ABAP Web Dynpro

by

vCentric
Outcome driven Solutions.
Contents

Unit 1: Web Dynpro Introduction
Web Dynpro Introduction

Unit 2: Web Dynpro Controllers
Web Dynpro Controllers

Unit 3: The Context at Design Time
The Context at Design Time

Unit 4: Defining the User Interface
Defining the User Interface

Unit 5: Controller and Context Programming
Controller and Context Programming

Unit 6: Internationalization and Messages
Internationalization and Messages

Unit 7: Value Help and Semantic Help
Value Help
Semantic Help

Unit 8: Component Reuse
Component Reuse
Web Dynpro Introduction

What is Web Dynpro?
From a technological point of view, SAP’s Web Dynpro for Java and ABAP is a revolutionary step in the development of Web-based user interfaces. It is completely unlike any design paradigm previously used by SAP and represents a quantum leap in the development of Web-based enterprise resource planning (ERP) applications.

Web Dynpro applications are built using declarative programming techniques based on the Model View Controller (MVC) paradigm. That is, you specify which user interface elements you wish to have on the client, and where those elements will get their data from. You also define the possible navigation paths declaratively in your application. All the code to create the user interface is then generated automatically within a standard runtime framework. This relieves you from the repetitive coding tasks involved in writing HTML and making it interactive with JavaScript.

ABAP Web Dynpro has been available since SAP NetWeaver 7.0 (SAP NetWeaver Application Server 7.0). For developing the entities of a Web Dynpro application, the Object Navigator (transaction code SE80) has been enhanced.

Web Dynpro is designed to support structured development. The software modularization units are Web Dynpro components, which can be combined to build up complex applications.

Meta Model Declaration vs. Custom Coding
A Web Dynpro application is developed using a declarative programming approach. Within the ABAP Workbench there are special tools that allow you to build an abstract representation of your application in the form of a Web Dynpro meta model. The necessary source code is then generated automatically and conforms to a standard architecture known as the Web Dynpro framework. The Web Dynpro framework allows you to place the custom source code at predefined positions within the generated code. All Web Dynpro applications are constructed from the same basic units. However, through the use of custom coding, the standard framework can be extended to supply any required business functionality.

Not all implementation decisions need to be made at design time. It is possible to implement a Web Dynpro application in which the appearance of the user interface is decided at runtime. This allows a highly flexible application to be written without the need to directly write any HTML or JavaScript.
Figure 1: Meta Model Declarations vs. Custom Coding

Figure 2: Application Scenarios with Web Dynpro
Web Dynpro applications can access different kinds of data sources:

- From an ABAP Web Dynpro application, all kinds of reuse components can be addressed directly (for example, function modules or methods). It is even possible to access the database via an ABAP SELECT. However, this leads to a mixing between flow logic and business logic and should therefore be omitted.
- Web Services can be used after having generated a Web Service client object.
- The SAP Java Connector (JCo) can be used to call methods of Enterprise JavaBeans located on a J2EE Engine.

Model objects are not yet supported in ABAP Web Dynpro. The best way to have reusable entities encapsulating business logic is to create ABAP classes containing the source code. It is also possible to develop UI-free (faceless) Web Dynpro components, which only offer reusable functionality. These components can then be accessed by other Web Dynpro components by means of component reuse.

**Benefits of Web Dynpro**

Web Dynpro’s main goal is to enable application developers to create powerful Web applications with minimum effort using descriptive tools in a structured design process. One guiding principle in the Web Dynpro philosophy is: The fewer lines of hand-written code, the better. Web Dynpro pursues this goal in two ways.

- Web Dynpro uses a declarative, language-neutral meta model for defining a user interface. From this abstract definition, the development environment generates the required source code. Hand-written code still has its place, but is confined to that required to manipulate the business data, not the user interface.
- Web Dynpro provides technical features such as support for internationalization, flicker-free interaction, and a clean separation of the business logic and the user interface. This separation is achieved through a modified implementation of the Model View Controller (MVC) design paradigm.

Since the repetitive tasks of UI coding have been eliminated, the developer can focus his attention on the flow of business data through the application. Web Dynpro applications can run on a range of devices and on various types of networks — an important feature for collaboration scenarios.

![Web Dynpro Benefits](image-url)
Web Dynpro components allow structuring complex Web applications and developing reusable, interacting entities. This enables the nesting of large application sections.

Web Dynpro components are containers for other entities related to the UI and the Web Dynpro program.

Entities related to the UI are windows and views. The layout of a view represents a rectangular part of a page displayed by the client (for example, a browser). The view contains UI elements such as input fields and buttons. The complete page sent to the client can be set up by only one view, but can also be a combination of multiple views. The possible combinations of views and flow between the views is defined in a window. A window can contain an arbitrary number of views. A view can be embedded in an arbitrary number of windows.

The Web Dynpro source code is located in Web Dynpro controllers. The hierarchical storage for the global variables of controllers is called the context.

Web Dynpro components can be addressed in three different ways:
- Using a Web Dynpro application, a Web Dynpro component can be related to a URL, which can be called from a Web browser or another Web Dynpro client.
- When reusing a Web Dynpro component as a sub-component, the visual interface of a Web Dynpro component can be combined with the visual entities of the main component to form the UI.
- When reusing a Web Dynpro component as a sub-component, all methods and data defined in the programming interface can be accessed by the main component.
Context Mapping and Data Binding

The variables defined in a Web Dynpro Controller context can be referenced from other Web Dynpro controllers. This is called **context mapping**. This allows to share common attributes between different controllers, so copying these attributes between the controller contexts is not necessary.

The value of UI elements, that allow a user input, have to be connected to context attributes of the corresponding controller. This is called **data binding**. Through data binding, an automatic data transport between the UI elements and the context attributes is established.

Combining these two concepts, the data transport between UI elements located in different views can be defined in a purely declarative way.
Context mapping allows a context node in one controller to be supplied automatically with data from a corresponding context node in another controller. This is the primary mechanism for sharing data between controllers. When two controllers within the same component share data through a mapping relationship, it is known as internal mapping. The context node that acts as the data source is known as the mapping origin node, and the context node that is mapped is known as the mapped node.

The mapping between controller contexts located in different Web Dynpro components is known as external mapping.

**Note:** External mapping will not be covered in this lesson.

For a mapping relationship to be established, the following steps must first be performed:

- A node must exist in the context of the controller acting as the mapping origin. This node need not have any declared child nodes or attributes.
- The mapping origin controller must not be a view controller.
- The controller containing the mapped node must declare the use of the mapping origin controller as a used controller.

![Diagram of ABAP Web Dynpro Context Mapping](image)

**Figure 7: Putting Data on the Screen: Data Binding**
**Data binding** is the means by which data is automatically transported from a view controller's context to a UI element in its layout, and visa versa. You may not bind UI elements to context nodes or attributes defined in some other controller. UI elements are private to the view controller within which they are declared. The data binding process decouples the UI element object from the application code within the view controller. Therefore, in order to manipulate UI element properties, the application code in the view controller needs only to manipulate the values of context nodes and attributes to which the UI elements are bound.

The Web Dynpro framework then manages the following two tasks:

- The transport of data from the context attribute to the UI element during the screen rendering process.
- Repopulating the context attribute from the UI element after data has been entered by the user and the next server round trip is initiated. Hereby, the values entered by the user are automatically converted and checked for type conformity. If an error occurs, a message is displayed.

Data binding is powerful, since not only the value of a UI element can be bound to a context attribute, but other UI properties like visibility can also be bound. This way, UI element properties can be manipulated from the view controller by acting on context attributes.

**Navigation Between Views**

Figure 8: Navigation Between Views

- To define the navigation between two views, you need to create exit and entry points for each view using outbound and inbound plugs.
- Only then can you specify the navigation flow using navigation links.
Navigation between views is triggered by firing **outbound plugs**. Firing an outbound plug causes a navigation event to be raised. Navigation events are special asynchronous events that are placed into a navigation queue. Multiple outbound plugs can be fired from one view. This can be used to define the next UI, consisting of multiple views. The navigation queue is processed at a certain point of time in the Web Dynpro processing phase. Up to this point of time, the navigation stack can be extended by firing additional outbound plugs, or the complete navigation stack can be deleted. Outbound plugs should be named according to the action that caused navigation away from the current view.

**Inbound plugs** are special event handler methods that subscribe to navigation events raised when outbound plugs are fired. Inbound plug methods are called only when the navigation queue is processed. This will only take place once the views in the current view assembly have fired their outbound plugs and no validation errors have occurred that would cause navigation to be cancelled. Inbound plugs should be named according to the reason for which the view is being displayed. Outbound and inbound plugs are joined together using **navigation links**.

Technically, linking an inbound plug to an outbound plug means registering the inbound plug event handler method to the navigation event, called by firing an outbound plug. Outbound plugs and event handler methods related to the inbound plug can have parameters, allowing you to pass data between the views.

**Windows and Nested Views**

A window is the set of all possible views that can make up a visible screen.

A window can have zero or more views embedded within it.

A view can have ViewContainerUlelements. This allows nesting views within a window.

A ViewContainerUlelement can only display one view at a time.

**Figure 9: Windows and Nested Views**

A **window** defines which views are displayed in which combination and how the view combination may be changed by firing outbound plugs. Thus, when creating a window, you define three things:
• All the possible views that could exist in the component’s visual interface must be embedded in the window.
• If multiple views must be displayed in parallel, the layout and position of these views is defined by a special view containing View Container UI elements in its layout. This container view is embedded in the window and, inside each area defined by a View Container UI element, all possible views for this view area are embedded (nested embedding). For each View Container UI element, there is one default view displayed at startup.
• The navigation links between the different views must be defined. Only one view can be displayed in the view area at a time. Navigation links must be defined between views in order for the contents of a view area to be replaced. View areas can be blanked out by creating an empty view whose inbound plug responds to an appropriate navigation event.

![View Assembly Diagram]

The subset of views visible at any one time is known as a **view assembly**.

Navigation causes either specific views within a view area to be replaced, or it can cause entire view combinations within the window to be replaced.

**Figure 10: View Assembly**

Outbound plugs are the triggers that cause a view area to contain a particular view. For a given window definition in which multiple views are embedded into view areas, there could be many permutations of views visible. The permutation that is visible depends on which navigation links the user follows.

The subset of views visible at any time is known as a **view assembly**.
SAP's Web Dynpro is built on the foundation of the **Model View Controller** (MVC) design paradigm. MVC was originally invented by the Norwegian software designer Trygve Reenskaug while working at Xerox PARC in the late 1970s. The first implementation of this design paradigm was with the release of the Smalltalk-80 programming language. MVC was a revolutionary design paradigm because it was the first to describe software components in terms of:

- The functional responsibilities each should fulfill
- The message protocols to which each component should respond

SAP has modified and extended the original MVC specification to create the Web Dynpro toolset.

**Web Dynpro Component Architecture**

**Figure 11: Model View Controller (MVC)**

**Figure 12: Internally Visible Web Dynpro Entities**
The architecture of a Web Dynpro component can be divided into two parts: **external and internal visibility**. The horizontal dashed line in the above figure separates the entities that are visible from outside the component from those that are only visible from within the component.

The internally visible parts can further be divided into visual entities and programming entities. Visual entities are those related to the UI, generated by the Web Dynpro framework, and passed to the client. The internally visible entities consist of windows and views.

A view consists of a view layout and the corresponding view controller. The view controller can contain navigation plugs, methods, and a context. A window embeds one or more views and has a corresponding window controller.

A window controller can contain navigation plugs, methods, and a context. Each view can be embedded in multiple windows.

The outbound plug of a window can be connected to any inbound plug of embedded views, and the outbound plug of a view can be connected to any inbound plug of the embedding window. However, navigation between windows of the same component is not possible.

---

**Figure 13: Internally Visible Web Dynpro Entities (2)**

The component controller acts as a component wide controller. Program logic related to only a certain view (forexample, checking user input) should be coded in the related view controller. Usage declarations between the controllers allow you to access the context data and methods of the declared controller (**used controller**). A view controller cannot be declared as a used controller for other controllers, since this would violate good programming (MVC programming paradigm). Business logic should not be part of the Web Dynpro component, but should be defined outside of the component to have high reusability. It is preferable that ABAP classes be used to encapsulate the related source code.
Custom controllers are optional controllers that must be defined by the developer. These controllers can be used to modularize the component content. For example, custom controllers can act as local controllers for some views, or they can be used to encapsulate the logic related to a certain model class (business logic). This allows you to reduce the content of the component controller by populating sub-functions.

If one Web Dynpro component (parent component) needs access to another Web Dynpro component (child component) the parent component can declare the use of the child component. A specific component usage instance is then created and the parent component accesses the functionality of the child component via its component interface controller.
The only parts of a Web Dynpro component that are visible to the user, are the **interface controller** and the **interface view(s)**.

- All Web Dynpro components have only one interface controller. Via the interface controller, data, methods, and event handlers can be exposed to other components.
- Interface views represent the visual interface of a Web Dynpro component. There is a one-to-one relationship between a Window and an interface view. Each time a window is defined, a related interface view is automatically generated, which makes the window accessible from outside the component. The interface view only exposes those inbound and outbound plugs to the component user that have the *interface* property checked. Methods and context data of the window are not accessible via the related interface view.
- If the component has no views, there is no need to have Windows. In this case, the component will not implement an interface view. Such components having no visual interface are known as **faceless components**.

- **An application is an entry point into a Web Dynpro component**
- **An application can be addressed via a URL**

![Diagram](image)

**Figure 16: Web Dynpro Application**

A **Web Dynpro application** is an entry point into a Web Dynpro component and is the only Web Dynpro entity that can be addressed via a URL. There is often (but not always) a one-to-one relationship between an interface view and an application. In the same way that the functionality in an ABAP module pool can be accessed by defining multiple transaction codes, the functionality of a single Web Dynpro component can be accessed by defining multiple applications, each addressing a different interface view and/or a different inbound plug of the interface view. In order to define a Web Dynpro application, you must specify:

- The component to be invoked; this component is then known as the **root component**
- Which interface view of the root component will be used; the default view(s) in
this interface view define(s) the default view assembly.

• Which inbound plug will act as the entry point for the nominated interface view (this inbound plug must be of type Startup)
Exercise 1: Web Dynpro Introduction

Exercise Objectives
After completing this exercise, you will be able to:
- Create a Web Dynpro component with a Web Dynpro window containing a single Web Dynpro view
- Place a simple UI element on the view layout and edit its properties
- Create a Web Dynpro application to make the view available for users

Business Example
You want to develop a Web Dynpro application. You will start by creating the Web Dynpro entities that are needed for an application that shows a view with a simple text field.

Template: n/a
Solution: VCT_INTR_S

Task 1:
Create a package that will contain all the repository objects you are going to develop.
1. Create package ZVCT_##. Assign application component BC-WD and software component HOME.

Task 2:
Create a Web Dynpro component with a Web Dynpro window.
1. Create Web Dynpro component ZVCT_INTR_## with a main window MAIN_WINDOW.
2. What other Web Dynpro entities have been created automatically in this step?

Task 3:
Create a Web Dynpro view with a simple TEXTVIEW element and make it part of the main window.
1. Create a Web Dynpro view (suggested name: MAIN_VIEW) inside your component.
2. Create a simple UI element of type TEXTVIEW (suggested name: TEXT_VIEW_1) on the view layout and edit its properties.
3. Maintain the text to be displayed in the TEXTVIEW element. You may want to change some other properties of the element and see the result on the layout preview.
4. Make the Web Dynpro view part of the Web Dynpro window.

Task 4:
Create a Web Dynpro application to access your Web Dynpro component and test it.
1. Create a Web Dynpro application (suggested name: ZVCT_INTR_##) that accesses the main window of your Web Dynpro component.
2. Activate the Web Dynpro component and test the Web Dynpro application.
Solution 1: Web Dynpro Introduction

Task 1:
Create a package that will contain all the repository objects you are going to develop.

1. Create package ZVCT##. Assign application component BC-WD and software component HOME.
   a) Perform this step like you (hopefully) have done often before.

Task 2:
Create a Web Dynpro component with a Web Dynpro window.

1. Create Web Dynpro component ZVCT_INTR## with a main window MAIN_WINDOW.
   a) In the navigation area, open the context menu for the package and choose Create WebDynpro WebDynpro Component (Interface).
   b) In the dialog box, enter the name of the component, a description, and the name of the main window.

2. What other Web Dynpro entities have been created automatically in this step?

   Answer:
   A component controller
   An interface controller
   An interface view for the window with its ABAP objects interface ZIWCI_VCT_INTR##

Task 3:
Create a Web Dynpro view with a simple TEXTVIEW element and make it part of the main window.

1. Create a Web Dynpro view (suggested name: MAIN_VIEW) inside your component.
   a) In the context menu for the Web Dynpro component choose Create View.
   b) Enter the name of the view and a short description.

Continued on next page
2. Create a simple UI element of type TEXTVIEW (suggested name: TEXT_VIEW_1) on the view layout and edit its properties.
   a) Edit your Web Dynpro view MAIN_VIEW and open the Layout tab.
   b) In the display of the UI element hierarchy (upper-right corner), open the context menu for ROOTUIELEMENTCONTAINER and choose Insert Element.
   c) Enter the name of the element and the element type: TEXTVIEW.
3. Maintain the text to be displayed in the TEXTVIEW element. You may want to change some other properties of the element and see the result on the layout preview.
   a) Double-click on the TEXTVIEW element in the UI element hierarchy.
   b) In the Properties list (below the UI element hierarchy) maintain the text in the text field.
4. Make the Web Dynpro view part of the Web Dynpro window.
   a) Edit the Web Dynpro window and open the Window tab.
   b) In the window structure, open the context menu for the Web Dynpro window and choose Embed View.
   c) In the dialog box, enter the name of the view.

**Task 4:**
Create a Web Dynpro application to access your Web Dynpro component and test it.
1. Create a Web Dynpro application (suggested name: ZVCT_INTR_##) that accesses the main window of your Web Dynpro component.
   a) In the navigation area, open the context menu for your Web Dynpro component and choose Create WebDynpro Application.
   b) In the dialog box, check the name of the application (same as component) and enter a description.
2. Activate the Web Dynpro component and test the Web Dynpro application.
   a) Open the Web Dynpro Component and activate it.
   b) Open the Web Dynpro Application and test it.
Unit 2

Web Dynpro Controllers

**Controller Types**
There are four types of controllers in an ABAP Web Dynpro component. These different controller types differ in the entities they are composed of:

**Component controller**
There is only one component controller per Web Dynpro component. This is a global controller, visible to all other controllers. The component controller drives the functionality of the entire component. This controller has no visual interface.

**Custom controllers**
Custom controllers are optional. They have to be defined at design time and can be used to encapsulate sub-functions of the component controller. Multiple custom controllers can be defined in a component. Custom controllers are instantiated automatically by the Web Dynpro framework and the instantiation order is undefined; therefore, the coding in a custom controller should not depend on the existence of any other custom controller.

**Configuration controller**
This is a special custom controller. It is only necessary if the corresponding component implements special configuration and personalization functionality. Only one configuration controller may exist in any component. Any other controller can access the configuration controller, but the configuration controller cannot access any other controller.

**View controllers**
Each view consists of the layout part and exactly one view controller. This controller cares for view-specific flow logic, like checking user input and handling user actions.

**Window controllers**
Each window has exactly one window controller. This controller can be used to care for the data passed via the inbound plugs when being reused as a child controller. Methods of this controller can be called from the inbound plug methods of the window.

At runtime, all controller instances are singletons in respect to their parent component. This is also true for view controllers; thus, embedding a view in a view assembly more than one time is not allowed.

The global data of a controller is stored in a hierarchical data storage, the **controller context**. This context and the methods defined in a controller are private unless another controller explicitly declares the usage of this controller. However, a view controller can not be declared as a used controller, so the context data and the methods of a view controller are always private.
Web Dynpro is a stateful implemented technology, meaning that the lifetime of controller instances is not limited to the time used to process the program code and to process the UI. Depending on the controller type, the controller instance lifetime is as follows:

**Component controller**
The lifetime of the component controller equals the lifetime of the component. When starting a Web Dynpro application, the component controller is instantiated by the Web Dynpro runtime.

**Custom controllers**
The instantiation of a custom controller is delayed until the first method of the controller is called. Custom controller instances cannot be deleted explicitly.

**Configuration controllers**
This controller is instantiated as the first controller of the component. It lives as long as the component lives.

**View controllers**
The instantiation of a view controller is delayed until the first method of the controller is called. The lifetime of a view controller can be controlled by the views properties:
- If *framework controlled* is selected, the view instance will be deleted with the component.
- If *when visible* is selected, the view instance is deleted as soon as the view no longer belongs to the view assembly.

**Window controllers**
The instantiation of a window controller is delayed until the first method of this controller is called. This is done by starting a Web Dynpro application or by embedding the related interface view in the parent component's window. Window controller instances cannot be deleted explicitly.

**Common Controller Entities**
Each controller has its own context. The context root node already exists. All other nodes and attributes have to be defined statically or by source code.
For all controllers, there exist methods that are called by the Web Dynpro framework in a predefined order. These are called **hook methods**. Depending on the controller type, there are different hook methods available. At least two hook methods are contained in all controller types. These methods are processed only once during the lifetime of a controller instance: when a controller instance is created (wddoinit()) and when a controller instance is deleted (wddoexit()).

**Additional methods** can be defined using the **Methods** tab.

**Attributes** not related to the value or property of UI elements can be declared using the **Attributes** tab. These attributes are then visible in all methods of this controller. There are two predefined attributes, which are used to access the functionality of the controller (WD_THIS) and of the context (WD_CONTEXT).

