Data Access**

The Web Dynpro framework has been built to follow the principle of Lazy Data Access. This means that the processing required to generate data will not be invoked until the data is actually needed. When this principle is applied to the architecture of the context, it means that unless there is an attempt to access the data in a singleton child node, then even though the lead selection in the parent node has changed, the child node’s supply function will not be called.

**Recursion Nodes**

The recursion node is a special type of node used when a node hierarchy with a recursive structure needs to be created. This is needed when, for instance, the depth of the node hierarchy is not known until runtime. Using a recursion node, you can declare that a particular node structure be replicated as a child of itself. A good example here is if your context needs to hold information in the same structure as a file system, containing directories and subdirectories.

A simplistic file system can be represented by the structure shown in Figure 21. At runtime, the Directory node will hold a collection of file names in its attribute FileName. The node SubDirectory is a recursion node whose repeatedNode property points up to the node Directory.

Figure 21: A simple example of a recursion node

---

22 The cardinality of the Directory node must be set to 0..n
A recursion node is really a dummy node that acts simply as a design time placeholder. The repeatedNode property is then used to determine which node instance will occupy that location in the context hierarchy at runtime.

⚠️ You cannot nominate a recursive node to act as the data source in a context mapping relationship. Recursive node structures are restricted to the scope of a single controller.

At design time, it is necessary to write some coding that will create an instance of the recursion node. Then at runtime, an instance of the node named in the repeatedNode property is created and inserted into the node hierarchy at the location of the recursion node.

⚠️ The root node of a context cannot be used for a recursion.

If the particular element in the node collection of Directory represents a file, then the SubDirectory instance in that element will remain null. However, if you wish to create a subdirectory, then the value held in attribute FileName will be the name of the subdirectory, and the recursion node SubDirectory will hold an instance of node Directory.

After two instances of the recursion node SubDirectory have been created, the context structure shown in Figure 21 will look like this:

![Context Metadata Diagram]

⚠️ This is the only situation in which you are permitted to create multiple node instances in the context with non-unique names.
**Context Mapping**

**Definition**

As has been stated earlier, Web Dynpro controllers are independent programs that never normally share information with each other. However, one of the guiding principles of MVC design is that the tasks of data generation and data consumption should be handled by separate coding blocks. This has been achieved in Web Dynpro by the creation of two different categories of controller. The controllers responsible for generating data are the component and custom controllers, and those controllers responsible for consuming data are the view controllers.

Therefore, before a view controller can function correctly in its capacity as a data consumer, it must be able to identify another controller to act as a data source. This is achieved by declaring the name of the component controller and/or various custom controllers as “Required Controllers” (see both Figure 8 and section in order to gain access to the data and functionality found in other Web Dynpro coding entities, usage declarations must be made. These declarations are represented by the three boxes down the left of Figure 6 entitled “Other WD Controllers”, “Other WD Components”, and “Models”.

Required Controllers). Once controllers have been declared as “Required Controllers”, all their context data and instance methods become accessible for use by the consuming controller.

Any controller (component, custom, or view) can act as a data consumer. However, only component and custom controllers are permitted to act as data sources. In other words, you are not permitted to name a view controller as a “Required Controller”.

![Diagram of Web Dynpro Component](image)

**Web Dynpro Component**

- Mapped node collection
- Node collection of mapping origin

Mapping Relationship
When a mapping relationship is declared the context node acting as the consumer of data is known as the “Mapped node”, and the context node acting as the data source is known as the “mapping origin”. A mapped node can only refer to a single mapping origin node; however, a node acting as a mapping origin can be referred to by many mapped nodes.

Mapping mechanism

Before a mapping relationship can be established, the following two criteria must be met:

- There must be a suitable node available to act as a mapping origin
- A context node must exist in the consuming controller

All context nodes are runtime objects that contain, amongst other things, a collection. When a mapping relationship is declared, the node collection of the mapped node is replaced by a reference to the node collection in the mapping origin node. In this manner, context mapping never causes data to be duplicated. There is only one copy of the data, and it lives in the collection belonging to the mapping origin node.