For information to be shared between different controllers, one controller must declare the use of another controller. This is done on the **Properties** tab of the controller that needs to access another controller. The most frequent requirement for this kind of data sharing is when you wish to create a mapped context node or you wish to access another controller's user defined methods.

You may not specify the name of a view controller as a used controller, as this would violate good MVC design principles. A view controller should be written such that it is responsible for nothing more than the display of data and user interaction. A view controller is not responsible for generating the data it displays; that is the role of a custom controller.

Business logic, such as function modules, BAPIs, or methods in helper classes can be accessed from the methods of all controllers.
Special Entities of Component / Custom Controllers

For both component and custom controllers, **events** can be defined with arbitrary parameters. Any method of any other controller (also view and window controllers) can register to these events if this method is defined as an event handler method. A typical use of such events is the invocation of processing in a view controller after processing in the component controller has been completed. This can be achieved when a method in the view controller subscribes to an event raised by the component controller.

Using your design time declarations, the Web Dynpro framework will automatically manage the definition, triggering, and handler subscription to such events for you. You also have the additional option of dynamic event subscription at runtime.

**Caution:** If two or more methods subscribe to the same event, the order in which they will be executed is undefined.

The component controller and **only** the component controller has additional standard hook methods. These are processed directly before the navigation requests are processed (\texttt{wddobefor navigation()} ) and after all views of the view assembly to be sent have been processed (\texttt{wddopostprocessing()}).

Attributes, methods, context elements, and events can be marked as interface elements. These elements are then exposed to other components by means of the interface controller.
Special Entities of View Controllers

A view controller is a visual building block of a Web Dynpro component and is designed to handle all aspects of data display and user interaction. For navigation to take place between different Web Dynpro view controllers, special navigation events and navigation event handlers have been created. These are called navigation plugs.

A navigation event is raised when an outbound plug is fired. Using the generic name `<Outbound plug>` for an outbound plug, its declaration will cause a method to be generated in the view's component controller, called `FIRE_<Outbound plug>_PLG`. This method is only visible for the Web Dynpro framework; it is not visible to the developer. An inbound plug is the navigation event handler that can be registered to a navigation request. Using the generic name `<Inbound plug>` for an inbound plug, its declaration will cause a method to be generated in the view's component controller, called `HANDLE<Inbound plug>`.

A static registration of an inbound plug (method `HANDLE<Inbound plug>`) to the navigation event raised by an outbound plug (method `FIRE_<Outbound plug>_PLG`) is called a navigation link. Navigation links are not part of a view but are defined in a window embedding the view. Thus, in different windows, the event registration can be defined differently.

An action links a client-side event (for example, pressing a button in a browser) to an event handler method defined in the corresponding view controller. When defining an action having the name `<Action>`, an event handler method (`ONACTION<Action>`) is automatically generated. If a view will be replaced as a result of a client event, the related outbound plug can be fired in this action event handler method.
There are two special hook methods found in view controllers:

- `wddobeforeaction()` is processed if a client event is fired in any view. Before the action event handler methods are processed, the `wddobeforeaction()` methods are executed for all views that are part of the last view assembly.

- `wddomodifyview()` is a method that allows you to programmatically manipulate the layout of the view. This event can be used to define the UI dynamically.

In addition to the standard attributes of all controllers, a view controller has a reference to the component controller of its component: `WD_COMP_CONTROLLER`. This reference can be used to access functionality related to the component controller. It is used when accessing messages or when calling a method declared in the component controller.

### Special Entities of Window Controllers

Window controllers are very similar to view controllers. Technically, it is like a view controller without a UI (view layout). All views that are to be displayed when using a Web application have to be embedded in the window that is referred to by the application or the calling component.

![Figure 20: Window Controller Architecture](image)

The Web Dynpro window contains the structure of all views to be displayed and (if statically defined) the navigation links defining the possible view assemblies. Each Web Dynpro window contains outbound plugs and inbound plugs, like views. You can use the plugs to set up cross-component navigation. To expose the plugs to the component interface, select the property `Interface` for each plug. These plugs will then be part of the related interface view.

The window controller also has the special attribute `WD_COMP_CONTROLLER`, which holds the reference to the component controller.
Exercise 2: Web Dynpro Controllers

Exercise Objectives
After completing this exercise, you will be able to:
• Provide buttons on Web Dynpro views
• Handle the pressing of such a button by the user
• Implement navigation between Web Dynpro views

Business Example
You want to develop a Web Dynpro application with more than one view. On your views, you wish to provide buttons to allow the user to initiate navigation to the next view.

Solution: VCT_CTRL_S

Task 1:
Create a Web Dynpro component with a Web Dynpro window and two Web Dynpro views.
1. Create Web Dynpro component ZVCT_CTRL_## with a main window AIN_WINDOW.
2. Create two Web Dynpro views (suggested names: INPUT_VIEW and OUTPUT_VIEW) inside your component and embed them into your window.

Task 2:
Create an outbound plug and an inbound plug for each view. Define two navigation links between the two views: each going from the outbound plug of one view to the inbound plug of the other view.

1. For your first view (INPUT_VIEW), create an outbound plug (suggested name: TO_OUTPUT_VIEW) and an inbound plug (suggested name: FROM_OUTPUT_VIEW).

Continued on next page
2. What new parts have been created in the view controller by completing the previous step (refer to the various tabs of the view editor)?

3. Do the same for your other view (suggested names for the plugs: TO_INPUT_VIEW and FROM_INPUT_VIEW).

4. Define a navigation link from the outbound plug of the first view (TO_OUTPUT_VIEW) to the inbound plug of the second view (FROM_INPUT_VIEW).

5. Define a navigation link from the outbound plug of the second view (TO_INPUT_VIEW) to the inbound plug of the first view (FROM_OUTPUT_VIEW).

Task 3:
Provide a button on each of the two views and make sure the outbound plug of the respective view is fired when the button is pressed by the user.

1. Create a simple UI element of type \textit{BUTTON} (suggested name: BUTTON\_GO) on the first view and maintain the text to be displayed on the button.

2. Create an action for the button (suggested name: GO) and relate it to the outbound plug of the view (TO\_OUTPUT\_VIEW).

3. What new parts have been created in the view controller by the previous step (refer to the various tabs of the view editor)?

4. Create a button on the second view (suggested name: BUTTON\_BACK) with text and action (suggested name: BACK), and link to the outbound plug of that view.

5. Create a Web Dynpro application for your component, activate, and test.
Solution 2: Web Dynpro Controllers

Task 1:
Create a Web Dynpro component with a Web Dynpro window and two Web Dynpro views.
1. Create Web Dynpro component ZVCT_CTRL_## with a main window MAIN_WINDOW.
   a) Perform this step as in the previous exercise.
2. Create two Web Dynpro views (suggested names: INPUT_VIEW and OUTPUT_VIEW) inside your component and embed them into your window.
   a) Perform this step as in the previous exercise.

Task 2:
Create an outbound plug and an inbound plug for each view. Define two navigation links between the two views — each going from the outbound plug of one view to the inbound plug of the other view.

1. For your first view (INPUT_VIEW), create an outbound plug (suggested name: TO_OUTPUT_VIEW) and an inbound plug (suggested name: FROM_OUTPUT_VIEW).
   a) Edit Web Dynpro view INPUT_VIEW.
   b) Select the Outbound plugs tab and enter the outbound plug name and a description in the topmost table.
   c) Select the Inbound plugs tab and enter the inbound plug name and a description.
2. What new parts have been created in the view controller by completing the previous step (refer to the various tabs of the view editor)?

   Answer:
   Event handler method —HANDLEFROM_OUTPUT_VIEW“ (empty).

3. Do the same for your other view (suggested names for the plugs: TO_INPUT_VIEW and FROM_INPUT_VIEW).
   a) Perform this step as before.

Continued on next page
4. Define a navigation link from the outbound plug of the first view (TO_OUTPUT_VIEW) to the inbound plug of the second view (FROM_INPUT_VIEW).
   a) To edit the Web Dynpro window, choose the Window tab and expand all nodes of the window structure.
   b) Drag the outbound plug of the first view and drop it on the inbound plug of the second view. Confirm the data in the dialog box.
5. Define a navigation link from the outbound plug of the second view (TO_INPUT_VIEW) to the inbound plug of the first view (FROM_OUTPUT_VIEW).
   a) Perform this step as before.

Task 3:
Provide a button on each of the two views and make sure the outbound plug of the respective view is fired when the button is pressed by the user.
1. Create a simple UI element of type BUTTON (suggested name: BUTTON_GO) on the first view and maintain the text to be displayed on the button.
   a) Edit the view and choose the Layout tab.
   b) Open the context menu for ROOTUIELEMENTCONTAINER and choose Insert Element.
   c) Enter the name of the button and choose element type BUTTON.
   d) Go to property text and maintain the text to be displayed in the value column.
2. Create an action for the button (suggested name: GO) and relate it to the outbound plug of the view (TO_OUTPUT_VIEW).
   a) Go to property OnAction of the button. Choose Create in the Binding column.
   b) In the dialog box, enter the name of the action, a description, and the name of the outbound plug.
3. What new parts have been created in the view controller by the previous step (refer to the various tabs of the view editor)?

Answer:
Action —GO
Event handler method —ONACTIONGO (with coding to fire the outbound plug)

4. Create a button on the second view (suggested name: BUTTON_BACK) with text and action (suggested name: BACK), and link to the outbound plug of that view.
   a) Perform this step as before.
5. Create a Web Dynpro application for your component, activate, and test.
   a) Perform this step as in the previous exercise.
Unit 3

The Context at Design Time

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

Figure 21: The Context: The Heart of a Controller
The structure (that is, the meta data) of a context will typically be defined at design time. However, at runtime, it is possible not only to modify the contents of the context, but also to modify its structure itself.

**Figure 22: Changing the Context**

Unless declared differently, all runtime data within a controller's context is private to that component.
Information held in the context of a custom controller can be made easily 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. This is the primary mechanism for sharing data between controllers within a single component. It is not possible for a view controller to share its context data.

---

**Figure 23: Creating New Context Elements**

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 **context root node**. The root node is created automatically when the controller is initialized and always has fixed properties. The context root node can not be deleted or modified in any way. A **context node** is the main abstraction class used for runtime data storage within the Web Dynpro framework. Context nodes are arranged hierarchically. The node 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 can then be thought of as a collection of such elements in the same way that a table is a collection of rows.

**Caution:** The name of a context node must be unique within the complete structure of a controller's context.

A **context attribute** is an 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.

If a collection of structured data objects should be stored in a controller context at runtime, a context node must be defined to store the collection itself. Context attributes or other context nodes must be defined as sub-elements, storing the elements of each structured data object.
Context Element Properties

**Figure 24: Context Structure at Design Time**

**The Collection Cardinality**

At design time, you create the metadata structure within which the runtime data will be stored. The diagram above gives the impression that the context structure is a flat, two-dimensional tree, much like the display of directories and files shown in Windows Explorer. However, all context nodes are collections, so there could potentially be multiple instances of each child node and attribute within a nodes collection.

Every context node has a property called *cardinality*. This property is composed of two values that, taken together, describe the maximum and minimum number of elements the node collection may hold at runtime:

Cardinality minimum: 0 or 1
Cardinality maximum: 1 or n
Therefore, there are four possible cardinality values (specified as $<\text{Min}>..<\text{Max}>$):

- **0..1**: Zero or one element permitted
- **0..n**: Zero or more elements permitted
- **1..1**: Exactly one element permitted
- **1..n**: One or more elements permitted

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

**Figure 25: Context Structure at Runtime: Cardinality Property (1)**

The context root node always contains exactly one element (notice the cardinality is **1..1**). This is known as the default element and it cannot be deleted! This also implies that any node collection being a direct child of the context root node does exist at runtime as an empty collection ($\text{cardinality} = 0..<\text{something}>$) or with one default element ($\text{cardinality} = 1..<\text{something}>$). These child nodes of the context root node are therefore called **independent nodes**. All other node collections exist only if their parent collection contains at least one element. The existence of these node collections at runtime is therefore not guaranteed. Thus, these nodes are called **dependent nodes**.

If you attempt to perform any action on a node collection that would violate its cardinality, you will get a runtime error from the Web Dynpro framework. Such actions would be:

- Trying to delete the default element from a node of cardinality $1..<\text{something}>$
- Trying to add a second element to a node of cardinality $<\text{something}>..1$
Figure 26: Context Structure at Runtime: Cardinality Property (2)

You can now see that the structure of the context at runtime will not be the flat, two-dimensional hierarchy seen at design time. Instead, a context node takes on depth.

This can be compared to an internal table in ABAP at design time and runtime. At design time, the metadata of the internal table comprises the structure given by the line type. The dimension of the internal table (how many lines the data object will hold at runtime) is not part of the definition. At runtime, multiple lines can be appended to the internal table data object (as multiple elements can be appended to a collection originating from a context node at runtime). However, since there are no restrictions on the minimum and maximum number of lines an internal table can hold, there is no counterpart for the cardinality property. Internal tables always have zero lines after the declaration of the object, and they can hold an unrestricted number of lines at runtime.

The Lead Selection

A node's element collection can be accessed using a 1-based index value. Exactly one element of the node collection can be marked as the element at lead selection. The lead selection of a context node points to either a single selected node element (value of the lead selection = number of the selected node element) or, if no element is selected, it has the value of the constant IF_WD_CONTEXT_NODE=>NO_SELECTION. The lead selection can be set automatically by the Web Dynpro framework if the context node property Initialize Lead Selection is set to true. In this case, the first element in a collection will automatically be marked as the element at lead selection. The lead selection can also be set by program source code or it can be set by user actions related to UI elements (e.g. mark line in table view element, which is bound to the node).
If the lead selection is set, the following is true:

- In the controller code, special methods can be used to access the lead selection of a node, as well as its position.
- UI elements such as input fields can be bound to the attributes of this element.

**The Singleton Property**

![Diagram of Context Structure at Runtime: Singleton Property](image)

**Figure 27: Context Structure at Runtime: Singleton Property (1)**

Notice that the context node FLIGHTS has a child node called BOOKINGS. The BOOKINGS node is a distinct node with its own element collection.

The Boolean property *Singleton* critically affects the relationship between a dependent node and its parent node!

If the BOOKINGS node has its singleton property set to false (node BOOKINGS at runtime will be a **non-singleton**), then for every element in the parent node collection (FLIGHTS, in this case), there will be a distinct instance of the child node BOOKINGS.

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

Therefore, if there are *n* elements in the parent node, then there will be *n* distinct instances of a non-singleton child node.
Figure 28: Context Structure at Runtime: Singleton Property (2)

If the node BOOKINGS now has its \textit{Singleton} property set to true (which is the default), it does not matter how many elements are present in the parent node collection (FLIGHTS, in this case). There will only ever be one instance of the child node BOOKINGS, so the BOOKINGS collection will be a singleton at runtime.

\textbf{Supply Function}

In the above example, there could be many different elements in the FLIGHTS node collection. However, since there is only ever one instance of the singleton child node BOOKINGS, we need to ensure that when this child node is accessed, it contains the correct data for the selected element in the parent node.

Figure 29: Repopulating a Dependent Singleton Node: Supply Function
Supply functions are a mechanism to repopulate child nodes. A supply function can be assigned to each context node of a controller. This supply function is called by the runtime automatically when the data of an invalid context node is accessed. Invalidation of a context node occurs under the following conditions:

- The node collection is initial.
- The lead selection in the parent node collection is changed.
- The node collection is invalidated programmatically.

The supply function is especially useful in combination with singleton nodes. The values of child node elements of the type Singleton depend on the element of the parent node to which the lead selection is currently assigned. If the lead selection is changed by the user, the supply function can access the new lead selection element and recalculate the values of the child node elements accordingly.

The name of a supply function can be entered when defining or changing a context node from its context menu. As a result, an additional controller method is generated with a given signature:
- A reference to the element at lead selection in the parent node allows access to the attributes of this element.
- A reference to the node the supply function is related to allows you to store the dependent data in the child nodes collection.

Figure 30: Context Structure: Singleton Nodes

When a typical business transaction is being used, information will often be presented in the form of a list of header records of some sort (for instance, sales order headers). From this list, the user will typically select one order header and then look at its line items. It makes much more sense to read the line item information only when it is needed, rather than to read all the line items for all the sale orders in the hope that the user might want to look at the information! This would be a highly inefficient application architecture both in terms of processing requirements and memory usage.
Web Dynpro follows the principle of **lazy data instantiation**. This means that data is only created when it is actually needed. When this principle is applied to the context architecture, it means that unless the program actually needs to look at the data in a child node, the child node will remain unprocessed. Hence, there is no need to hold multiple instances of data the user has not requested.

Lazy data instantiation also means that dependent collections are not automatically created for all elements of the parent node. The creation of a collection of the dependent node is delayed to the point when the related element of the parent collection gets the lead selection.

**Context Mapping**

![Diagram of Context Mapping](image)

1. Drag and drop context node to target node.

2. The mapped node appears as child node.

In the property of the node, the mapping path is visible.

**Figure 31: Context Mapping**
Context mapping allows a controller (typically a view controller) to access data that has been pre-processed by some other controller. Since a mapping relationship allows a direct reference to be made to data in another controller, there is no need for this data to be copied or moved.

Note: A mapped node ceases to maintain its own element collection! Instead, it references the element collection of the mapping origin node. A view controller can not act as a data source for a mapping relationship because this would violate the principles of MVC program design.

Hint: In programs designed according to the MVC principle, a view controller should be written in such a way that it takes no part in the generation of the data it displays. Instead, the view controller should only be concerned with displaying pre-generated data and then handling the resulting user interaction (validation, error handling, and so on). It is the custom controller's job to interact with back-end logic to generate the required data.

Once the data has been generated, the custom controller then supplies it to a view controller for display. If a view controller were allowed to act as the data source (origin) for a mapping relationship, then a situation could be created in which a custom controller is dependent upon functionality found in a view controller. This would be bad MVC design and is therefore not permitted.

To establish the context mapping between two controllers, the target controller has to declare the source controller as a used controller in its properties. If the Context tab is chosen in the edit mode of the target controller, the contexts of all used controllers are displayed on the right side. A node of a source context can then be referenced as follows:

- If an independent node to be mapped is not defined yet in the target controller's context, the source node structure is mapped by dragging and dropping it from the source controller context to the root node of the target controller context.
- If the structure of a node is already defined in the context of the target controller, the target node is dragged and dropped from the context of the target controller to the mapping origin in the source controller context.

However, for dependent nodes this is only possible if the related independent nodes are already mapped.
Caution: Nodes can only be mapped if the mapped node does not contain more attributes than the mapping origin!

Hint: Having established the mapping between two nodes, all of their elements are mapped to both attributes and child nodes.

If the node structure of the mapping origin changes (elements are added, deleted, modified), the mapping can be updated (synchronized) using the context menu of the mapped node.

Figure 32: Context Mapping: Internal Mapping

Internal mapping is the name given to a mapping relationship in which both the mapped node and the mapping origin node lie within the boundaries of one component. This means that the mapping relationship can be fully established when writing the current component.

There is however, one special case where a mapping relationship can only be partially established when writing the current component. This is known as external mapping. As the name implies, this is a situation in which the mapping origin node lies outside the boundaries of, or is external to, the current component. External mapping will be discussed later.
Exercise 3: The Context at Design Time

Exercise Objectives
After completing this exercise, you will be able to:

- Create nodes in contexts of components and views
- Map view context nodes to component context nodes
- Create input/output fields on Web Dynpro views
- Bind UI elements to view context nodes

Business Example
You want to develop a Web Dynpro application with input/output fields on the views. The data the user enters on one view (a carrier ID and a connection ID) should be available as output on the next view.

Template: VCT_CTRL_S
Solution: VCT_COND_S

Task 1:
Copy your solution from the previous exercise or the template component.

1. Copy your solution from the previous exercise (ZVCT_CTRL_##) or the Web Dynpro component VCT_CTRL_S to the new component, ZVCT_COND_##.

Task 2:
In the context of the component controller, create a context node that contains attributes for a carrier ID and a connection ID. Create the same context node in each of the two views and map the view controller contexts to the component controller context.

1. In the context of the component controller, create a context node (suggested name: FLIGHTINFO) based on dictionary type SFLIGHT and with cardinality 1...1. Add two attributes from the structure components CARRID and CONNID.
2. Create the same node in the context of each view controller and map the contexts node of the view controllers to the context node FLIGHTINFO of the component controller.

Task 3:
For both views, create input fields corresponding to the context node attributes and bind them to these attributes.

1. On the layout of the view INPUT_VIEW, create a form with reference to the context attributes CARRID and CONNID. Use the Web Dynpro Code Wizard to create the form.
2. Repeat the previous step for the view OUTPUT_VIEW.
3. Create a Web Dynpro application for your component, activate, and test.
Solution 3: The Context at Design Time

Task 1:
Copy your solution from the previous exercise or the template component.
1. Copy your solution from the previous exercise (ZVCT_CTRL_##) or the Web Dynpro component VCT_CTRL_S to the new component, ZVCT_COND_##.
   a) One possible solution: In the object navigator choose Other objects and select the Web Dynpro component to copy and choose Copy.
   b) Enter a name for the new Web Dynpro component.

Task 2:
In the context of the component controller, create a context node that contains attributes for a carrier ID and a connection ID. Create the same context node in each of the two views and map the view controller contexts to the component controller context.

1. In the context of the component controller, create a context node (suggested name: FLIGHTINFO) based on dictionary type SFLIGHT and with cardinality 1...1. Add two attributes from the structure components CARRID and CONNID.
   a) Edit the component controller and choose the Context tab page.
   b) Open the context menu of the root context node and choose Create Node.
   c) Enter the node name, the name of the dictionary type (structure), and the cardinality.
   d) Choose Add Attribute from Structure, mark the structure components CARRID and CONNID, and choose .

2. Create the same node in the context of each view controller and map the contexts node of the view controllers to the context node FLIGHTINFO of the component controller.
   a) Edit one of the Web Dynpro views and choose the Context tab page.
   b) Drag the context node FLIGHTINFO from the component controller context (right-hand side) and drop it on the root node of the view controller context (left-hand side). Confirm that you want to copy and map the node.
   c) Repeat this step for the second view.

Continued on next page
Task 3:
For both views, create input fields corresponding to the context node attributes and bind them to these attributes.

1. On the layout of the view INPUT_VIEW, create a form with reference to the context attributes CARRID and CONNID. Use the Web Dynpro Code Wizard to create the form.
   a) Edit the Web Dynpro view and choose the Layout tab.
   b) Start the Web Dynpro Code Wizard by clicking the button in the Web Dynpro Explorer bar. Select the template for generating forms.
   c) Choose Context and, in the dialog box, double-click the context node (FLIGHTINFO).
   d) Confirm the entries by choosing .

Hint: You can rearrange the UI elements as follows:
- Dragging and dropping the button to the UI element group will make it the last element in the group.
- Setting the LayoutData button's property to MatrixHeadData will move the button to a new line in the group.
- Setting the text property of the group's caption will add a caption to the group.

2. Repeat the previous step for the view OUTPUT_VIEW.
   a) Perform this step as before.
   b) If you want to change the read-only property of the input fields of this view toggle the readOnly checkbox for each input field.

3. Create a Web Dynpro application for your component, activate, and test.
   a) Perform this step as in the previous exercises.
Unit 4

Defining the User Interface

Defining a View Layout

![Figure 33: UI Elements](image)

A **UI element** is any graphical entity that occupies a position within a view layout. However, that does not mean that all UI elements are visible on the screen. There are certain UI elements that are not visible on the screen, such as the `TransparentContainer` or the `ViewUIElementContainer`. These elements are used to structure the UI without being visible, but they occupy a position in the UI element hierarchy just like any other visible UI element. In addition, all UI elements can be set to invisible at runtime without freeing the space they would occupy as UI elements being visible. For example, hiding the `Label` UI element located left of an `InputField` UI element does not automatically mean that the `InputField` UI elements moves left and appears at the position where the `Label` UI element was previously displayed!

Web Dynpro has been designed to operate with any form-based user interface. This is probably going to be a standard browser, but it could also be a Smart Client, a Pocket PC™, or a Blackberry™ in a future release. This implies that any UI element is only an abstract description of the source code to be rendered at runtime, and not just an HTML or WML representation of this element.
There are numerous user interface elements available for designing the user interface of a Web Dynpro application. All available user interface elements are divided into categories, which are displayed in the view designer, if the layout preview is visible.

The **Standard Simple** category contains elements that are used frequently in Web Dynpro applications and that are also known from other UI technologies like **Button**, **Label**, or **InputField**.

The **Standard Complex** category contains elements that need to have child elements to define a valid, renderable UI element. A good example is a **Table**, which needs to have a **TableColumn** element as a child for each column to be displayed.

The **Standard Container** category comprises elements that may have child elements. Container UI elements structure the layout visibly (for example, **Group**) or non visibly (for example, **TransparentContainer**).

Other categories contain elements to display ActiveX-based diagrams (**Active Component**), Adobe interactive forms (**doe**), or business graphics rendered by the Internet Graphics Server (**BusinessGraphics**), or to embed office documents like Microsoft Word or Excel documents (**OfficeIntegration**) and some other special UI elements.
Figure 35: Arrangement of UI Elements

All view layouts are composed from a hierarchy of UI elements. The root node is always of type `TransparentContainer`, and is always called `RootUIElementContainer`. You cannot change this, it is hard coded! All subsequent UI elements added to a view layout are hierarchically subordinate to `RootUIElementContainer`.

Container Elements and Layout Managers

Container elements are UI elements that may have child elements. They occupy a rectangular area in a view's layout. All UI elements that are children of a container element are located in this rectangular area. All container elements define how their children will be arranged. This is done by evaluating the `Layout` property which assigns a layout manager to the UI element. All child elements of a container UI element inherit a set of properties that are related to the value of the container's `Layout` property section `Layout Data`. The `Layout` property may have four values:

- `FlowLayout`
- `RowLayout`
- `MatrixLayout`
- `GridLayout`
Figure 36: Layout Managers: FlowLayout

The default layout manager is the FlowLayout layout manager. All child attributes of this container will be displayed in a row, as long as the container is wide enough. If the container UI element is too narrow for all child elements to be displayed in one row (for example, the browser window is narrowed), they will be automatically wrapped to the next line(s). This wrapping cannot be forced at design time. Elements in different lines are not related to each other. This kind of container can be used to arrange sub-containers.

Figure 37: Layout Managers: RowLayout
If the `RowLayout` layout manager is used with the container UI element, all children inherit the property `LayoutData`, which can have the values `RowData` and `RowHeadData`. By setting this property to `RowHeadData`, a line break is forced. If you set the property to `RowData`, this child element will appear in the same line as the previous element, even if the right-hand margin is reached. UI elements located in different rows are not related to each other and thus are not aligned in columns. The width of each cell can be set by the `width` attribute of each child element.

![Image of ABAP Web Dynpro](image)

**Figure 38: Layout Managers: MatrixLayout**

If the `MatrixLayout` layout manager is used with the container UI element, all children inherit the property `LayoutData`, which can have the values `MatrixData` and `MatrixHeadData`. By setting this property to `MatrixHeadData`, a line break is forced. When the property is set to `MatrixData`, the child elements will appear in the same line as the previous element, even if the right-hand margin is reached. The child elements in this container are arranged in columns. Using this layout manager, the number of columns is not defined statically, but it is defined by the maximum number of child elements in any row. The number of elements in different rows do not have to match. UI elements arranged in a `MatrixLayout` can occupy multiple cells `colSpan` (property).
Figure 39: Layout Managers: GridLayout

Like the MatrixLayout, the GridLayout layout manager can be used if a vertical alignment of the elements is desired. However, here the number of columns is defined via the colCount property of the container element. Thus, the single child element does not control whether it is the first element of a new row. A line break will take place when all cells of a row are occupied. If an element is removed from the hierarchy, the complete arrangement will change since all elements following in the hierarchy will move as many cells to the left as were occupied by the removed element.

This kind of layout manager should be used if all rows occupy the same number of columns and if only complete rows are inserted or deleted. When using this layout manager, UI elements should not be removed completely but replaced by an InvisibleElement in order to retain the original element arrangement.

Adding UI Elements to the Layout

The View Editor is a Web-Dynpro-specific tool that allows you to edit a view layout. The View Editor is only available when you are editing a view controller. It will not appear when you edit a custom controller because these controllers have no visual interface.
Figure 40: Using the View Editor

The view editor can be used with or without the layout preview. The following description assumes that you work with the layout preview.

Figure 41: View Editor
To add any UI element, you can drag and drop it from the toolbar on the left-hand side of the View Editor. Adding a new UI element in the hierarchical representation is also possible. This can be done from the context menu of any element in the hierarchy that can have child elements (for example, a TransparentContainer element or a Group element).

To change the position of a UI element in the element hierarchy, you can move it up or down using the related functionality from the element's context menu. The element can also be moved by dragging and dropping it to the new position in the hierarchical representation or in the layout preview.

The Properties tab displays all properties of a selected UI element. If a UI element supports events, the supported client-side events are listed in the Events properties section. Properties related to client-side events begin with the prefix on (for example, onFilter, onSort, or onAction). To handle these client-side events, actions have to be associated with each of the events.

**Hint:** Client-side events are events related to UI elements that are predefined by the Web Dynpro framework. It is not possible to handle additional events at the client side (for example, by using JavaScript).

**Data Binding**

Once a UI element property is bound to a context node or attribute, the context data is used to supply a value to the UI element property. If the UI element property is one that the user can update (such as the value property of an InputField UI element), the context is automatically updated with the new value during the next round trip.

Almost all of the properties of a UI element can be bound either to a context node or to a context attribute having the correct data type.

**Note:** A binding relationship can only exist between the context and UI elements of the same view controller.

The Web Dynpro view controller has been constructed in such a way that it is usually possible to have full control over the appearance of the screen layout without ever needing direct access the UI element objects. Any property over which you wish to have programmatic control should be bound to an appropriate context node or attribute.
Defining the Data Binding

For a UI element to display any data, at the very least its value property must be bound to an appropriate context node or attribute (there are very few UI elements that do not have a value property like \textit{HorizontalGutter} for instance). The following steps are the minimum requirement for putting some data on the screen:

1. Create a node or attribute in the view controller's context that will contain the data. It is not important whether this is a mapped context node or not.
2. Create the UI element in your view layout.
3. For all properties requiring a context binding, a button with a yellow icon and an empty circle is displayed on the right hand-side of the property. Assign the required binding by clicking on this button. A dialog box will be displayed showing the view controller's context. All nodes or attributes of the appropriate type to be bound to the UI element property are displayed. Select an appropriate node or attribute.

The context path to the node or attribute will then be displayed as the property's value. In addition, the empty circle will be replaced with a green check mark icon. The UI element on the layout preview will also display the context path of the node or attribute to which it is bound.

The establishment of a binding relationship instructs the Web Dynpro screen renderer to obtain the value for a UI element property from the context node or attribute to which it is bound.
Context binding is not limited to simply supplying an `InputField`, say, with a value. A UI element’s value property is just one of the many properties that can be supplied with data through a binding relationship.

This is the mechanism by which a view controller can adjust the appearance and behavior of its view layout without ever needing to access the UI element objects themselves.

![Figure 43: Putting Data on the Screen (1)](image)
As you can see in the figure above, there is no requirement to have the UI elements in the same order as the context attributes to which they are bound. Data binding is a **two-way** relationship.

Once a binding relationship has been declared, the data in the bound nodes and attributes are transported automatically to the corresponding UI elements. After the user has interacted with the screen and initiated an HTTP round trip, the new or modified data in the UI elements is transported back to the same nodes and attributes in the view controller’s context. By the time the Web Dynpro framework hands over control to your action handler, the context already holds the updated information.

This two-way transport process is entirely automatic, and requires no action on the part of the application developer. All you need to do is declare a binding relationship.
Controlling the UI Element Behavior

Figure 45: Defining UI Element Properties Statically

The value of a UI element property can either be hard coded or it can be bound to a context attribute of a suitable data type. If a property value is hard coded at design time, it can only be changed at runtime by accessing the UI element directly from the view controller's source code. This can only be done from the hook method `wddomodifyview()` , since only this method provides a reference to the UI element hierarchy. However, accessing the UI element hierarchy directly from a controller's method is considered poor design because the separation between flow logic and UI is no longer any more. This technique should be avoided!
In order to programmatically control the behavior of a UI element, you should create a context attribute having a data type that fits to the property you wish to control. This allows you to control the behavior of the UI element by modifying the related attribute value in any method of any controller that has access to this context attribute. Directly accessing the UI element object from the controller source code is no longer necessary.

Once the context attribute has been created, it must be bound to the appropriate UI element property. In this case, the `readOnly` property of an `InputField` UI element has been bound to a Boolean value attribute.
The value of the context attribute can now be manipulated from any controller hook method, or from your additionally defined methods. This technique is applicable to most of the UI element properties. Appropriate data types can be found using the *Types of the Web Dynpro Runtime* tab on the dialog box displayed when typing a context attribute.

**Texts from the ABAP Dictionary**

Many UI elements (TextViews, Labels, Captions, and so on) display texts in the rendered UI. These texts can be obtained from the ABAP Dictionary in two ways:

- The property related to the text is bound to a context attribute, which itself is typed with a data element defined in the ABAP Dictionary. Example: The `Text` property of a Button is bound to a context attribute. Then the button text originates from the data element related to the context attribute.
- The UI element is related to a second UI element, and this second element is bound to a context element being typed with a data element. In this case, the property related to the text must be left blank in order to use the dictionary text. Example: A Label is related to an InputField and the `Text` property of the Label is left blank. The label text then originates from the data element related to the InputField.

**Composite UI Elements**

Certain UI elements are displayed on the screen as aggregations of simpler, more basic, UI elements. The Table UI element is a good example.

*Composite UI elements are incapable of displaying information on their own.*

*They must have child UI elements in order to function correctly.*

![Hierarchical representation of a Table UI element](image)

*Fig. 48: Composite UI Elements (1)*
Without the subordinate, or child, UI elements, a composite UI element is not capable of displaying any information. Composite UI elements such as Group and Tray have a mandatory Caption child UI element, but beyond that, their structure is entirely user defined. Composite UI elements such as Table and Tree, however, require a more complex mandatory child structure.

**Figure 49: Composite UI Elements (2)**

### The Table UI Element

The Table UI element acts as a parent of several TableColumn UI elements; each of which acts as the parent for a header (implemented by a Caption UI element) and a cell editor. cell editor is an abstract expression for all kinds of UI elements that can serve as cell elements in a given column. The default for the cell editor is the TextView UI element. However, depending on the cell value and the necessity that the cell value should be changeable, other UI elements can be used as cell editors (for example, InputField, DropDownListByKey, Checkbox, or Button).

**Hint:** For SAP NetWeaver Application Server having installs a support package = 12, scrolling in a table is possible by setting the application parameter WDTABLENAVIGATION to SCROLLBAR.
The Table is an example of a composite UI element. A Table's child elements are also composite.

Figure 50: The Table UI Element

Binding a Table to the Context
The Table UI element allows a two-dimensional display of data in cells arranged in rows and columns. The UI element consists of an optional header area, zero or more rows, and a footer area. The Table UI element must be bound to a context node of cardinality 0..n or 1..n. The element at the lead selection in the context node becomes the highlighted row when displayed on the screen (this does not depend on having a selection column).
The Table UI element with its child elements can be created using the Web Dynpro Code Wizard. In this case defining the complete context binding is part of the wizard process. However, if the Table UI element has been added to the element hierarchy manually, the context menu entry Create Binding should be used to create the complete binding. The caption appearing at the top of the table (Flights in the figure below) is optional.

![Diagram showing context hierarchy and table design time and runtime](image)

**Figure 51: Binding a Table UI Element to the Context**

**TableColumn UI Elements**
The Table UI element must contain at least one TableColumn UI element. TableColumn UI elements should be bound to the node attributes to which the parent Table UI element is bound. Any node attribute of a simple data type is a candidate for becoming a table column.

**Hint:** A TableColumn UI element can be bound to an attribute that is not a child of the node to which the Table UI element is bound. In this case, the values of all cells in the column are equal.

TableColumn UI elements are composite UI elements! This means that they must have child UI elements in order to function correctly.

The column header is created using a Caption element. The text displayed in the column header can be obtained from the ABAP Dictionary if the Text property of the Caption UI element is left blank and if the context attribute displayed in this column is typed with an ABAP Dictionary data element.
Figure 52: TableColumn UI elements

Child UI Elements of a TableColumn
To present information to the user, a TableColumn UI element must have a child UI element that will act as the cell editor. In order to decide which UI element to use, the kind of interaction between the user and the data in each column has to be clarified.

Do not think that because the name of this role contains the word —editor“ that the user can necessarily change the data. If you select some sort of display-only UI element such as a TextView, then the user will not be able to modify the data; the UI element you chose does not allow it. If you choose a Button, then the cell value is displayed as the button's text. Here, the user cannot input data, but he can fire a client event. If, on the other hand, you chose an InputField to be the table cell editor, then all the cells appearing in that column would, by default, be open for input.

The caption appearing as the column header is optional, but if defined will always be of type Caption.
Figure 53: Defining Child UI Elements for a TableColumn

Selecting a Table Row and the Effect on the Node's Lead Selection

Any time you click on the selection button of a table row on the rendered screen, a round trip is initiated. This round trip will cause the lead selection of the corresponding context node to which this table is bound to be adjusted.

In the figure below, the node collection contains two elements, and the user has clicked on the second table row. This corresponds to element 2 of node FLIGHTS, and the lead selection of this node is altered to reflect the user’s selection.

There will be as many rows in the table as there are elements in the node collection.

You can also define a Web Dynpro action and associate it with the table’s onLeadSelect event. Then, not only will the node collection’s lead selection element be adjusted when the user clicks on a table, but the hook method wddobeforeaction( ) and the corresponding action handler method of the view containing the Table View UI element will be processed.
Multiple Selection of Rows from a Table
Before a user is permitted to select multiple rows from a table (by holding down SHIFT or CTRL, and clicking on the required rows), the context node to which the Table UI element is bound must have a selection cardinality of either 0..n or 1..n. The default selection cardinality setting for any context node is 0..1, meaning that zero or one element may be selected at any one time. Each node provides the method get_selected_elements(), which returns all selected elements in an internal table (type WDR_CONTEXT_ELEMENT_SET).

Note: A table line in a view that is selected with the lead selection will be only part of the result of the method get_selected_elements() if the INCLUDING_LEAD_SELECTION parameter of the method is set to true.
The topics discussed in the two sections above are only true for the default value of the `Table` property `selectionMode` (`auto`). However, this property can have six values, influencing the possibility to mark rows and set the lead selection as follows:

**The Property `selectionMode`**

<table>
<thead>
<tr>
<th>value</th>
<th>minimum of selected rows</th>
<th>maximum of selected rows</th>
<th>selecting table row triggers round trip</th>
<th>selecting table row sets lead selection</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>auto</code></td>
<td>selection cardinality</td>
<td>selection cardinality</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><code>single</code></td>
<td>selection cardinality</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td><code>multi</code></td>
<td>selection cardinality</td>
<td>selection cardinality</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Test Page for UI Elements
In each system based on SAP NetWeaver 7.0, the Web Dynpro application `WDR_TEST_UI_ELEMENTS` is delivered. This can be used to check each UI element's functionality.

Figure 56: Test Page for UI Elements
Exercise 4: User Interface: Using Layout Managers and displaying MIME Repository Objects

Exercise Objectives
After completing this exercise, you will be able to:

• Arrange UI elements
• Display graphic files stored in the MIME Repository

Business Example
You want to develop a Web Dynpro application having an image displayed on the first view. In addition, the UI elements should be arranged in a predefined way.

Template: VCT_COND_S
Solution: VCT_UI_S1

Task 1:
Copy your solution from the previous exercise or the template component.
1. Copy your solution from the previous exercise (ZVCT_COND_##) or the Web Dynpro component VCT_COND_S to the new component, ZVCT_UI1_##.

Task 2:
Add an IMAGE UI element to the layout of view INPUT_VIEW. The IMAGE UI element should display the file reisen_1.jpg located in the MIME Repository folder SAP BC WebDynpro SAP PUBLIC VCT.

1. In the layout of view INPUT_VIEW create a UI element of type IMAGE (suggested name: IMG_PLANE) and maintain the source of the IMAGE element. The source should be file reisen_1.jpg located in the MIME Repository folder SAP BC Web Dynpro SAP PUBLIC VCT.

Continued on next page
Task 3:
Arrange the UI elements of view `INPUT_VIEW`. The group containing the input fields and the image should be arranged in one row. Both element together should occupy the complete page width. Group and Image should occupy the same height. The labels, input fields and the button in the group should be located in the left upper corner of the group UI element (not stretched across the group).

1. Assign the layout manager `Matrix Layout` to the `ROOTUIELEMENT-CONTAINER`.

2. Change the properties of the GROUP UI element and of the IMAGE element in a way, that they are arranged in one row.

3. Change the properties of the UI elements in a way, that GROUP and IMAGE occupy 100% of the page width. The width of the image should not differ from its original size.

4. Change the properties of the UI elements in a way, that GROUP and IMAGE occupy the same height. The height of the image should not differ from its original size.

5. Change the properties of the GROUP UI element in a way that the form elements are not stretched across the group (neither vertically nor horizontally).

Task 4:
Arrange the UI elements of view `OUTPUT_VIEW`. The group containing the form fields should occupy 100% of the page width.

1. Arrange the UI elements of the view using the `Matrix Layout` manager.
2. Change the properties of the UI elements in a way, that the GROUP occupies 100% of the page width.

Task 5:
Activate and test your Web Dynpro component.
1. Activate your component. Create a Web Dynpro application having the same name as your component. Start the Web Dynpro application.
Solution 4: User Interface: Using Layout Managers and displaying MIME Repository Objects

Task 1:
Copy your solution from the previous exercise or the template component.

1. Copy your solution from the previous exercise (ZVCT_COND ##) or the Web Dynpro component VCT_COND_S to the new component, ZVCT_UI1_##.
   a) One possible solution: In the object navigator choose Other object and select he Web Objects tab page.
   b) Enter the name of the Web Dynpro component to copy and choose Copy.
   c) Enter a name for the new Web Dynpro component.

Task 2:
Add an IMAGE UI element to the layout of view INPUT_VIEW. The IMAGE UI element should display the file reisen_1.jpg located in the MIME Repository folder SAP BC WebDynpro SAP PUBLIC VCT.