Now, when the coding in a view controller accesses the data in a mapped node, it has full read/write access to the data in the mapping origin node, as if it were accessing a node in its own local context. Refer to Figure 22.
Displaying Data

Supplying data to UI Elements

All UI elements have a set of properties – for instance, the InputField UI element has properties such as value, password, and readOnly. By setting the value of these properties, you can modify both the appearance and behaviour of the UI element. However, it would be very tedious if, every time you wished to switch the value of the readOnly property from false to true, you had to first find the instance of the UI element object (which would require remembering its name) and then manipulate the required property values.

In Web Dynpro there is a much simpler and more elegant mechanism for controlling the appearance and behaviour of UI elements. This is known as UI element binding.

UI Element binding

Almost all UI element properties can be “bound” to appropriate attributes in the view controller’s context.23

It is not permitted to bind the property of a UI element declared in one view controller to a context attribute in another view controller. UI elements can only receive information from context attributes in the same view controller.

In this manner, the behaviour of the UI can be controlled as follows:

1. At design time, create context attributes of suitable data types to hold UI element property values.24
2. Declare UI elements in the view layout and bind the properties you wish to control to the context attributes you declared in step 1.
3. Now the coding in the view controller need not care exactly which UI elements are being used. All it need be concerned with is setting the appropriate value into the context attribute, and the UI element property will automatically receive the value.

Figure 23: UI element properties can be manipulated via the context

---

23 The only UI Element property that may never be bound to a context attribute is the id property. This property holds the name of the UI Element object.

24 All the steps listed here, can be done dynamically at runtime, but the processes by which this is done are beyond the scope of this introductory text.
This design principle has several key advantages:

- Usually, no direct access to the UI element objects is required in order to control their appearance and/or behaviour.
- The Web Dynpro developer needs to write significantly less code in order to control the behaviour of the UI.
- The coding remains client agnostic. No changes are required to the application code to account for the technical differences found in different client devices.
Appendix: Window Controller

Window Controller

⚠️ This information applies only to Web Dynpro for ABAP.

Within Web Dynpro for ABAP a window is treated much like a view in so much as it has its own controller. This provides the full level of controller functionality with methods, attributes and a context as well as inbound and outbound plugs. The reason for this is to give you the ability to decide dynamically which view(s) will be seen first.

The most important aspect of this design becomes evident when you glance at the call chain triggered by the Web Dynpro application:

The user calls a URL connected to a Web Dynpro application, the application is connected to the interface view of a Web Dynpro window. Within this window, at least one view has to be checked as the start view. Without the functionality provided by the window controller, you would be limited to having only those views defined at design time as start views.

It is entirely likely that an application developer may wish to define the start views of their application run time. The concept of plugs as a part of the window controller makes this possible. Window plugs add another step into the call chain so that the Web Dynpro application does not call the window as a whole, but calls one of the window’s inbound plugs.

As we have learned before, the call of an inbound plug always results in the call of a corresponding event handler method. On the other hand, there may be several different outbound plugs created for this window in which each outbound plug is pointed to a different target view inside the window. The coding for dynamically controlling which of these outbound plugs will be called is then implemented in the event handler method.

In other words: Your application can dynamically vary its start view(s) based on any set of received parameters (E.G. user authorizations or country specific criteria).

![Component Diagram](Image)

Figure 24: A window controller allows for dynamic calculation of the start view(s)

⚠️ Please exercise care when using the context and methods of a window controller. It is easy to violate the principles of the MVC paradigm in this controller, and whilst this will not cause your program to fail, it will create future maintainability issues.
Author Bio

Chris Whealy started working with SAP software in 1993 making assembler modifications to the RF and RV modules of R/2. He then went on to work as a Basis consultant installing and upgrading R/3 systems, starting with R/3 version 2.0B.

In May 1995, he joined SAP (UK) as a Basis Consultant and ABAP programmer; however, when the Internet boom started in 1996, he turned his attention to web-based interfaces into SAP. This lead to work with the earliest versions of the Internet Transaction Server (ITS) and consequently, he taught the first course on this subject in January, 1997. Since then, web-based front ends for SAP functionality have been the main focus of Chris’ attention.

In January 2003, he started working with Web Dynpro, and has worked closely with development team in Walldorf both learning the product and writing proof-of-concept applications. The knowledge gained whilst working with the developers became the foundation for the book "Inside Web Dynpro for Java," published by SAP Press in November, 2004.

Chris now works as the Web Dynpro expert for the SAP NetWeaver Regional Implementation Group (RIG) EMEA in Walldorf, Germany.