1. In the layout of view INPUT_VIEW create a UI element of type IMAGE (suggested name: IMG_PLANE) and maintain the source of the IMAGE element. The source should be file reisen_1.jpg located in the MIME Repository folder SAP BC WebDynpro SAP PUBLIC VCT.
   a) From the context menu of the ROOTUIELEMENTCONTAINER UI element, choose Insert Element.
   b) Enter the name of IMAGE element and choose element type IMAGE.
   c) Open the MIME Repository (choose the MIME Repository button at the top of the navigation area of the ABAP Workbench).
   d) Open the specified folder. Drag reisen_1.jpg and drop it on the IMAGE UI element (in the UI element hierarchy).

Continued on next page
Task 3:

Arrange the UI elements of view \textit{INPUT\_VIEW}. The group containing the input fields and the image should be arranged in one row. Both element together should occupy the complete page width. Group and Image should occupy the same height. The labels, input fields and the button in the group should be located in the left upper corner of the group UI element (not stretched across the group).

1. Assign the layout manager \textit{Matrix Layout} to the ROOTUIELEMENT- CONTAINER. 
   a) Set the property \textit{Layout} to \textit{MatrixLayout}.

2. Change the properties of the GROUP UI element and of the IMAGE element in a way, that they are arranged in one row. 
   a) For the GROUP UI element, choose \textit{LayoutData} = \textit{MatrixHeadData}, for the IMAGE UI element choose \textit{LayoutData} = \textit{MatrixData}.
   b) For both UI elements set \textit{vAlign} = top.

3. Change the properties of the UI elements in a way, that GROUP and IMAGE occupy 100\% of the page width. The width of the image should not differ from its original size. 
   a) For the ROOTUIELEMENTCONTAINER choose \textit{width} = 100\%.
   b) Set the property \textit{width} related to the cell surrounding the IMAGE (\textit{LayoutData} (MatrixData)) to the image's original width.

\textbf{Hint:} To find out the image's dimensions, double click the image in the MIME Repository. This will display the image in the preview pane. Right mouse click on the image in the preview pane and choose \textit{Properties} from the context menu. A popup appears displaying the dimensions of the image.

   c) For the GROUP UI element, set the property \textit{width} to 100\%.

\textit{Continued on next page}
4. Change the properties of the UI elements in a way, that GROUP and IMAGE occupy the same height. The height of the image should not differ from its original size.
   a) Set the property `height` related to the cell surrounding the IMAGE (`LayoutData (MatrixData)`) to the image's original height.

**Hint:** To find out the image's dimensions, double click the image in the MIME Repository. This will display the image in the preview pane. Right mouse click on the image in the preview pane and choose `Properties` from the context menu. A popup appears displaying the dimensions of the image.

b) For the GROUP UI element, set the property `height` to `100%`.

c) For the `ROOTUIELEMENTCONTAINER`, switch of the vertical stretching (property `stretch Vertically`).

5. Change the properties of the GROUP UI element in a way that the form elements are not stretched across the group (neither vertically nor horizontally).
   a) Set the properties `stretch Vertically` and `stretch Horizontally` to `space`.

**Task 4:**
Arrange the UI elements of view `OUTPUT_VIEW`. The group containing the form fields should occupy 100% of the page width.

1. Arrange the UI elements of the view using the `Matrix Layout` manager.
   a. For the `ROOTUIELEMENTCONTAINER`, set the property `Layout` to `MatrixLayout`.
   b. For the GROUP UI element, choose `LayoutData = MatrixHeadData`.

2. Change the properties of the UI elements in a way, that the GROUP occupies 100% of the page width.
   a) For the `ROOTUIELEMENTCONTAINER` choose `width = 100%`.
   b) For the GROUP UI element, set the property `width` (GROUP property and LayoutData property) to `100%`.

**Task 5:**
Activate and test your Web Dynpro component.

1. Activate your component. Create a Web Dynpro application having the same name as your component. Start the Web Dynpro application.
   a) Perform this step as in previous exercises.
Exercise 5: User Interface: Displaying Tables

Exercise Objectives
After completing this exercise, you will be able to:
• Create a service call to call a function module and bind the parameters to the controller context
• Create a UI element of type TABLE to display mass data

Business Example
Template: n/a
Solution: VCT_UI_S2

Task 1:
Create a Web Dynpro component with two views (input view and display view) embedded into the main window.

1. Create Web Dynpro component ZVCT_UI2 with two views (suggested names: INPUT_VIEW and DISPLAY_VIEW) and embed the views into the main window.

Task 2:
Create a service call of function module BAPI_FLIGHT_GETLIST in the component controller. This will create a method encapsulating the BAPI call. In addition, the BAPI's interface parameters will be bound to the component controller context.

1. Create a service call in the existing component controller for the function module BAPI_FLIGHT_GETLIST. In the Adapt Context dialog, choose Object Type = Context for the parameters DESTINATION_FROM, DESTINATION_TO, and FLIGHT_LIST. When asked for the name of the service method, accept the default name (EXECUTE_BAPI_FLIGHT_GETLIST).

Continued on next page
2. What entities of the component have been created?

3. What are the cardinalities of nodes DESTINATION_FROM and FLIGHT_LIST? Why are they different?

Task 3:

On the input view layout, create a group with input fields for departure city and destination city. In order to bind the input fields to view context attributes, copy the relevant nodes of the component context to the context of the input view and define the context mapping.

1. Copy the nodes DESTINATION_FROM and DESTINATION_TO from the component context to the context of the input view and define the mapping between the context nodes of the different controllers.
2. On the view layout, use the Web Dynpro Code Wizard to create a form providing an input field corresponding to attribute CITY of context node DESTINATION_FROM. The wizard will also create a group UI element embedding the form fields. Adjust the text to be displayed by UI element CITY_LABEL.
3. Repeat the last step for the attribute CITY of context node DESTINATION_TO. This will also create a second group. Adjust the text to be displayed by UI element CITY_LABEL_1. Move all labels and input fields in one of the groups and delete the second group.
4. Adjust the properties of the UI elements to make sure that the input fields (with their labels) are displayed one below the other.

Continued on next page
Task 4:
Copy the relevant nodes of the component context to the context of the display view and map the context nodes of the different controllers. Use the Web Dynpro Code Wizard to create a table displaying the flight info on the display view.
   1. Copy node FLIGHT_LIST from the component context to the context of the display view and map the two nodes.
   2. Use the Web Dynpro Code Wizard to create a table display with binding to the context node FLIGHT_LIST.

Task 5:
Provide a button on each view and implement navigation between the two views.
   1. Create an inbound plug and an outbound plug for each of the two views.
   2. Create navigation links between the plugs of the views (both directions).
   3. Provide a button on each of the views. Create actions related to these buttons and make sure the respective outbound plug is fired when the user presses the button.

Task 6:
Use the WebDynpro Code Wizard to implement a call of the service method EXECUTE_BAPI_FLIGHT_GETLIST in the event handler for the inbound plug of the display view.
   1. On the display view, implement the method to handle the inbound plug (method with name HANDLE... ). Start the Web Dynpro Code Wizard to implement a call of method EXECUTE_BAPI_FLIGHT_GETLIST of the component controller (method Call In Used Controller).
   2. Activate your component. Create a Web Dynpro application and test.
Solution 5: User Interface: Displaying Tables

Task 1:
Create a Web Dynpro component with two views (input view and display view) embedded into the main window.
1. Create Web Dynpro component ZVCT_UI2_## with two views (suggested names: INPUT_VIEW and DISPLAY_VIEW) and embed the views into the main window.
   a) Perform this step as in previous exercises.

Task 2:
Create a service call of function module BAPI_FLIGHT_GETLIST in the component controller. This will create a method encapsulating the BAPI call. In addition, the BAPI's interface parameters will be bound to the component controller context.
1. Create a service call in the existing component controller for the function module BAPI_FLIGHT_GETLIST.
   In the Adapt Context dialog, choose Object Type = Context for the parameters DESTINATION_FROM, DESTINATION_TO, and FLIGHT_LIST.
   When asked for the name of the service method, accept the default name (EXECUTE_BAPI_FLIGHT_GETLIST).
   a) Open the context menu for the Web Dynpro component in the navigation area of the ABAP Workbench (left-hand side) and choose Create Service Call.
   b) Choose Continue to confirm the first window of the wizard.
   c) Select Use Existing Controller, enter the component controller of your component, and choose Continue.
   d) Select Function Module and choose Continue.
   e) On the next screen, enter the name of the function module in the Function Module field and choose Continue.
   f) Check the parameters to be stored in the controller's context and choose Continue.
   g) Enter the name of the service method and a description and choose Continue.

Continued on next page
2. What entities of the component have been created?

**Answer:**

- Context node BAPI_FLIGHT_GETLIST with several sub-nodes for the selected IMPORTING and TABLES parameters of the function module
- Method EXECUTE_BAPI_FLIGHT_GETLIST in the component controller, which encapsulates the call of the function module
- Attributes for the function module parameters AIRLINE and MAX_ROWS.

3. What are the cardinalities of nodes DESTINATION_FROM and FLIGHT_LIST? Why are they different?

**Answer:**

- DESTINATION_FROM: 1...1
- FLIGHTLIST: 0...n

The parameter DESTINATION_FROM is a structure. The parameter FLIGHT_LIST is an internal table.

**Task 3:**

On the input view layout, create a group with input fields for departure city and destination city. In order to bind the input fields to view context attributes, copy the relevant nodes of the component context to the context of the input view and define the context mapping.

1. Copy the nodes DESTINATION_FROM and DESTINATION_TO from the component context to the context of the input view and define the mapping between the context nodes of the different controllers.

   a) Perform this step as in the previous exercises.

2. On the view layout, use the Web Dynpro Code Wizard to create a form providing an input field corresponding to attribute CITY of context node DESTINATION_FROM. The wizard will also create a group UI element embedding the form fields. Adjust the text to be displayed by UI element CITY_LABEL.

   a) Perform this step as in previous exercises.

3. Repeat the last step for the attribute CITY of context node DESTINATION_TO. This will also create a second group. Adjust the text to be displayed by UI element CITY_LABEL_1.

*Continued on next page*
Move all labels and input fields in one of the groups and delete the second group.
   a) Create the group as before. Move all elements from this group to the first
group by dragging and dropping them in the UI element hierarchy.
4. Adjust the properties of the UI elements to make sure that the input fields (with their
labels) are displayed one below the other.
   a) Set the Layout property of ROOTUIELEMENTCONTAINER to MatrixLayout
   b) Set the Layoutdata property of UI Element group to MatrixHeadData.
   c) Set the Layoutdata property of CITY_LABEL to MatrixHeadData.
   d) Set the Layoutdata property of CITY_LABEL_1 to MatrixHeadData.

Task 4:
Copy the relevant nodes of the component context to the context of the display
view and map the context nodes of the different controllers. Use the Web Dynpro
Code Wizard to create a table displaying the flight info on the display view.
   1. Copy node FLIGHT_LIST from the component context to the context of the
display view and map the two nodes.
      a) Perform this step as in the previous exercises.
   2. Use the Web Dynpro Code Wizard to create a table display with binding to the context
node FLIGHT_LIST.
      a) Place the cursor on ROOTUIELEMENTCONTAINER, press the Web
Dynpro Code Wizard button, and choose Standard Table.
      b) Choose Context and double-click context node FLIGHT_LIST.
      c) Accept the default (standard cell editor TextView and binding for all fields) and
choose.

Task 5:
Provide a button on each view and implement navigation between the two views.

1. Create an inbound plug and an outbound plug for each of the two views.
   a) Perform this step as in previous exercises.
2. Create navigation links between the plugs of the views (both directions).
   a) Perform this step as in previous exercises.

Continued on next page
3. Provide a button on each of the views. Create actions related to these buttons and make sure the respective outbound plug is fired when the user presses the button.
   a) Perform this step as in previous exercises.

**Task 6:**

Use the WebDynpro Code Wizard to implement a call of the service method EXECUTE_BAPI_FLIGHT_GETLIST in the event handler for the inbound plug of the display view.

1. On the display view, implement the method to handle the inbound plug (method with name HANDLE...). Start the Web Dynpro Code Wizard to implement a call of method EXECUTE_BAPI_FLIGHT_GETLIST of the component controller (method Call In Used Controller).
   a) Open the method in the editor. Choose *Web Dynpro Code Wizard* and select *Method Call In Used Controller*.
   b) Select the component controller and the method to be called from the input help and choose .

2. Activate your component. Create a Web Dynpro application and test.
   a) Perform this step as in previous exercises.
Unit 5

Controller and Context Programming

Controller Methods
Each Web Dynpro controller is a separate ABAP class. The definition of these classes is generated automatically when a new component or controller is declared. The source code that implements these controllers is generated automatically.

Each controller provides a number of predefined methods, which are called from the Web Dynpro runtime in a predefined order. These methods are called hook methods. At the time of controller creation, these methods are empty and can hold any coding the developer wishes to place within them. In addition to these hook methods, the developer can define additional methods: ordinary methods, event handler methods, or supply functions.

Figure 57: Controller Methods

From all controller methods, hook methods, and user-defined methods, the controller class is generated every time the controller is activated. The ABAP Workbench allows you to enter source code inside the controller methods only the generated controller class is not accessible.
Hook Methods

There are two hook methods that exist for all controller types:

- **wddoinit()**:  
  This is the first method processed in the controller's lifetime. It is only called once in the controller's life cycle. All your initialization code should go here, since this method is called immediately after the controller has been instantiated.

- **wddoexit()**:  
  This is the last method, which is processed at the end of a controller's life cycle. All your cleanup code should go here.

Figure 58: Standard Hook Methods for All Controllers

Depending on the controller type, additional hook methods are available. For the component controller, there are three additional standard methods. Two are of importance:

- **wddobeforenavigation()**:  
  Whenever a client event is raised, the corresponding action method in the view controller is processed. The method **wddobeforenavigation()** is called after the action method has been processed and just before the Web Dynpro framework processes the events in the navigation queue.

- **wddopostprocessing()**:  
  In complex Web Dynpro applications, it is possible that the data from multiple components must be validated before the next step in the business process can be taken. This method has been implemented so that cross-component validation can take place. It is the last controller method that is processed before the UI is sent to the client.
For view controllers, there are two additional hook methods:

- **wddobeforeaction( )**
  After the Web Dynpro application has been started and the user has raised a client event, the first methods to be processed in the Web Dynpro application are the wddobeforeaction( ) methods of all view controllers of the previous rendered view assembly. This happens even before the action handler method related to the client event is processed.

- **wddomodifyview( )**
  The only method that allows to access the UI element hierarchy is the view's wddomodifyview( ) method. The interface parameter VIEW is a reference to the UI element hierarchy; the FIRST_TIME parameter can be used to find out if this method has been previously processed in the view controller's life cycle. This method can be used to manipulate the UI element hierarchy dynamically.
**Additional Controller Methods**

For all controllers, you can create additional methods by declaring the method name and its parameters in the *Methods* tab of the controller editor window. To define an ordinary method, choose *Method Type = Method*. If you choose *Method Type = Event Handler*, an event handler method is created. It can be registered statically (in the *Event* column) to any event fired in a controller defined as a *Used Controller* on the *Properties* tab. Finally, *Method Type = Supply Function* can be set to define methods that can be bound to context nodes (*Supply Function* property). These methods are automatically called from the Web Dynpro framework if the node is accessed and it is marked as non-valid.

After having double-clicked on the method name, the signature of the method and the source code of the method can be defined.

All methods are public and can be used by any other controller of the same component. As a prerequisite, the controller that will call the method defines a usage relation to the controller that owns the methods. The ABAP source code to call a method in the used controller can be generated using the Web Dynpro Code Wizard.

If the *Interface* flag is set for a user-defined method, this method will appear in the component interface, so the method is also visible to other components.

![Controller Diagram](image)

*Figure 61: Additional Controller Methods*
Controller Attributes
Each controller is a separate ABAP program, having not only predefined and user-defined methods, but also attributes, which are at least visible for each method of the controller.

Standard Attributes
After having defined a controller, at least two attributes are predefined. These attributes have a visibility, which is restricted to the controller methods. The standard attributes are:

- **WD_THIS**
  WD_THIS is a self reference to the controller interface. This attribute must be distinguished from the standard ABAP self reference ME, which should not be used in the source code of any controller. WD_THIS is a reference to the current controller's interface, IF_<controller name>, and represents all the functionality implemented in the generated class. This also gives you access to standard Web Dynpro functionality, such as validation.

- **WD_CONTEXT**
  WD_CONTEXT is the reference to the controller's context root node, and thus to the entire context. Any access to the controller's context starts with this reference.

Figure 62: Standard Controller Attributes WD_CONTEXT and WD_THIS
If the component controller is declared as a used controller on the Properties tab of any other controller, an additional attribute is automatically created for the controller that declared the usage:

- **WD_COMP_CONTROLLER**

  *WD_COMP_CONTROLLER* is the reference to the component controller. Using this reference, all methods and all public attributes of the component controller can be accessed (wd_comp_controller-><meth>, where <meth> is a placeholder for the methods name).

![Figure 63: Standard Controller Attribute WD_COMP_CONTROLLER](image)

**Figure 63: Standard Controller Attribute WD_COMP_CONTROLLER**

For all other controllers, even if declared as used controllers, such a reference is not available. However this does not mean, that user-defined methods and public attributes are not available, but that the reference must be evaluated first. To have the reference to the <ctrl> controller defined as a used controller, the following statement must be used:

```
DATA: lo_ctrl TYPE REF TO ig_<ctrl> .
lo_ctrl = wd_this->get_<ctrl>_ctr( ).
```

**User-Defined Attributes**

On the Attributes tab, additional attributes can be defined for the related controller. If the Public flag is set, these attributes are also visible for other controllers of the same Web Dynpro component. Attributes can not be exposed to the components interface.

In order to access public controller attributes in one of the controller's methods, the self-reference variable *WD_THIS* has to be used. Accessing public attributes defined in other controllers of the same component is implemented similarly to accessing methods defined in other controllers.
In the last section we saw that we can use controller attributes to provide data objects that are visible throughout the controller. However, it is not possible to bind UI element properties to these controller attributes. UI element properties can only be bound to variables defined in the controller context. This hierarchical data storage is also preferable if information has to be shared between controllers. Accessing the controller context at runtime requires knowledge of the appropriate Web Dynpro methods. This section deals with how to read, change, add, or delete information stored in the controller context.

Accessing a Context Node

Accessing a context element or a context attribute requires you to first have a reference to the related context node.
The following information is of importance:

- For each controller (\(<ctrl>\), an interface is generated having the name \(\text{IF}_{<ctrl>}\).
- For each node \(<node>\) of a controller context, a constant \(\text{WDCTX}_{<node>}\) is generated in this interface; it has the name of the node (in uppercase letters) as its value. This constant can be used to access the context node.

The context root node can be accessed by the standard attribute \(\text{WD\_CONTEXT}\). Child nodes of the context root node can be identified using the \(\text{get\_child\_node}(\) method. This method returns a reference to the node instance of type \(\text{IF\_WD\_CONTEXT\_NODE}\). The \(\text{get\_child\_node}(\) method requires the name of the node and, optionally, the index of the element in the parent node to which the desired node instance belongs. In this case, the parent node is the context root, which only ever has one element; therefore, the index parameter is 1 (default = lead selection of parent element).

**Accessing a Node Element**

After having accessed a context node, the reference to the element at lead selection of this node can be obtained by calling the method \(\text{get\_element}(\) method. This method returns a reference to the element instance of type \(\text{IF\_WD\_CONTEXT\_ELEMENT}\).
Figure 66: Accessing the Node Element at Lead Selection

The element with index $n$ can be accessed using the method `get_element(index=n)`.
The number of elements in a collection can be obtained from the method `get_element_count()`.

**Summary**

<table>
<thead>
<tr>
<th>Action</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference to node <code>&lt;node&gt;</code> of a controller <code>&lt;ctrl&gt;</code></td>
<td><code>lo_nd_&lt;node&gt; = wd_context-&gt;get_child_node( name = wd_this-&gt;wdctx_&lt;node&gt; ).</code></td>
</tr>
<tr>
<td>Reference to element at lead selection</td>
<td><code>lo_el_&lt;node&gt; = lo_nd_&lt;node-&gt;get_element( ).</code></td>
</tr>
<tr>
<td>Reference to element with index $n$</td>
<td><code>lo_el_&lt;node&gt; = lo_nd_&lt;node-&gt;get_element( index = n ).</code></td>
</tr>
<tr>
<td>Get number of elements in collection</td>
<td><code>n = lo_nd_&lt;node-&gt;get_element_count( ).</code></td>
</tr>
</tbody>
</table>
Reading and Changing Attribute Values

The Web Dynpro framework offers a variety of methods to access the attributes of one node element or to access the attributes of all elements of a context node.

Accessing Attributes of a Single Node Element

Once you have obtained the reference to a node element, there are two ways to obtain the attribute values of this element:

1. Any attribute of a node element can be accessed using the method `get_attribute()` . Here, the name of the attribute must be exported and the attribute value is returned in an import parameter.
2. Statically defined attributes can be obtained by calling the method `get_static_attributes()` . Here, a structure is returned in an import parameter. The target structure can be different from the node structure. This is possible because structure elements are copied individually to the target structure.

```plaintext
DATA lo_nd_flights TYPE REF TO if_wd_context_node.
DATA lo_el_flights TYPE REF TO if_wd_context_element.
DATA lv_connid TYPE wd_this->element_flights->connid.

lo_nd_flights = wd_context->get_child_node( name = wd_this->wdctx_flights ) .
lo_el_flights = lo_nd_flights->get_element( ) .

* get single attribute
lo_el_flights->get_attribute( 
EXPORTING
   name = 'CONNID'
IMPORTING
   value = lv_connid ) .
```

Figure 67: Accessing a Single Attribute of a Node Element
The following information is of importance in this context:

- For each node `<node>` of a controller context, a structure type `element_<node>` is implicitly generated in the interface `IF_<ctrl>`. The structure fields correspond to the attributes a node element consists of. This constant can be used to type a variable, which is filled by the methods listed above.

- In addition, for each node `<node>` of a controller context, a standard table type `elements_<node>` is implicitly generated in the interface `IF_<ctrl>`. The line type of this table is `element_<node>`. This constant can be used to type an internal table that can hold the attributes of multiple node elements.

### Accessing the Attributes of all Node Elements

```plaintext
DATA lo_md_flights TYPE REF TO if_wd_context_node.
DATA lo_el_flights TYPE REF TO if_wd_context_element.
DATA ls_flights TYPE wd_this->element_flights.

lo_md_flights = wd_context->get_child_node(name = wd_this->wdctx_flights).
lo_el_flights = lo_md_flights->get_element().

* get all statically declared attributes
lo_el_flights->get_static_attributes(
  IMPORTING
  static_attributes = ls_flights).
```

---

**Figure 68: Accessing All Statically Defined Attributes of a Node Element**

Accessing the Attributes of all Node Elements

```plaintext
DATA lo_md_flights TYPE REF TO if_wd_context_node.
DATA lt_flights TYPE wd_this->elements_flights.

lo_md_flights = wd_context->get_child_node(name = wd_this->wdctx_flights).

* get all statically declared attributes of all node elements
lo_md_flights->get_static_attributes_table(
  IMPORTING
  table = lt_flights).
```
Figure 69: Access to the Static Attributes of All Node Elements

With the method `get_static_attributes_table()` , the attributes of all elements can be retrieved as an internal table.

Changing Attribute Values of a given Node Element

After the reference to a certain node element has been determined, the attribute values cannot only be read, using the appropriate getter methods, but it is also possible to change existing attribute values by calling related setter methods. The method `set_attribute()` can be used to change the value of any attribute of the node element. Multiple attributes can be changed if they are statically defined using the method `set_static_attributes()` .

Figure 70: Changing a Single Attribute of a Node Element

Figure 71: Changing Multiple Attributes of a Node Element
Summary

<table>
<thead>
<tr>
<th>Action</th>
<th>Method</th>
</tr>
</thead>
</table>
| Read value of attribute \(<attr>\) | DATA:
    \[lv\_<attr>\ TYPE
    \quad \text{wd_this-}\Rightarrow\text{element}_<node>-<attr>\].

\[
\text{lo_el}_<node>->\text{get_attribute}(
    \text{EXPORTING}
    \quad \text{name} = '<attr>'
    \quad \text{IMPORTING}
    \quad \text{value} = \text{lv}_<attr> \).
\] |
| Read value of multiple static attributes | DATA:
    \[\text{ls}_<node> \ TYPE
    \quad \text{wd_this-}\Rightarrow\text{element}_<node>\].

\[
\text{lo_el}_<node>->\text{get_static_attributes}(
    \text{IMPORTING}
    \quad \text{static_attributes} = \text{ls}_<node> \).
\] |

<table>
<thead>
<tr>
<th>Action</th>
<th>Method</th>
</tr>
</thead>
</table>
| Read static attribute values for all node elements | DATA:
    \[\text{lt}_<node> \ TYPE \text{wd_this-}\Rightarrow\text{elements}_<node>\].

\[
\text{lo_nd}_<node>->\text{get_static_attributes_table}(
    \text{IMPORTING}
    \quad \text{table} = \text{lt}_<node> \).
\] |
| Change value \(lv\_<attr>\) of a single attribute \(<attr>\) | DATA:
    \[lv\_<attr> \ TYPE
    \quad \text{wd_this-}\Rightarrow\text{element}_<node>-<attr>\].

\[
\text{lv}_<attr> = \ldots
d
\]

\[
\text{lo_el}_<node>->\text{set_attribute}(
    \text{EXPORTING}
    \quad \text{name} = '<attr>'
    \quad \text{value} = \text{lv}_<attr> \).
\] |
Adding new Elements to a Context Node

Up to now we have discussed the reading and changing of data already stored in the controller context. Now we will learn how to add new elements to a context node. Adding a new element to a node is performed in two steps. The first step is to create an element that can be added to a certain context node later. After the attribute values have been defined, the new element can be added to the context node. This is like the process of adding a new line to an internal table. The first step is to define the cell values of a work area having the correct line type, and the second step is to insert the work area into the internal table.

Creating a New Node Element

In order to create an element that can be added to a certain context node, the reference to this node has to be determined first. This is done by using the `get_child_node()` method of the standard attribute `WD_CONTEXT`, pointing to the context root node.

Figure 72: Getting the Reference to a Context Node

Once you have obtained this reference, the method `create_element()` is used to create the new element. Initial values for the static attributes can be submitted.
using the `static_attribute_values` parameter or the attribute values can be defined using the setter methods `set_attribute()` or `set_static_attributes()`.

**Caution:** The new element is not yet part of the context node!

**Figure 73: Creating a New Node Element**

**Figure 74: Setting the Attribute Values of the New Element**
**Adding Elements to a Context Node**

Finally, an element, which is not yet part of the context node can be added to the node using the method `bind_element()` related to the node reference. This method has two import parameters:

- The element reference is submitted via the parameter `new_item`.
- The parameter `set_initial_elements` defines if the new element is just added to the element collection (value = `abap_false`) or if the new element replaces all existing elements in the collection (value = `abap_true`).

![Diagram: Binding an Element to a Context Node](image)

```abap
DATA lo_md_flights TYPE REF TO if_wd_context_node.
DATA lo_el_flights TYPE REF TO if_wd_context_element.

lo_md_flights = wd_context->get_child_node( name = wd_this->wctx_flights ).
lo_el_flights = lo_md_flights->create_element( ).
lo_el_flights->set_attribute( name = 'CARRID' value = '2H' ).
lo_el_flights->set_attribute( name = 'CONNID' value = '400' ).

* bind element FIRST_ELEMENT to node
lo_md_flights->bind_element( 
  new_item = lo_el_flights
  set_initial_elements = abap_false ).
```

**Figure 75: Binding an Element to a Context Node**

**Binding a Structure to a Context Node**

In ABAP programs, data sets are handled as structures. However, to be able to display the structure elements in the UI, the structure content must be copied to a context element. This means that a new element has to be defined, the attribute values have to be set, and this element has to be bound to the appropriate context node.

There is an easier way to copy the structure content as a new element to the context node. Instead of using the method `bind_element()` and submitting the element reference, the method `bind_structure()` can be used with parameter `new_item` to submit the structure. As for the `bind_element()` method, the existing collection can be extended or replaced (parameter `set_initial_elements`).
Multiple data sets having an identical structure are handled as internal tables in ABAP programs. However, to be able to display the data sets in the UI, the internal table content has to be copied to as many context elements as there are lines in the internal table.

The best way to do this is to use the method `bind_table()` . Here, the internal table is submitted via the parameter `new_items` . The existing collection can be extended or replaced (parameter `set_initial_elements` ).

---

**Figure 76: Binding a Structure to a Context Node**

**Figure 77: Binding an Internal Table to a Context Node**
Deleting Elements from a Context Node
To remove an element from a collection, the method `remove_element()` has to be called. The reference to the element must be submitted using the parameter `element`.

Summary

<table>
<thead>
<tr>
<th>Action</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a new element</td>
<td><code>lo_el_&lt;node&gt; = lo_nd_&lt;node&gt;-create_element( );</code></td>
</tr>
<tr>
<td>Add element to collection</td>
<td><code>lo_nd_&lt;node&gt;-bind_element( </code>new_item = lo_el_&lt;node&gt;`</td>
</tr>
<tr>
<td></td>
<td><code>set_initial_elements = abap_false );</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bind structure <code>ls_&lt;node&gt;</code> to collection</td>
<td>DATA:  <code>ls_&lt;node&gt; TYPE wd_this-&gt;element_&lt;node&gt;.</code></td>
</tr>
<tr>
<td></td>
<td><code>...</code></td>
</tr>
<tr>
<td></td>
<td><code>lo_nd_&lt;node&gt;-bind_structure( </code>new_item = ls_&lt;node&gt;`</td>
</tr>
<tr>
<td></td>
<td><code>set_initial_elements = abap_false );</code></td>
</tr>
<tr>
<td>Bind internal table <code>lt_&lt;node&gt;</code> to collection</td>
<td>DATA:  <code>lt_&lt;node&gt; TYPE wd_this-&gt;elements_&lt;node&gt;.</code></td>
</tr>
<tr>
<td></td>
<td><code>...</code></td>
</tr>
<tr>
<td></td>
<td><code>lo_nd_&lt;node&gt;-bind_table( </code>new_items = lt_&lt;node&gt;`</td>
</tr>
<tr>
<td></td>
<td><code>set_initial_elements = abap_false );</code></td>
</tr>
<tr>
<td>Remove element from collection</td>
<td><code>lo_nd_&lt;node&gt;-remove_element( </code>element = lo_el_&lt;node&gt; );`</td>
</tr>
</tbody>
</table>
Exercise 6: Accessing the Context at Runtime

Exercise Objectives
After completing this exercise, you will be able to:
• Set the value of context node attributes dynamically
• Set default values for input fields dynamically

Business Example
You want to develop a Web Dynpro application where input fields of the first view are filled with default values at run time.

Template: VCT_COND_S
Solution: VCT_CONR_S1 / VCT_CONR_S1_OPT

Task 1:
Copy your Web Dynpro component ZVCT_COND_## or the template VCT_COND_S to Web Dynpro component ZVCT_CONR1_##.
1. Copy the template.

Task 2:
Make AA the default value of the CARRID field before the input view is displayed.
1. Implement method WDDOINIT of the view controller. Use the Web Dynpro Code Wizard to read context node FLIGHTINFO.
2. Replace the last method call (get_static_attributes method) with a call of a context element method to set the value of attribute CARRID to AA.

Task 3: (Optional)
Set default values for all input fields before the view is displayed. Use a structure and a single method call instead of setting the default values one by one.
1. Turn the last method call into a comment. Replace it as follows: Fill structure stru_flightinfo with default values and call a method of the context element to set all attributes in one step.
Solution 6: Accessing the Context at Runtime

Task 1:
Copy your Web Dynpro component ZVCT_COND_## or the template VCT_COND_S to Web Dynpro component ZVCT_CONR1_##.

1. Copy the template.
   a) Perform this step as in previous exercises.

Task 2:
Make AA the default value of the CARRID field before the input view is displayed.
1. Implement method WDDOINIT of the view controller. Use the Web Dynpro Code Wizard to read context node FLIGHTINFO.
   a) Press the Web Dynpro Code Wizard button.
   b) Select Read Context Node/Attribute and use the input help to choose the context node.
   2. Replace the last method call (get_static_attributes method) with a call of a context element method to set the value of attribute CARRID to AA.
      a) See source code sample from the model solution under Result.

Task 3: (Optional)
Set default values for all input fields before the view is displayed. Use a structure and a single method call instead of setting the default values one by one.
1. Turn the last method call into a comment. Replace it as follows:
   Fill structure stru_flightinfo with default values and call a method of the context element to set all attributes in one step.
      a) See source code sample from the model solution.

Result

Model Solution: VCT_CONR_S1.

METHOD wddoinit.

DATA lo_nd_flightinfo TYPE REF TO if_wd_context_node.
DATA lo_el_flightinfo TYPE REF TO if_wd_context_element.

Continued on next page
* DATA ls_flightinfo TYPE wd_this->element_flightinfo.
* DATA lv_carrid LIKE ls_flightinfo-carrid.

* navigate from <CONTEXT> to <FLIGHTINFO> via lead selection
lo_nd_flightinfo = wd_context->get_child_node( name = wd_this->wdctx_flightinfo ).

* get element via lead selection
lo_el_flightinfo = lo_nd_flightinfo->get_element( ).

* set single attribute
lo_el_flightinfo->set_attribute(
    value = 'AA'
    name = 'CARRID' ).

ENDMETHOD.

Model Solution: VCT_CONR_S1_OPT

METHOD wddoinit .

DATA lo_nd_flightinfo TYPE REF TO if_wd_context_node.
DATA lo_el_flightinfo TYPE REF TO if_wd_context_element.
DATA ls_flightinfo TYPE wd_this->element_flightinfo.

* navigate from <CONTEXT> to <FLIGHTINFO> via lead selection
lo_nd_flightinfo = wd_context->get_child_node( name = wd_this->wdctx_flightinfo ).

* get element via lead selection
lo_el_flightinfo = lo_nd_flightinfo->get_element( ).

* set structure fields
ls_flightinfo-carrid = 'AA'.
ls_flightinfo-connid = '0017'.

* set all declared attributes
lo_el_flightinfo->set_static_attributes(
    EXPORTING
    static_attributes = ls_flightinfo ).

ENDMETHOD.
Exercise 7: Context at Runtime: Binding Internal Tables to Context Nodes

Exercise Objectives
After completing this exercise, you will be able to:
• Bind internal tables to context nodes at runtime

Business Example
You want to develop a Web Dynpro application where the user can enter selection criteria on a first view. After the user presses a button, data is selected and navigation to a second view is triggered, on which the selected data is displayed as a table.

Template: VCT_CONR_S1_OPT
Solution: VCT_CONR_S2

Task 1:
Copy your Web Dynpro component ZVCT_CONR1_## or the template VCT_CONR_S1_OPT to Web Dynpro component ZVCT_CONR2 ##.
In the component context, create a new context node to store data sets for flights read from database table SFLIGHT.
  1. Copy the template.
  2. In the component context, create a new context node (suggested name: FLIGHTTAB ) with reference to ABAP Dictionary structure SFLIGHT and cardinality 0...n. The node should contain the following attributes: CARRID, CONNID, FLDATE, PLANETYPE, SEATSMAX, and SEATSOCC.

Task 2:
Copy the new context node to the context of the view OUTPUT_VIEW and map the context nodes of the different controllers. Extend the layout of the view to display the data in a table.
  1. Copy component context node FLIGHTTAB to the context of the view OUTPUT_VIEW. Map the node of the view controller context to the node in the component controller context.
  2. Use the Web Dynpro Code Wizard to create a table display with binding to the context node FLIGHTTAB.

Continued on next page
Task 3:
Create a method in the component controller in which you select flights from database table SFLIGHT and store them in an internal table. Use the static method CL_VCT_FLIGHTMODEL=>READ_FLIGHTS() to collect the data. Store the result in context node FLIGHTTAB.

1. Create a new method in the component controller (suggested name: FLIGHTTAB_FILL).
2. Use the Web Dynpro Code Wizard to read context node FLIGHTINFO.
3. Create an internal table (suggested name: LT_FLIGHTTAB) of table type VCT_T_SFLIGHT. Use the static method CL_VCT_FLIGHTMODEL=>READ_FLIGHTS() to fill the internal table (export parameter e_flights). Use CARRID and CONNID of context node FLIGHTINFO to restrict the data selection.
4. Use the Web Dynpro Code Wizard to read context node FLIGHTTAB after having called the static method CL_VCT_FLIGHTMODEL=>READ_FLIGHTS() . Since you do not want to read data from the context but store data in the context, remove all method calls created by the wizard except for the first one (method GET_CHILD_NODE).
5. Call method BIND_TABLE for context node FLIGHTTAB to store the content of the internal table in the context node.

Task 4:
Make sure your new component controller method is executed after the navigation, immediately before the output view is displayed.

1. Edit method HANDLEFROM_INPUT_VIEW of the OUTPUT_VIEW view controller and use the Web Dynpro Code Wizard to implement a call of the component controller method.
**Solution 7: Context at Runtime: Binding Internal Tables to Context Nodes**

**Task 1:**
Copy your Web Dynpro component `ZVCT_CONR1_##` or the template `VCT_CONR_S1_OPT` to Web Dynpro component `ZVCT_CONR2_##`.

In the component context, create a new context node to store data sets for flights read from database table `SFLIGHT`.

1. Copy the template.
   a) Perform this step as in previous exercises.
2. In the component context, create a new context node (suggested name: `FLIGHTTAB`) with reference to ABAP Dictionary structure `SFLIGHT` and cardinality `0..n`.

The node should contain the following attributes: CARRID, CONNID, FLDATE, PLANETYPE, SEATSMAX, and SEATSOCC.

   a) Perform this step as in previous exercises.

**Task 2:**
Copy the new context node to the context of the view `OUTPUT_VIEW` and map the context nodes of the different controllers. Extend the layout of the view to display the data in a table.

1. Copy component context node `FLIGHTTAB` to the context of the view `OUTPUT_VIEW`. Map the node of the view controller context to the node in the component controller context.
   a) Perform this step as in previous exercises.
2. Use the Web Dynpro Code Wizard to create a table display with binding to the context node `FLIGHTTAB`.
   a) Perform this step as in previous exercises.

*Continued on next page*
Task 3:
Create a method in the component controller in which you select flights from database table SFLIGHT and store them in an internal table. Use the static method CL_VCT_FLIGHTMODEL=>READ_FLIGHTS() to collect the data. Store the result in context node FLIGHTTAB.

1. Create a new method in the component controller (suggested name: FLIGHTTAB_FILL).
   a) In the component controller, choose the Methods tab, enter the name of the method, and double-click on method's name.

2. Use the Web Dynpro Code Wizard to read context node FLIGHTINFO.
   a) Perform this step as in previous exercises.

3. Create an internal table (suggested name: LT_FLIGHTTAB) of table type VCT_T_SFLIGHT. Use the static method CL_VCT_FLIGHTMODEL=>READ_FLIGHTS() to fill the internal table (export parameter e_flights). Use CARRID and CONNID of context node FLIGHTINFO to restrict the data selection.
   a) See source code of model solution.

4. Use the Web Dynpro Code Wizard to read context node FLIGHTTAB after having called the static method CL_VCT_FLIGHTMODEL=>READ_FLIGHTS(). Since you do not want to read data from the context but store data in the context, remove all method calls created by the wizard except for the first one (method GET_CHILD_NODE).
   a) See source code of model solution.

5. Call method BIND_TABLE for context node FLIGHTTAB to store the content of the internal table in the context node.
   a) See source code of model solution.

Task 4:
Make sure your new component controller method is executed after the navigation, immediately before the output view is displayed.

1. Edit method HANDLEFROM_INPUT_VIEW of the OUTPUT VIEW view controller and use the Web Dynpro Code Wizard to implement a call of the component controller method.
   a) Open method HANDLEFROM_INPUT_VIEW.
   b) Choose Web Dynpro Code Wizard.
   c) Select Method Call in Used Controller.

Continued on next page
d) Select the component controller and enter the name of the method.

**Result**

Model Solution: VCT_CONR_S2, Method FLIGHTTAB_FILL

METHOD flighttab_fill.

DATA lo_nd_flightinfo TYPE REF TO if_wd_context_node.
DATA lo_el_flightinfo TYPE REF TO if_wd_context_element.
DATA ls_flightinfo TYPE wd_this->element_flightinfo.
DATA lo_nd_flighttab TYPE REF TO if_wd_context_node.
DATA lt_flighttab TYPE VCT_t_sflight.

* navigate from <CONTEXT> to <FLIGHTINFO> via lead selection
  lo_nd_flightinfo = wd_context->get_child_node(name = wd_this->wdctx_flightinfo).

  * get element via lead selection
   lo_el_flightinfo = lo_nd_flightinfo->get_element() .

  * get all declared attributes
    lo_el_flightinfo->get_static_attributes(IMPORTING
      static_attributes = ls_flightinfo).

  * read all flights realed to CARRID and CONNID entered by user
    cl_VCT_flightmodel=>read_flights(EXPORTING
      i_carrid = ls_flightinfo-carrid
      i_connid = ls_flightinfo-connid
    IMPORTING
    e_flights = lt_flighttab).

  * navigate from <CONTEXT> to <FLIGHTTAB> via lead selection
    lo_nd_flighttab = wd_context->get_child_node(name = wd_this->wdctx_flighttab).

  * bind table to context node <FLIGHTTAB>
    lo_nd_flighttab->bind_table(new_items = lt_flighttab).

ENDMETHOD.
Exercise 8: Context at Runtime: Lead Selection and Supply Functions

Exercise Objectives
After completing this exercise, you will be able to:
• Use supply functions to populate context nodes

Business Example
You want to develop a Web Dynpro application with a view that displays two tables: a list of flights and a list of bookings. The data displayed in the second table should depend on the selected row in the first table. You want to use a supply function to make sure that the data in the second table is changed whenever the user selects a different row in the first table.

Template: VCT_CONR_S2
Solution: VCT_CONR_S3

Task 1:
Copy your Web Dynpro component ZVCT_CONR2_# or the template VCT_CONR_S2 to Web Dynpro component ZVCT_CONR3_##.
In the component context, create a new context node as sub-node of node FLIGHTTAB to store data for a set of bookings read from database table SBOOK.
1. Copy the template.
2. In the component context, create a sub-node of context node FLIGHTTAB(suggested name: BOOKINGTAB) with reference to the ABAP Dictionary structure SBOOK and cardinality 0...n. The node should contain the following attributes: BOOKID, CUSTOMID, CUSTTYPE, LUGGWEIGHT, WUNIT, CLASS, and PASSNAME.

Continued on next page
Task 2:
Make sure the context node BOOKINGTAB also exists in the view OUTPUT_VIEW and that this node is mapped to the BOOKINGTAB node of the component controller context. Extend the layout of the view to display the bookings in a second table below the flights table.

1. Update the mapping for context node FLIGHTTAB in the OUTPUT_VIEW view context.

Hint: You have to use the update functionality here. It is not possible to map sub-nodes directly by dragging and dropping them.

Use the Web Dynpro Code Wizard to create a table display with binding to the context node BOOKINGTAB.

Task 3:
Create and implement a supply function to fill sub-node BOOKINGTAB according to the lead selection of node FLIGHTTAB.

1. In the component context, assign a supply function to sub-node BOOKINGTAB (suggested name: BOOKINGS_READ) and create it using forward navigation.

2. The supply method contains commented source code. Remove the comments or the following statements:
   - The declaration of the structure LS_PARENT_ATTRIBUTES.
   - The call of method GET_STATIC_ATTRIBUTES() for the parent element.
   - The declaration of the internal table LT_BOOKINGTAB.
   - The call of method BIND_TABLE() used to bind the table LT_BOOKINGTAB to the context node. The rest of the commented lines can be deleted.

3. Call the static method CL_VCT_FLIGHTMODEL=>READ_BOOKINGS() read all bookings for the selected flight. Use the attributes CARRID, CONNID, and FLDATE of the parent element to restrict the data selection. Change the type of the internal table LT_BOOKINGTAB to the type of the related parameter E_BOOKINGS of method READ_BOOKINGS().
Solution 8: Context at Runtime: Lead Selection and Supply Functions

Task 1:
Copy your Web Dynpro component ZVCT_CONR2_# or the template VCT_CONR_S2 to Web Dynpro component ZVCT_CONR3_##.

In the component context, create a new context node as sub-node of node FLIGHTTAB to store data for a set of bookings read from database table SBOOK.

1. Copy the template.
   a) Perform this step as in previous exercises.

2. In the component context, create a sub-node of context node FLIGHTTAB (suggested name: BOOKINGTAB) with reference to the ABAP Dictionary structure SBOOK and cardinality 0...n. The node should contain the following attributes: BOOKID, CUSTOMID, CUSTTYPE, LUGGWEIGHT, WUNIT, CLASS, and PASSNAME.
   a) Perform this step as in previous exercises.

Task 2:
Make sure the context node BOOKINGTAB also exists in the view OUTPUT_VIEW and that this node is mapped to the BOOKINGTAB node of the component controller context. Extend the layout of the view to display the bookings in a second table below the flights table.

1. Update the mapping for context node FLIGHTTAB in the OUTPUT_VIEW view context.

   Hint: You have to use the update functionality here. It is not possible to map sub-nodes directly by dragging and dropping them.
   a) Open the context menu for context node FLIGHTTAB and choose Update Mapping.
   b) Confirm the following two dialog boxes by choosing Yes.

2. Use the Web Dynpro Code Wizard to create a table display with binding to the context node BOOKINGTAB.
   a) Perform this step as in previous exercises.

Continued on next page
Task 3:
Create and implement a supply function to fill sub-node BOOKINGTAB according to the lead selection of node FLIGHTTAB.

1. In the component context, assign a supply function to sub-node BOOKINGTAB (suggested name: BOOKINGS_READ) and create it using forward navigation.
   a) In the component context, double click the sub-node.
   b) Enter the name for the supply function as attribute of property Supply Function and double-click on the name.

2. The supply method contains commented source code. Remove the comments for the following statements:
   - The declaration of the structure LS_PARENT_ATTRIBUTES.
   - The call of method GET_STATIC_ATTRIBUTES() for the parent element.
   - The declaration of the internal table LT_BOOKINGTAB.
   - The call of method BIND_TABLE() used to bind the table LT_BOOKINGTAB to the context node. The rest of the commented lines can be deleted.
     a) See source code of model solution.

3. Call the static method CL_VCT_FLIGHTMODEL=>READ_BOOKINGS() to read all bookings for the selected flight. Use the attributes CARRID, CONNID, and FLDAT of the parent element to restrict the data selection. Change the type of the internal table LT_BOOKINGTAB to the type of the related parameter E_BOOKINGS of method READ_BOOKINGS( ).
   a) See source code of model solution.

Result

Model Solution: VCT_CONR_S3, Method BOOKINGS_READ

METHOD bookings_read .*

* General Notes
* ================
* A common scenario for a supply method is to acquire key informations from the parameter <parent_element> and then

Continued on next page
* to invoke a data provider.
* A free navigation thru the context, especially to nodes on
* the same or deeper hierachical level is strongly discouraged,
* because such a strategy may easily lead to unresolvable
* situations!!

* data declarations
DATA ls_parent_attributes TYPE wd_this->element_flighttab.
DATA lt_bookingtab TYPE VCT_t_sbook.

* get all static attributes of parent element
parent_element->get_static_attributes(
  IMPORTING
  static_attributes = ls_parent_attributes ).

* read related bookings
cl_VCT_flightmodel=>read_bookings(
  EXPORTING
  i_carrid = ls_parent_attributes-carrid
  i_connid = ls_parent_attributes-connid
  i_fldate = ls_parent_attributes-fldate
  IMPORTING
  e_bookings = lt_bookingtab ).

* bind all the elements
node->bind_table(
  new_items = lt_bookingtab
  set_initial_elements = abap_true ).

ENDMETHOD.
Unit 6

Internationalization and Messages

Internationalization
Internationalization means that all language-dependent parts of applications have to be defined in a way that a language-specific version of the entity can be defined and that the correct version is used at runtime according to the language the user used to log on.

Figure 78: Internationalization
Note: Due to the fact that internationalization is such a long word, it is often abbreviated to I18N. That is, the first letter I, the last letter N, and don’t bother about the 18 other letters in between!

Language-dependent entities are label texts, tool tips, button texts, and message texts, but also images and so on.

Using Texts Defined in the ABAP Dictionairy

Figure 79: Referring to ABAP Dictionary Texts

Texts defined in the ABAP Dictionary can be used in the following ways:

- The UI element Label is related to a UI element that allows user input (for example InputField). This related UI element, in turn, must be bound to a context attribute in respect to the property holding the field value. If the context attribute is typed with an ABAP Dictionary type, the corresponding data element text (middle text) will automatically be used as the label text. The same mechanism applies to Captions of TableColumns.

- Each data element text can also be referred directly. This is done by pressing the button right of a properties value field. On the pop up that appears, press the button DDIC binding on/off to choose a DDIC type and the kind of text (short, middle, long...).

- From the source code, the method get_available_texts() of the class CL_TEXT_IDENTIFIER can be used to read texts from the ABAP Dictionary.
**Defining and using Online Text Repository (OTR) Texts**

The Online Text Repository (OTR) is a central storage area for texts that can be used not only in a Web Dynpro context but also in BSPs, in classes, and in normal ABAP programs (type 1 / type M). Different kinds of texts can be defined in the OTR: OTR long texts, OTR short texts without alias, or OTR short texts having an alias.

OTR long texts do not have a limit for their length. However, the disadvantage is that they can only be used once. If you want to use the same long text a second time, it has to be rewritten in the original language again and (even worse) it has to be translated again. In the Web Dynpro context, OTR long texts are automatically created when a text is entered in a value field of a UI element's property.

OTR short texts are limited to 255 characters and an alias can be assigned to these texts. OTR short text having an alias (OTR alias texts) can be reused, and they have to be translated only once. In the Web Dynpro context, only OTR alias texts should be used.

The OTR provides services for accessing these texts at runtime, and it supports the entry and translation of texts. To create a new OTR alias text, the Online Text Repository browser can be used. This tool can be found in the *Goto* menu when editing a view. The alias of an OTR short text consists of the package name followed by a slash and an arbitrary identifier (\texttt{<package>/<alias>}). OTR texts can also be created using transaction SOTR>Edit. This transaction also allows you to search for existing texts, translate texts, and find where OTR texts are used.

To use an OTR alias text as the value of a UI element property, perform the following steps:

- Press the value help button in the *Properties* value field. The OTR browser will appear, showing all OTR alias texts of your package and of the \texttt{SOTR_VOCABULARY_BASIC} package, which contains standard texts.
- Select a text from the list and choose *Enter*.
- The OTR directive for using this text will be automatically entered in the *Properties* value field.

The OTR directive is a combination of the text's alias name and the prefix OTR: \texttt{$OTR:<package>/<alias>$}.
Figure 80: Using OTR Alias Texts as UI Element Property Values

OTR short texts having an alias can also be accessed from a controller's source code. The class `CL_WD_UTILITIES` provides appropriate service methods. The method `get_otr_text_by_alias()` allows you to access the value of an OTR short text language dependent by providing the alias. The value returned is of type `string`.

```plaintext
DATA lv_ctr_text TYPE string.
...
* Get OTR short text in language sy-lang by providing its alias
lv_ctr_text = cl_wd_utilities->get_otr_text_by_alias(
  alias = 'NET310/CAPTION'
)
...

* Set language dependent value of single attribute
lo_el_lable_text->set_attribute(
  name = 'GROUP1/CAPTION'
  value = lv_ctr_text
).
```

Figure 81: Accessing OTR Alias Texts from the Controller Source Code
Using Texts Defined in an ABAP Class

Instead of using texts defined in the OTR, it is possible to define the texts as text elements in an ABAP class. To access the text element values, a public method of the class can be used. A class offering methods to read the text elements is already available in each SAP NetWeaver AS 7.0, having the name `CL_WD_COMPONENT_ASSISTANCE`. Each class inheriting from this parent class will have the method `get_text()` . Passing the identifier of the text symbol to the method will return the text in the language `sy-langu`.

To use this class in a Web Dynpro component, it has to be instantiated in each controller. However, an easier and more consistent way is to let the Web Dynpro runtime create the instance only once. This is done by writing the class name in the `Assistance Class` field, which can be found on the `Properties` tab for the Web Dynpro component. The single steps are:

- Edit the Web Dynpro component. Navigate to the `Properties` tab of the component.
- Enter a class name in the `Assistance Class` field.
- Double-click the class name. If the class does not exist, it will be created. The class will be derived from the parent class `CL_WD_COMPONENT_ASSISTANCE`.

![Figure 82: Using the Component Assistance Class](image)

If an assistance class is associated, an instance of this class is automatically instantiated.

This instance can be accessed using the attribute `wd_Assist`, available in every controller of the component.

Figure 82: Using the Component Assistance Class

Text symbols can be added to the class by choosing `Goto Text Symbols`. Be sure that the length of the text elements is long enough to support different languages.

- Optionally, you can define a public alias for the interface method `get_text()` (just for convenience).
- If you want to refer to the text elements using arbitrary identifiers, you can define constants of type `WDR_TEXT_KEY` for each text element. The constant value equals the text element identifier in apostrophes.
Once you have assigned the assistance class to the component, an attribute, `WD_ASSIST`, is added to the attribute list of each controller. This attribute is set by the Web Dynpro runtime to the assistance class instance. Using `WD_ASSIST`, a text can be accessed as follows:

- Using the text element identifier `CAR`:
  ```abap
  DATA: lv_text TYPE string.
  lv_text = wd_assist->if_wd_component_assistance~get_text( key = 'CAR' ).
  ```

- Using a constant `CARRID`, having the text element identifier `CAR` as its value:
  ```abap
  DATA: lv_text TYPE string.
  lv_text = wd_assist->if_wd_component_assistance~get_text( key = wd_assist->carrid ).
  ```

Texts defined in the assistance class can also have placeholders. These are defined by using an ampersand (&), followed by the placeholder's name. In general, there are no restrictions in respect to the number and the names of the placeholders. The method `get_text()` allows you to export values for the placeholders. It returns the corresponding text in the logon language, with the placeholder substituted by the provided values.

⚠️ **Caution:** Only a maximum of four placeholders having the names (including the starting ampersand) `&PARA1&` ... `&PARA4&` are replaced by the method `get_text()` Here, the placeholder names must be defined in capital letters.

If the text is obtained without providing values for the placeholders, the text is returned as defined. The placeholders can then later be substituted by another algorithm (for example, if the text is used as the text of a message).

**Messages and Error Handling**
Messages are used to provide the user with status information of the application, and to display warnings and errors. Web Dynpro allows you to define what kind of message is sent, where it is displayed, and how it is displayed.
Defining the Position of the Message Area

There are two ways to define where messages are to be displayed on the screen:

1. The default position of the message area is on top of the page. If this position is fine, the developer does not have to do anything.

2. If a position different from the default message area position is desired, the MessageArea container UI element has to be used. A MessageArea UI element can be defined as a sub-element of all other container elements.

Figure 83: Positioning of Messages

Message Handling

In the Web Dynpro application, you can set the way messages are displayed. The way of displaying messages depends on the support package level. For SP 11 and higher, all messages are displayed in a scroll container having a fixed height. Messages related to UI elements are displayed as links and the related UI elements are highlighted. If a message link is clicked, the cursor will be set in the related UI element. On the Properties tab of a Web Dynpro application, message handling can be defined by setting the radio button in the group labeled —Handling of Messages“. The following settings are possible:
• **Show Message Component on Demand**
  If at least one message is reported, the message area is displayed.

• **Always Display Message Component**
  The message area is also displayed if no message is reported.

![Image](image.png)

**Figure 84: Message Handling**

**Methods for Defining Messages: Categories**

In order to report a message, methods of the `IF_WD_MESSAGE_MANAGER` interface have to be used. The Web Dynpro runtime automatically instantiates a class implementing this interface. The reference to this class can be obtained from the self reference `WD_THIS`, using the expressions:

```plaintext
lo_api_ctrl = wd_this->wd_get_api().
lo_message_manager = lo_api_ctrl->get_message_manager().
```

All the methods allowed to report methods can be divided into three categories:

• **TEXT**
  Messages belonging to this category allow messages containing an arbitrary text. Translatable texts can originate from the OTR (short texts having an alias), from text elements defined in an ABAP class (preferably the assistance class), or from the ABAP Dictionary.

• **T100**
  These methods use texts defined in database table T100 as messages in the Web Dynpro context.

• **EXCEPTIONS**
  Catchable runtime errors and the related texts can be used when choosing a message of this category.
Figure 85: Methods for Defining Messages: Categories

Methods of categories *TEXT* and *T100* allow you to export parameters in addition to the message text. This allows the use of message texts containing placeholders. All categories methods allow relating the message to an UI element. The message text is then displayed as a link. Clicking on the link sets the cursor in the related field. The field and its content are highlighted.

Summary
The list below summarizes the methods available to report messages.

### Category: TEXT

<table>
<thead>
<tr>
<th>Method (REPORT ...)</th>
<th>Parameters can be used in message text ?</th>
<th>Message can be related to UI element ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTRIBUTE_ERROR_MESSAGE</td>
<td>YES (Parameter PARAMS)</td>
<td>YES (Parameters ELEMENT, ATTRIBUTE_NAME)</td>
</tr>
<tr>
<td>SUCCESS</td>
<td>YES (Parameter PARAMS)</td>
<td>NO</td>
</tr>
<tr>
<td>WARNING</td>
<td>YES (Parameter PARAMS)</td>
<td>NO</td>
</tr>
<tr>
<td>ERROR_MESSAGE</td>
<td>YES (Parameter PARAMS)</td>
<td>NO</td>
</tr>
<tr>
<td>FATAL_ERROR_MESSAGE</td>
<td>YES (Parameter PARAMS)</td>
<td>NO</td>
</tr>
</tbody>
</table>
### Category: EXCEPTIONS

<table>
<thead>
<tr>
<th>Method (REPORT_...)</th>
<th>Parameters can be used in message text?</th>
<th>Message can be related to UI element?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTRIBUTE_EXCEPTION</td>
<td>NO</td>
<td>YES (Parameters ELEMENT, ATTRIBUTE_NAME)</td>
</tr>
<tr>
<td>EXCEPTION</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>FATAL_EXCEPTION</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>

### Category: T100

<table>
<thead>
<tr>
<th>Method (REPORT_...)</th>
<th>Parameters can be used in message text?</th>
<th>Message can be related to UI element?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTRIBUTE_T100_MESSAGE</td>
<td>YES (Parameters P1, P2, P3, P4)</td>
<td>YES (Parameters ELEMENT, ATTRIBUTE_NAME)</td>
</tr>
<tr>
<td>T100_MESSAGE</td>
<td>YES (Parameters P1, P2, P3, P4)</td>
<td>NO</td>
</tr>
</tbody>
</table>
Reporting Messages

The source code for reporting messages can be generated using the Web Dynpro Code Wizard. The generated code consists of a common part, containing the source code for obtaining the reference to the message manager, and a part that is dependent on the method to be called. The message manager cares for collecting and sending the messages reported in the actual component and all sub-components.

Since obtaining the reference to the message manager is obligatory before a message can be reported, the corresponding source code should be moved to the standard hook method wddoinit() and the reference should be stored in a user-defined controller attribute.

```
DATA lo_api_controller  TYPE REF TO if_wd_controller.
DATA lo_message_manager TYPE REF TO if_wd_message_manager.

* get reference to actual component controller
lo_api_controller = wd_this->wd_get_api().

* get reference to message manager
lo_message_manager = lo_api_controller->get_message_manager().
* report message
```

Figure 86: Reporting a Message: Common Part

Category TEXT

All methods for reporting messages of category TEXT have a common interface. At least the message text has to be exported using the parameter message_text. If the message text contains placeholders, the parameter list must be defined before calling the method. Placeholders in message texts can have arbitrary names (in the example below, X1). If a method of category TEXT is to be related to a UI element, a reference to the context element and the name of the attribute in this context element keeping the faulty value have to be provided. Since this attribute is bound to the property of a UI element, the Web Dynpro runtime can relate the message to this UI element.
Figure 87: Example: Reporting a Message: Category TEXT

Category T100
To use texts already defined in database table T100, two methods exist. The message number, the message class, and the message type have to be provided. If the message text contains placeholders, appropriate values can be exported. Depending on the method, all message fragments are exported by using scalar parameters or by using an export structure. At least the message text has to be exported using the parameter `message_text`.

If a method of category T100 is to be related to a UI element, a reference to the context element and the name of the attribute in this context element keeping the faulty value have to be provided. Since this attribute is bound to the property of a UI element, the Web Dynpro runtime can then relate the message to this UI element.
Category EXCEPTIONS
Runtime exceptions can be caught in an exception object, which contains then the text related to the error. This text is defined in the exception class and thus cannot be influenced. The message object can be used to create a Web Dynpro message by calling methods of the category EXCEPTIONS. The message object is exported using the parameter message_object. If a method of category EXCEPTIONS is to be related to a UI element, a reference to the context element and the name of the attribute in this context element keeping the faulty value have to be provided. Since this attribute is bound to the property of a UI element, the Web Dynpro runtime can then relate the message to this UI element.
Coding for Reporting Messages

If validation coding is only to be performed for a certain user action, it can be placed in the corresponding action handler method (\textit{onaction<action>}). However, if multiple action handler methods exist, some validations may have to be performed in more than one of the action handler methods. In this case, it is a good idea to put the coding in an extra method in order to have it written only once. For this reason, a special hook method exists for each view, which is processed before any of the action handler methods is called. The name of this hook method is \textit{wddobeforeaction()}. If a screen is composed of multiple views (nested views), then all \textit{wddobeforeaction()} methods are processed before any action handler method is called.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{wddobeforeaction.png}
\caption{Reporting Messages: Hook Method WDDOBEFOREACTION()}
\end{figure}
The following graphic displays how to find out the name of the action that was triggered by the user. From the controllers API, the method `get_current_action()` is used to receive the reference to an object describing the action. The attribute `name` contains the name of the last action that was triggered.

```abap
DATA lo_api_view TYPE REF TO if wd_view_controller.
DATA lo_action TYPE REF TO if wd_action.

* get reference to controller api of view
  lo_api_view = wd_this->wd_get_api( ).

* get reference to object related to current action
  lo_action = lo_api_view->get_current_action( ).

* depending on name of current action do different
  * things
    CASE lo_action->name.
      WHEN 'DISPLAY'.
        ...
      WHEN 'SAVE'.
        ...
      WHEN OTHERS.
      ...
    ENDCASE.
```

Figure 91: Determining the Name of the current Action

**Reporting Messages: Impact on Navigation and Method Processing**

By default, messages of type *Success*, *Warning*, or *Error* do not influence the navigation. However, all methods contain an interface parameter `CANCEL_NAVIGATION` that allows to change this default behavior. Messages of the kinds listed here have no impact on the order of the controller methods processed by the framework.

If a messages of type *Fatal* is reported in one of a view's event handler methods `wddobeforeaction()` or `onaction<action>()`, then the navigation is cancelled. All the view's subsequent hook methods (besides `wdomodifyview()` ) are skipped. The source code of the hook method containing the method call (used to report the fatal message) is only processed up to the call.
For messages that are related to context attributes, the behavior depends not only on the name of the hook method containing the method call, but also on the property *Action Type* of the triggered action. By default the property *Action Type* is set to validation dependent (*standard*). In this case the navigation is cancelled if the message is reported in any of the hook methods `wddobeforeaction()`, `onaction<action>()`, or `wddoafteraction()`. In addition, the method `wddomodifyview()` is skipped. If the message is reported in `wddobeforeaction()` , the methods `onaction<action>()` and `wddoafteraction()` are also skipped. This default behavior can be changed by means of the interface parameter `IS_VALIDATION_INDEPENDENT` existing for all methods used to report context attribute related messages.

If the *Action Type* is set to *validation-independent*, then the navigation is not cancelled. Reporting the message does not influence the number and order of the hook methods processed by the Web Dynpro runtime.
Exercise 9: Internationalization: 
Translatable Text in the UI

Exercise Objectives

After completing this exercise, you will be able to:

- Create OTR texts
- Use OTR texts for UI elements

Business Example

You want to develop a Web Dynpro application where all the UI texts can be translated. For those texts that are not retrieved from ABAP Dictionary, you will use the OTR.

Template: VCT_CONR_S3
Solution: VCT_I18N_S1
Task:
Copy your Web Dynpro component ZVCT_CONR3_## or the template VCT_CONR_S3 to Web Dynpro component ZVCT_I18N1_##.

Use OTR texts for group captions and button texts.

1. Copy the template.
2. Create the required OTR texts in your package (identifiers starting with ZVCT_##/).
3. For all UI elements of type GROUP or BUTTON, set property text with reference to an OTR text.
Solution 9: Internationalization: Translatable Text in the UI

Task:
Copy your Web Dynpro component ZVCT_CONR3_## or the template VCT_CONR_S3 to Web Dynpro component ZVCT_I18N1_##.

Use OTR texts for group captions and button texts.
1. Copy the template.
   a) Perform this step as in previous exercises.
2. Create the required OTR texts in your package (identifiers starting with ZVCT_##/).
   a) Choose Goto Online Text Repository Browser.
   b) Choose Create.
   c) Enter Identifier, length, and text. Choose Save.
3. For all UI elements of type GROUP or BUTTON, set property text with reference to an OTR text.
   a) Edit the respective UI element and open the value help for property text.
   b) Double-click the appropriate OTR text on the list.
Exercise 10: Value Checks and Messages

Exercise Objectives
After completing this exercise, you will be able to:

• Implement value checks after user input
• Access the message controller and send messages
• Send messages with reference to UI elements
• Provide values for placeholders

Business Example
You want to develop a Web Dynpro application with a view that offers fields for user input. You want to check the user input and send messages in case of missing or wrong values. If the messages are related to a single UI element, this should be visible to the user.

Template : VCT_I18N_S1
Solution : VCT_I18N_S2

Task 1:
Copy your Web Dynpro component ZVCT_I18N_II or the template VCT_I18N_S1 to Web Dynpro component ZVCT_I18N_II. In the controller of the INPUT_VIEW view, create a new method in which you will implement the value checks. Make sure the new method is processed after the user has pressed the button, but before the outbound plug of the view is fired.

1. Copy the template.
2. In the controller of the INPUT_VIEW view, create a new method, (suggested name: CHECK_INPUT ) in which you will implement the value checks.
3. Within method WDDOBEFOREACTION, implement a call of the new method.

Task 2:
Implement the new method. The first step is to read all attributes of context node FLIGHTINFO.

1. Use the Web Dynpro Code Wizard to read all attributes of context node FLIGHTINFO.

Continued on next page
Task 3:
Implement checks to make sure the user has made an entry in both input fields. If a check fails, send an error message with reference to the respective input field. Use text symbols from assistance class CL_VCT_I18N_S as message texts.
1. Assign the assistance class to your Web Dynpro component.
2. Navigate to the text symbols and obtain the identifiers of appropriate texts.
3. In method CHECK_INPUT, call method IF_WD_COMPONENT_ASSISTANCE~GET_TEXT of the assistance class instance to retrieve the texts you need.

Hint: Use attribute WD_ASSIST of the view controller.

4. Implement the checks and, in each case (carrier not supplied, connection ID not supplied), call method REPORT_ATTRIBUTE_ERROR_MESSAGE with the respective message text.
5.
Task 4:
Implement a check to make sure that some flights will be found for the selection. Do not relate the error message to an input field. Use a text symbol as message text and replace the placeholders with the user input.
1. Check whether or not at least one entry that satisfies the condition is found in database table SFLIGHT.

Hint: Use the static method CL_VCT_FLIGHT-MODEL=>CHECK_FLIGHT_EXIST() to perform this check.

2. If no flights are found (parameter E_EXIST <> 'X'), retrieve an appropriate text from the assistance class and call method REPORT_ERROR_MESSAGE of the message manager.
3. Declare and fill an internal table (suggested name: lt_params ) to supply an actual parameter for parameter PARAMS of method REPORT_ERROR_MESSAGE.

Alternative:
Provide values for the placeholders in the call of method get_text .

Caution: If you choose the alternative, be aware that you must use a text symbol with place holders &PARA1&, &PARA2&, and so on.
Solution 10: Value Checks and Messages

Task 1:
Copy your Web Dynpro component ZVCT_I18N1_## or the template VCT_I18N_S1 to Web Dynpro component ZVCT_I18N2_##.

In the controller of the INPUT_VIEW view, create a new method in which you will implement the value checks. Make sure the new method is processed after the user has pressed the button, but before the outbound plug of the view is fired.

1. Copy the template.
   a) Perform this step as in previous exercises.
2. In the controller of the INPUT_VIEW view, create a new method, (suggested name: CHECK_INPUT ) in which you will implement the value checks.
   a) Perform this step as in previous exercises.
3. Within method WDDBEFOREACTION, implement a call of the new method.
   a) Perform this step as in previous exercises.

Task 2:
Implement the new method. The first step is to read all attributes of context node FLIGHTINFO.

1. Use the Web Dynpro Code Wizard to read all attributes of context node FLIGHTINFO.
   a) Perform this step as in previous exercises.

Task 3:
Implement checks to make sure the user has made an entry in both input fields. If a check fails, send an error message with reference to the respective input field. Use text symbols from assistance class CL_VCT_I18N_S as message texts.

1. Assign the assistance class to your Web Dynpro component.
   a) Edit the Web Dynpro component and enter the name of the class in the assistance Class field.
2. Navigate to the text symbols and obtain the identifiers of appropriate texts.
   a) From the Web Dynpro view, choose Goto Text Symbols.

Continued on next page
3. In method CHECK_INPUT, call method IF_WD_COMPONENT_ASSISTANCE~GET_TEXT of the assistance class instance to retrieve the texts you need.

**Hint:** Use attribute WD_ASSIST of the view controller.

   a) See source code of model solution. Use the pattern for calling methods to generate the source code.

4. Implement the checks and, in each case (carrier not supplied, connection ID not supplied), call method REPORT_ATTRIBUTE_ERROR_MESSAGE with the respective message text.

   a) See the source code of the model solution.

   b) Create the first method call by using the Web Dynpro Code Wizard.

This will also create the statements to obtain the reference to the message manager. For all other calls, use the pattern for calling methods so that only the method call statement will be generated.

**Task 4:**
Implement a check to make sure that some flights will be found for the selection. Do not relate the error message to an input field. Use a text symbol as message text and replace the placeholders with the user input.

1. Check whether or not at least one entry that satisfies the condition is found in database table SFLIGHT.

   **Hint:** Use the static method CL_VCT_FLIGHT-MODEL=>CHECK_FLIGHT_EXIST() to perform this check.

   a) See source code of the model solution.

2. If no flights are found (parameter E_EXIST <> 'X'), retrieve an appropriate text from the assistance class and call method REPORT_ERROR_MESSAGE of the message manager.

   a) See source code of the model solution.

3. Declare and fill an internal table (suggested name: lt_params) to supply an actual parameter for parameter PARAMS of method REPORT_ERROR_MESSAGE.

   **Alternative:**
   Provide values for the placeholders in the call of method get_text.

*Continued on next page*
Caution: If you choose the alternative, be aware that you must use a text symbol with place holders \&PARA1\& , \&PARA2\& , and so on.

a) See source code of the model solution.

Result

Model Solution: VCT_I18N_S2, Method CHECK_INPUT

METHOD check_input.

DATA lo_nd_flightinfo TYPE REF TO if_wd_context_node.
DATA lo_el_flightinfo TYPE REF TO if_wd_context_element.
DATA ls_flightinfo TYPE wd_this->element_flightinfo.

DATA lo_api_controller TYPE REF TO if_wd_controller.
DATA lo_message_manager TYPE REF TO if_wd_message_manager.

DATA lt_params TYPE wdr_name_value_list.
DATA ls_param LIKE LINE OF lt_params.
DATA lv_conoid TYPE c LENGTH 4.
DATA lv_text TYPE string.
DATA lv_exist TYPE c.

* Read context attributes of element at lead selection
  lo_nd_flightinfo = wd_context->get_child_node( name = wd_this->wdctx_flightinfo ).
  lo_el_flightinfo = lo_nd_flightinfo->get_element( ).
  lo_el_flightinfo->get_static_attributes( IMPORTING
    static_attributes = ls_flightinfo ).

* get message manager
  lo_api_controller ?= wd_this->wd_get_api( ).
  lo_message_manager = lo_api_controller->get_message_manager( ).

******************************************************************************CHECKS******************************************************************************

IF ls_flightinfo-carrid IS INITIAL.
  Continued on next page
lv_text = wd_assist->if_wd_component_assistance~get_text( key = '001' ).
* report message
lo_message_manager->report_attribute_error_message(
message_text = lv_text
element = lo_el_flightinfo
attribute_name = 'CARRID' ).
ENDIF.

IF ls_flightinfo-connid IS INITIAL.
lv_text = wd_assist->if_wd_component_assistance~get_text( key = '002' ).
* report message
lo_message_manager->report_attribute_error_message(
message_text = lv_text
element = lo_el_flightinfo
attribute_name = 'CONNID' ).
ENDIF.

* does flight exist?
cl_VCT_flightmodel=>check_flight_exist(
EXPORTING
i_carrid = ls_flightinfo-carrid
i_connid = ls_flightinfo-connid
IMPORTING
e_exist = lv_exist ).

* flight does not exist
IF lv_exist <> 'X'.

ls_param-name = '1'.
ls_param-value = ls_flightinfo-carrid.
APPEND ls_param TO lt_params.

ls_param-name = '2'.
ls_param-value = ls_flightinfo-connid.
APPEND ls_param TO lt_params.

lv_text = wd_assist->if_wd_component_assistance~get_text( key = '003' ).
* report message
lo_message_manager->report_error_message(
message_text = lv_text
params = lt_params ).

Continued on next page
*>>> Alternative: Replace parameters in method get_text
* Attention: Parameters have to be &PARA1&, &PARA2& etc.

* lv_connid = ls_flightinfo-connid.
* lv_text = wd_assist->if_wd_component_assistance~get_text(
  * key = '004'
  * para1 = ls_flightinfo-carrid
  * para2 = lv_connid ).
* 
  ** report message
  * lo_message_manager->report_error_message(
    * message_text = lv_text ).
*<<<

ENDIF.

ENDMETHOD.
Value Help and Semantic Help

Value Help for Input Fields
If the user should not be restricted to a given value set, or if the value set is large and the user needs filter functions to find a certain value, a value help related to an input field is the right choice. This value help always gathers its value set from the meta data of the bound attribute (key binding).

Figure 92: Different Types of Value Help for Input Fields
The developer can influence the value help by setting the property *Input Help Mode* of the context attribute bound to the input field. There are five settings for this property:

- **Deactivated**
  No value help is available. This is independent from the referred data type.
- **Automatic**
  The value help is determined by the Web Dynpro runtime.
- **Dictionary Search Help**
  The developer can choose a search help defined in the ABAP Dictionary.
- **Object Value Selector**
  A predefined, configurable Web Dynpro component implementing a search help can be used to define the search help dialog.
- **User-Defined Programming**
  A user defined Web Dynpro component implementing a search help dialog is used to define the search help.

**Figure 93: Input Help Modes**

**Determining the Value Help for Input Help Mode Automatic**

If the property *Input Help Mode* is set to *Automatic*, the Web Dynpro runtime decides which value help will be displayed (and if a value help is displayed at all). The mechanism of determining the value help is similar to the one used to determine the value help for input fields on classical Dynpro screens. However, the name and kind of value help is determined only once - when selecting this kind of value help at design time.

Depending on the attribute type, the value help is identified as follows:

1. If the field of a flat structure defined in the ABAP Dictionary has been used to type
the context attribute and a search help is related to this structure field, then this search help will be displayed.

2. Next, the Web Dynpro runtime checks if a foreign key relationship is defined for the structure field used to type the context attribute. If this is true and the check table is assigned a search help, this search help will be used.

3. If a foreign key relationship is defined but no search help is assigned to the check table, then the key fields of the check table are displayed.

4. Next, the value help related to the underlying data element is displayed.

5. Finally, fixed values or the fixed value range related to the underlying domain may be displayed.

If none of the above possibilities to determine a search help is successful, no value help is displayed.

**Hint:** Not only context attributes but also context nodes can be typed with a Dictionary structure. By this way, the type of multiple context attributes can be related to fields belonging to the same structure. Only then, a search help imports the input of multiple input fields to build up the value list and only then, selecting a value from this list will fill more than one input field.

![Diagram](image)

**Figure 94: Determining the Value Help for Input Help Mode AUTOMATIC**

If a data element is used to type the context attribute, a value help is only displayed if fixed values or a fixed value range is assigned to the underlying domain or if a simple search help is assigned to the data element.

If a Web Dynpro runtime type is referenced, the fixed values related to this type are displayed by the value help. If the Web Dynpro runtime types `DATA`, `F`, `I`, `OBJECT`, `STRING`, `STRING_TABLE`, or `XSTRING` are referenced, no value help is offered.
Value Help for Input Help Mode  Dictionary Search Help

Sometimes the automatic detection of the value help does not lead to the desired result. Then, the property Input Help Mode can be set to Dictionary Search Help. This way, the developer can assign any search help to the context attribute that has one import parameter and one export parameter.

Figure 95: Value Help for Input Help Mode DICTIONARY SEARCH HELP

Value Selectors

Value A value selector is a value help that allows a user to choose one value or multiple values from a predefined value set. The user is not allowed to enter values different from the ones displayed by the value selector. Typical UI elements to implement value selectors are dropdown boxes, radio button groups, and item list boxes. Value selectors should only be used if the number of values to be displayed is low (typically less than 30).
Figure 96: Overview: Value Selectors

The values displayed by the value selector can be stored in the context, or they can be related to the type to which the UI element is referring. In the first case, the UI element has to be bound to an attribute that is a child of a context node of cardinality 0..n or 1..n. Before the view (containing the value selector) can be rendered, the collection has to be populated. This means all values to be displayed in the value selector are to be stored in the context as attribute values of the collection elements. This kind of data binding is called index binding.

Figure 97: Value Selector Using Index Binding
Selecting a value leads to a round trip only if the corresponding event property is bound to an action.
Selecting a value will change the lead selection.
The data stored in the context will not be changed by selecting any value. The second possibility to define a value set displayed by the value selector is to relate the value set to the type to which the UI element is referring. In this case, the UI element has to be bound to a context attribute, which is typed as follows:
The attribute is an ABAP Dictionary type, which is based on a domain with fixed values. The short texts related to the fixed values are displayed by the value selector.
The attribute is a Web Dynpro runtime type with fixed values. These fixed values are displayed by the value selector.
The attribute's type is arbitrary. The value set is assigned dynamically by changing the attribute's meta data at runtime.

This kind of data binding is called **key binding**.

![Figure 98: Value Selector Using Key Binding (ABAP DDIC Example)](image)

The following statements apply to this binding type:

- Selecting a value leads to a round trip only if the corresponding event property is bound to an action.
- Selecting a value will **not** change the lead selection.
- The data stored in the context element at lead selection will be overwritten by the key value related to the selected data.
Dynamically Assigning Value Sets to Context Attributes

If a value selector is implemented using the key binding technique, the value set can be defined at runtime. This is done by manipulating the meta data of the attribute, which is bound to the UI element having the value selector. The value set has to be defined as an internal table having the line type `WDR_CONTEXT_ATTR_VALUE`. The reference to the meta data of a context node is determined using the method `get_node_info()` of the nodes reference. Assigning the value set to the meta data of a context attribute is established using the method `set_attribute_value_set()` of the reference to the node's meta data.

```
DATA:
  lo_nd_sbook TYPE REF TO if_wd_context_node,
  lo_nd_info TYPE REF TO if_wd_context_node_info,
  ls_value_set TYPE WDR_CONTEXT_ATTR_VALUE,
  lt_value_set TYPE TABLE OF WDR_CONTEXT_ATTR_VALUE.

* get object reference to metadata of node SBOOK
  lo_nd_sbook = wd_context->get_child_node( name = 'SBOOK' ).
  lo_nd_info = lo_nd_sbook->get_node_info().

* define value set
  ls_value_set-value = 'Y'.
  ls_value_set-text = 'Extra Class'.
  INSERT ls_value_set INTO TABLE lt_value_set.
  ...

* hand over value set to metadata of context attribute CLASS
  lo_nd_info->set_attribute_value_set( 
    name = 'CLASS'
    value_set = lt_value_set ).
```

Figure 99: Value Selector: Assigning the Value Set dynamically
Exercise 11: Search Help from ABAP Dictionary and Dropdown List Box

Exercise Objectives
After completing this exercise, you will be able to:
• Make use of ABAP Dictionary search help
• Assign individual value sets to input fields
• Provide value help as a dropdown list box

Business Example
You want to develop a Web Dynpro application where you offer the user value help for input fields. You will either use search help defined in the ABAP Dictionary or individually defined value sets. In the case of individual value sets, you want to display the input help as a dropdown list box.

Template: VCT_I18N_S2
Solution: VCT_HLP_S1

Task 1:
Copy your Web Dynpro component ZVCT_I18N_## or the template VCT_I18N_S2 to Web Dynpro component ZVCT_HLP1_##. Activate automatic usage of ABAP Dictionary search help for the two input fields.

1. Copy the template.
2. In the component controller context, change the property Input Help Mode to Automatically for the relevant context node attributes.
3. What are the technical names of the search helps that are automatically determined and where do they come from?

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Determined Search Help</th>
<th>Origin of the input help</th>
</tr>
</thead>
<tbody>
<tr>
<td>CARRID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONNID</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
4. Create a Web Dynpro application, activate the component, and test the input helps.

**Task 2:**
Assign an individual value set to attribute CARRID of view context node FLIGHTINFO.

1. In the controller of the view INPUT_VIEW, create a new method (suggested name: `VS_CARRID`).
2. Make sure the new method is processed during initialization of the view (that is, in method WDDOINIT).
3. Implement the new method. Obtain a reference to context node FLIGHTINFO, then call method GET_NODE_INFO to obtain a reference to the node's info object.
4. Call the static method `CL_VCT_FLIGHTMODEL=>READ_CARRIERS_VS()` to obtain a value set consisting of the key CARRID and the describing information CARRNAME for all flight carriers. Declare an internal table of appropriate type and fill this internal table by the static method.
5. Sort the internal table content by the TEXT column.
6. Call method `SET_ATTRIBUTE_VALUE_SET` to assign the value set to the node info object.

**Task 3:**
To choose the carrier, offer a dropdown list box instead of an input field.

1. Replace the input field for the carrier ID with a dropdown list box, that is, an UI element of type DROPOUT_DOWN_BY_KEY. Bind this UI element to the same context node attribute.
Solution 11: Search Help from ABAP Dictionary and Dropdown List Box

**Task 1:**
Copy your Web Dynpro component ZVCT_I18N2_## or the template VCT_I18N_S2 to Web Dynpro component ZVCT_HLP1_##.

Activate automatic usage of ABAP Dictionary search help for the two input fields.

1. Copy the template.
   a) Perform this step as in previous exercises.

2. In the component controller context, change the property *Input Help Mode* to **Automatically** for the relevant context node attributes.
   a) Edit the component controller. Choose the *Context* tab.
   b) Double-click attribute CARRID of context node FLIGHTINFO and use the dropdown list box to set the value of property *Input Help Mode*.
   c) Repeat for attribute CONNID of context node FLIGHTINFO.

3. What are the technical names of the search helps that are automatically determined and where do they come from?

<table>
<thead>
<tr>
<th>Attribute Determined</th>
<th>Origin of the input help</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
<td>Determined Search Help</td>
</tr>
<tr>
<td>CARRID</td>
<td>H_SCARR</td>
</tr>
<tr>
<td>CONNID</td>
<td>H_SPFLI</td>
</tr>
</tbody>
</table>

4. Create a Web Dynpro application, activate the component, and test the input helps.
   a) Perform this step as in previous exercises.

**Task 2:**
Assign an individual value set to attribute CARRID of view context node FLIGHTINFO.

1. In the controller of the view INPUT_VIEW, create a new method (suggested name: VS_CARRID).
   a) Perform this step as in previous exercises.

*Continued on next page*
2. Make sure the new method is processed during initialization of the view (that is, in method WD ООINIT).
   a) Perform this step as in previous exercises.
3. Implement the new method. Obtain a reference to context node FLIGHTINFO, then call method GET_NODE_INFO to obtain a reference to the node's info object.
   a) See source code of model solution.
4. Call the static method `CL_VCT_FLIGHTMODEL=>READ_CARRIERS_VS()` to obtain a value set consisting of the key CARRID and the describing information CARRNAME for all flight carriers. Declare an internal table of appropriate type and fill this internal table by the static method.
   a) See source code of model solution.
5. Sort the internal table content by the TEXT column.
   a) See source code of model solution.
6. Call method SET_ATTRIBUTE_VALUE_SET to assign the value set to the node info object.
   a) See source code of model solution.

**Task 3:**
To choose the carrier, offer a dropdown list box instead of an input field.

1. Replace the input field for the carrier ID with a dropdown list box, that is, an UI element of type DROPDOWN_BY_KEY. Bind this UI element to the same context node attribute.
   a) Open context menu for the UI element of type GROUP and choose *Create Container Form*.
   b) For context attribute CARRID change the value in the column with the header *Cell Editor of Table Column* to *DropDownByKey* and confirm your entry.
   c) Rearrange and realign the UI elements within the group.
Result

Model Solution: VCT_HLP_S1, Method VS_CARRID

METHOD vs_carrid .

Continued on next page

DATA lo_nd_flightinfo TYPE REF TO if_wd_context_node.
DATA lo_nd_info_flightinfo TYPE REF TO if_wd_context_node_info.
DATA lt_value_set TYPE TABLE OF wdr_context_attr_value.

* get node info for node <FLIGHTINFO>
lo_nd_flightinfo = wd_context->get_child_node(
  name = wd_this->wdctx_flightinfo ).
lo_nd_info_flightinfo = lo_nd_flightinfo->get_node_info( ).

* get value set (VALUE = CARRID , TEXT = CARRNAME)
cl_VCT_flightmodel=>read_carriers_vs(
IMPORTING
e_carrier_vs = lt_value_set ).

* sort the value set by the describing TEXT
SORT lt_value_set BY text.

* assign value set to context attribute
lo_nd_info_flightinfo->set_attribute_value_set(
  name = 'CARRID'
  value_set = lt_value_set ).

ENDMETHOD.
Lesson: Semantic Help

Field dependent Help Texts
There are several techniques to relate a help text to a given UI element:
• Tool tips
• Explanation texts
• Standard Dictionary F1 help

Tool Tips

Figure 100: Defining and Displaying Tool Tips

Tool tips allow to display a short text (maximum length: 255) in a yellow box. This box appears if the mouse cursor is moved on the field, the tool tip is assigned to. If the mouse cursor is moved out of the area occupied by the field, the tool tip vanishes. The tool tip can be assigned to a UI element by setting the property `tooltip` accordingly. The text can either be typed in the property field, it can originate from an alias text defined in the Online Text Repository (OTR), it can be the content of a context attribute, or it can be one of the texts related to a data element. Tool tips are inherited to other UI elements according to their position in the UI element hierarchy of the view.
Explanation Texts

Figure 101: Creating an Explanation Text

An explanation text is displayed if the user clicks in a UI element that has this explanation text assigned to it or if the user moves the mouse cursor on the label related to that UI element. If an explanation text is assigned to a UI element, then the related label is underlined green.

The explanation text can be assigned to a UI element by setting the property explanation accordingly. The text (maximum length: 255) can either be typed in the property field, or it can originate from an alias text defined in the Online Text Repository (OTR).

If the primary property of a UI element (e.g. value property of InputField) is bound to a context attribute, then the text can also originate from the data element related to this context attribute. The developer can decide if the short text stored in the data element, if the content of the key block &DEFINITION& , or if the content of the key block &USE& is to be displayed (these key blocks are taken from the data element documentation).
Figure 102: Switching on and off Explanation Texts

Displaying the explanation texts can be switched on and off from the context menu that appears when the user clicks anywhere on the screen. The explanation property can be set for the following UI elements: Abstract Table Column, Button, Check Box, Drop Down By Index, Drop Down By Key, File Upload, Input Field, Item List Box, Radio Button, Text Edit, Tri State Check Box.
Classic F1 Help

By default, the classic F1 help derived from the data element’s documentation is available for all UI elements having their primary property bound to a context attribute that is typed accordingly.

To display the F1 help, the cursor has to be placed in the field and the key combination \textit{CTRL + F1} has to be pressed. Another possibility to display the F1 help is provided by right mouse click in the field and choose \textit{More Field Help} from the context menu.

In the modal dialog box popping up, the data element documentation is displayed. In addition, a link allows to navigate to the technical information of the field. If no data element documentation is available, the technical information related to the field is displayed directly.

The data element documentation can be suppressed by adding the parameter \texttt{WDHIDEMOREFIELDHELPASDEFAULT} (value = \texttt{X}) to the application parameters. In this case only the technical field information is displayed as F1 help. However, this behavior can be changed for single fields as follows: If an explanation text is assigned to the field and this explanation text originates from the ABAP Dictionary, then the F1 help works (for this field) as if the application parameter \texttt{WDHIDEMOREFIELDHELPASDEFAULT} was set to \texttt{space} (default).
Field independent Help Texts
To display one help text for multiple fields or to display help information field independent, two techniques do exist:

- Display EXPLANATION UI element
- Display knowledge warehouse (KW) documents in the Help Center

Using the Explanation UI Element

![Image of Explanation UI Element](image.png)

Figure 104: Displaying long Texts by EXPLANATION UI Element

The UI element Explanation can be used to display a text that appears on the screen permanently and should contain explanations about the screen or parts of the screen.

The text to be displayed can be provided as follows:

Using the property `text`, a statical text can be assigned. Using the value help for this property field, the text (alias text) can also be loaded from the OTR. However, binding this property to a context attribute is not possible. In addition, the text cannot be read from an ABAP Dictionary data element either. Thus, if texts are assigned to the Explanation UI element as described above the maximum length of these texts is restricted to 255 characters.
There are different ways of assigning longer texts to the Explanation UI element. If the support package level of the SAP NetWeaver AS 7.0 < 11, the text to be displayed can be loaded in a controller method and stored in a corresponding variable. The value of this variable can then be assigned to the property text of the UI element dynamically. This has to be done in the hook method wddomodifyview() of the view containing the Explanation UI element. This procedure is not only restricted to simple text literals.

The text to be displayed can also be a documentation object. This can be loaded by calling the function module DOCU_GET. The name of the document, the language, and the object id (TX) has to be submitted to the function module. This method returns the documentation object and meta information about the object. These data can be passed to the statical method cl_wd_formatted_text=>create_from_sapscript() in order to create an object containing the text to be passed to the explanation UI element (attribute M_XML_TEXT of created object).
Figure 106: Assigning Document's Name to Property TEXTDOCUMENTNAME

If you want to create or change a document to be displayed by the Explanation UI element, you can call the function module `DOCU_CREATE`. Choose `General Class` when being asked for the documentation class and switch to the SAP Script editor to maintain the document content.

For a support package level = 11, the Explanation UI element allows to assign the name of the document to be displayed via the property `textDocumentName`. This name can be entered directly in the property field or it can be selected using the help dialog that appears when the value help button related to the property field is pressed.

There are two ways to hide the explanation at runtime. Displaying the explanation texts can be switched on and off from the context menu that appears when the user clicks anywhere on the screen. If the property `design` of the Explanation UI element is set to `emphasized`, then an additional link appears right of the element's text. Pressing this link will also let the text vanish.
Displaying KW documents in the Help Center
Documents created in the Knowledge Warehouse can be assigned to a Web Dynpro application or to a window as the object's help text (info object). To do this, the following steps are necessary:

• First the Properties tab of the application or of the window has to be edited. Here, two fields having the labels Help Menu Text and Help Link can be found.
• In the field with the label Help Menu Text a text can be entered that will be displayed as the title of the help window.
• In the field with the label Help Link the link to the info object in the Knowledge Warehouse system has to be specified using the icon right of the input field (Create/Change Link).

Hint: To do this you need to define the RFC connection AIO_FOR_HELP_LINKS for your Knowledge Warehouse system.

The first step of selecting a KW info object is to define the context (language, release, and enhancement) appropriate for the productive system. On the next screen the correct area (for instance documentation or training) has to be chosen. On the third screen, the object has to specified by defining selections (e.g. for the technical name, the title, the developer ...). Finally, all corresponding objects are displayed. Double clicking on a list entry will close the dialog and the link to the selected info object will be entered in the field with the label Help Link.

At runtime, the info object can be called by choosing F1 or the help button in the title bar. The help center opens in a new browser window and the info object for the respective application or window is displayed. The explanation for the quick help, the Knowledge Warehouse documentation, links to the SAP Library, and links defined for the application windows are contained in the help center. The help center can also be opened as a result of a user action. In this case, the appropriate source code can be put in an action handler method that is triggered by the client side event. It is also possible to replace the statically assigned info object by another object by typing source code. For details please refer to the online documentation.
Unit 8

Component Reuse

Web Dynpro Component Usage: Overview
Usually, real business applications based on the Web Dynpro programming model consist of several Web Dynpro components. This is because some functionality is needed in more than one application (for example, customer address display or order details display) or because some functionality is provided by the Web Dynpro framework by means of generic Web Dynpro components (WDR_OVS for defining search helps, WDR_MESSAGE_AREA to display messages, SALV_WD_TABLE for displaying mass data in the SAP List Viewer, or WDR_SELECT_OPTIONS for defining select options in Web Dynpro).

Another common scenario is that multiple components may be embedded in a consumer component. Thus, at design time, the developer has to prepare the application to reuse each of the possible sub components. However, at runtime the user decides (for example by pressing a link) which one of the potential sub components will be embedded. By this technique, multiple dependent information may be displayed in a consumer component's view, depending on the user's choice.
Component Interface
External access to the functionality of a component is provided by the interface controller and by the interface views. Each component has exactly one interface controller and an arbitrary number of interface views. Components having no visual interface are called \textit{faceless components}.

The interface controller may contain ordinary methods, events, and context nodes, defined in the component controller.

In order to expose ordinary methods and events to the interface controller, the developer has to mark the corresponding checkbox in the \textit{Interface} column. In order to expose context nodes defined in the component controller to the interface controller, the checkbox related to the node's \textit{InterfaceNode} property must be selected. If the checkbox for property \textit{Input Element (Ext.)} is also selected, the node \textbf{has to} be mapped to a node defined in a controller of a consumer component.

\textbf{Hint}: If a context node is exposed to the component's interface, all of its sub-nodes are also exposed to the component's interface.

For each window defined in a component, an interface view is generated. Each window can contain inbound plugs and outbound plugs that may be exposed to the interface view. In order to expose the plugs to the interface view, the developer has to mark the corresponding checkbox in the \textit{Interface} column.
Component Usage

The following section explains how the functionality of one component can be used by another component.

Declaring a Component Usage

If one component (consumer component) needs to access the functionality of another component (used component), it has to declare the usage of this component. This is done on the Used Components tab of the consumer component. The name entered in the Component Use column will be a fragment of method names that will be used at runtime to access the used component's functionality from any controller of the consumer component.
Declaring a Component Usage

Instantiating and Deleting a Component Usage

Declaring the usage of a component does not imply that an instance of this component usage is created automatically. The Web Dynpro framework instantiates a component usage if an interface view of this component usage is embedded in the view assembly of the consumer component and if no interface method is called before this interface view is displayed the first time. If this is not the case, the component usage has to be instantiated manually. The reference to a component usage with the name `<comp_usage>` can be obtained from the variable `wd_this` by calling the method `wd_cpuse_<comp_usage>()`.

This reference offers the method `has_active_component()` to find out if an instance of the component usage exists. To instantiate the component usage, call method `create_component()`.

**Hint:** The Web Dynpro Code Wizard can be used to generate the coding necessary to instantiate a component usage.

By default, the component usage instance lives as long as the consumer component lives. However, it is also possible to delete an instance of a component usage manually by calling the method `delete_component()`.
Figure 111: Creating and Deleting a Component Usage Instance

Interface Views
The interface view is the standardized mechanism by which all view assemblies can be presented through the generic Web Dynpro framework. By means of the interface view, a component’s visual interface becomes a reusable unit, thus allowing you to embed it into a view container defined in a view of a consumer component.

The inbound plugs of the interface view can be connected to the outbound plugs of views or a window defined in the consumer component. This allows you to embed different interface views dependent on the outbound plug fired in the consumer component. Information can be passed to the used component via the plug's parameters.

The outbound plugs of the interface view can be connected to inbound plug of views or a window defined in the consumer component. Using the plug's parameters, the used component can hand back data to the consumer component (for example, information about the view assembly displayed by the interface view when leaving the used component).
Interface Controller Methods

Methods defined in the interface controller of a used component can be called from any controller of the consumer component after the controller usage has been instantiated. Since the method is defined in the used component's interface, the controller in the consumer component has to determine the reference to the interface controller instance to be able to call the method. This reference can be obtained from the variable \texttt{wd_this} by calling the method \texttt{wd_cpifc_<used\_component>()}.

**Hint:** The Web Dynpro Code Wizard can be used to generate the coding necessary to call an interface controller method.
Figure 113: Calling Interface Controller Methods

Interface Controller Events
Methods defined in any controller of the consumer component can register to events that are defined in the interface controller of a used component. However, a prerequisite is the declaration of the used component's interface controller as a used controller for the controller containing the event handler method.

The used component can pass information to the consumer component using the parameters of the event.

Example: This concept is used for implementing the Object Value Selector (OVS). Here, the OVS component raises an event every time information is needed from the consumer component. A method in a controller of the consumer component has to register for this event. Event parameters are used to inform the consumer component about what kind of information is needed. Finally, the information is passed back to the OVS component by calling interface controller methods.
Figure 114: Interface Controller Events

Figure 115: Registering for Interface Controller Events
Cross-Component Context Mapping

There are two ways to map context structures defined in different components:

- The mapping origin is defined in the interface controller of the used component (direct context mapping).
- The mapping origin is defined in a component controller or in a custom controller of the consumer component (external context mapping).

Direct context mapping is typically used if the used component provides data needed in the consumer component. External context mapping is used if the used component needs information from the consumer component to perform its task.

**Hint:** When using direct context mapping, the lifetime of the used component instance is of importance. Deleting this instance means that the mapping origin will no longer be available.

To establish direct context mapping, a controller of the consumer component needs to access a context node defined in the interface controller of the used component instance. Thus, a controller usage for the interface controller has to be added to the list of used controllers defined on the Properties tab of the consumer component's controller.

![Figure 116: Cross-Component Context Mapping](image-url)
To establish external context mapping, the interface controller of the component usage instance needs to access the context of a consumer component's controller. Thus, a controller usage for the consumer component's controller has to be defined for the interface controller of the used component instance. This is a bit more complicated, since an interface controller usage is not defined automatically when declaring a component usage. The single steps of defining external context mapping are:

- Edit the properties of the context node in the used component's component controller, which should be used as the mapping target. Select the checkboxes for the Interface Node and Input Element (ext.) properties.
- Define this component as a used component in the consumer component.
- Create an interface controller usage for the component usage. This is done from the context menu in the navigation tree of the consumer component.
- Double-click the interface controller. In the object window, the interface controller usage can be edited.
- On the Properties tab of the interface controller usage, add the controller of the consumer component containing the mapping origin to the list of used controllers.
- On the Context tab, the mapping can now be established. Drag and drop the node of the interface controller to the node acting as the mapping origin (from left to right).

**Dynamic Component Usage**

In complex Web Dynpro applications, it is often desirable to embed interface views of other Web Dynpro components, depending on the user's choice at runtime. For the developer, this means that it is not known at design time which one of multiple possible sub components will be used at runtime.

Programming is significantly simplified, if all components that may be used at runtime offer the same component interface. Then, the consumer component can pass information via well-known interface parameters of well-known methods, or methods defined in the consumer component can register for well-known events defined in any used component.

To assure, that all used components implement the same interface, Web Dynpro component interfaces can be created as independent development objects. A Web Dynpro component interface consists of methods, events, and a context. Interface views can be added using the context menu.
Each Web Dynpro component can implement an arbitrary number of Web Dynpro component interfaces as follows:

- The names of the component interfaces to be implemented have to be added to the component using the *Implemented Interfaces* tab.
- On the same tab, the *Reimplement* button has to be pressed for each component interface. This will *copy* the component interface entities to the component controller.

**Caution:** Deleting the Web Dynpro component interface from the list of implemented interfaces will not delete the interface entities (methods, events, context, and interface views) from the component controller.

**Caution:** Web Dynpro component interfaces and Web Dynpro components use the same name space.

**Caution:** Methods, events, and the context defined in the Web Dynpro interface are not automatically added to the interface controller of the implementing component. For each of these constituents, the *Interface* flag has to be set manually.

---

**Figure 117: Dynamic Component Usage**

---
Exercise 12: Component Reuse

Exercise Objectives
After completing this exercise, you will be able to:
• Expose functionality of a Web Dynpro component to other components
• Declare the usage of one component by another component
• Call methods defined in the interface controller of a used component
• Embed an interface view of a used component in a view container of the consumer component

Business Example
While developing your Web Dynpro component, you find out that parts of the functionality have already been developed in another Web Dynpro component. This comprises the controller source code and also the related layout (for example, reading and displaying details for a given flight customer). Thus, you want to reuse the component's functionality from within your Web Dynpro component.

Template: VCT_HLP_S1
Solution: VCT_COMP_S1 (consumer component)
Solution: VCT_COMP_S2 (used component Æ created from scratch)

Task 1:
Copy your Web Dynpro component ZVCT_HLP1_## or the template VCT_HLP_S1 to Web Dynpro component ZVCT_COMP1_##. This component will act as the consumer component.
1. Copy the template, activate your component, and create a Web Dynpro application.

Task 2:
Create the new Web Dynpro component ZVCT_COMP2_##. This component will be used by the consumer component ZVCT_COMP1_##.
1. Create the Web Dynpro component, having one view (suggested name: CUSTOMER_DATA_VIEW ). Embed this view in the window (suggested name: MAIN_WIND ).

Continued on next page
Task 3:
Implement the Web Dynpro component \texttt{ZVCT\_COMP2\_##}. This component should read the details of a flight customer and display the data on the view. The related coding should be encapsulated in a method that is exposed to the consumer component. The method should have an interface parameter to pass the customer ID from the consumer component to the used component.

1. Edit the component controller of the component. Create a new method (suggested name: \texttt{SHOWCUSTOMER}) that can also be called from other components.
2. Add an importing parameter of type SCUSTOM-ID to the method's interface (suggested parameter name: \texttt{IV\_CUSTOMER\_ID}).
3. In the component controller, define a context node of type SCUSTOM (suggested name: \texttt{CUSTOMER\_DATA}) having the attributes ID, NAME, STREET, POSTBOX, POSTCODE, CITY, COUNTRY, and TELEPHONE. Do not change the default cardinality (\texttt{1..1}).
4. Copy this context node to the context of view \texttt{CUSTOMER\_DATA\_VIEW} and map the view node the component controller node.
5. Display the customer data in the layout of the view.
6. Go back to the method defined in Step 1 of this task. Read the customer data for the customer ID passed to this method via the interface parameter. Use the static method \texttt{CL\_VCT\_FLIGHTMODEL\textasciitilde{}READ\_CUSTOMER( )}. Store the data in the controller context.

Task 4:
Use the component defined in the last task to display details of a customer in the consumer component \texttt{ZVCT\_COMP1\_##}. The customer details should be displayed below the booking table on view \texttt{OUTPUT\_VIEW}. The customer details should always fit to the booking highlighted in the booking table on this view.

1. Edit the component \texttt{ZVCT\_COMP1\_##}. Define a usage of the component \texttt{ZVCT\_COMP2\_##} (suggested name: \texttt{CUSTOMER\_COMP\_USAGE}).
2. Edit the component controller. Create a new method (suggested name: \texttt{CUSTOMER\_READ}). This method will encapsulate the code to instantiate the used component and to call the used component's interface method.
3. Declare the usage of the interface controller of component usage \texttt{CUSTOMER\_COMP\_USAGE}.

\textit{Continued on next page}
4. Go back to the method you created in step 2. Implement the method: Read the customer id from the context. If necessary instantiate the used component. Finally call the interface method of the used component and pass the customer id to this method.

5. Edit the layout of view OUTPUT_VIEW. Between the booking table and the button, add a ViewContainerUIElement. Change the LayoutData property to MatrixHeadData.

6. In order to see the interface view of the used component, you have to embed it in the view container of the view OUTPUT_VIEW.

7. Go back to the view OUTPUT_VIEW. The customer information has to be read if the user navigates form view INPUT_VIEW to view OUTPUT_VIEW, if a new flight is selected, or if a new booking is selected. Thus call the component controller method CUSTOMER_READ for these three situations.

8. If time is remaining, you may translate the texts that are used on the layout of both views.

9. Activate and test your consumer component.
Solution 12: Component Reuse

Task 1:
Copy your Web Dynpro component ZVCT_HLP1_## or the template VCT_HLP_S1 to Web Dynpro component ZVCT_COMP1_##. This component will act as the consumer component.
1. Copy the template, activate your component, and create a Web Dynpro application.
   a) Perform this step as in previous exercises.

Task 2:
Create the new Web Dynpro component ZVCT_COMP2_##. This component will be used by the consumer component ZVCT_COMP1_##.
1. Create the Web Dynpro component, having one view (suggested name: CUSTOMER_DATA_VIEW). Embed this view in the window (suggested name: MAIN_WIND).
   a) Perform this step as in previous exercises.

Task 3:
Implement the Web Dynpro component ZVCT_COMP2_##. This component should read the details of a flight customer and display the data on the view. The related coding should be encapsulated in a method that is exposed to the consumer component. The method should have an interface parameter to pass the customer ID from the consumer component to the used component.

1. Edit the component controller of the component. Create a new method (suggested name: SHOWCUSTOMER) that can also be called from other components.
   a) To expose a method of the component controller to the component interface, select the checkbox in the Interface column.

2. Add an importing parameter of type SCUSTOM-ID to the method's interface (suggested parameter name: IV_CUSTOMER_ID).
   a) Double-click on the method name to open the source code editor. In the parameter list that is displayed above the source code, enter the name of the new parameter (suggested name: IV_CUSTOMER_ID).
   b) Choose Type = Importing and Associated Type = SCUSTOM-ID.

3. In the component controller, define a context node of type SCUSTOM (suggested name: CUSTOMER_DATA) having the attributes ID, NAME, STREET, POSTBOX, POSTCODE, CITY, COUNTRY, and TELEPHONE. Do not change the default cardinality (1..1).
   a) Select the Context tab.
   b) Choose Create Node from the context menu of the root node.
   c) Enter the node's name and the dictionary type SCUSTOM in the relatedfields.
   d) Choose Add Attribute from Structure and select the fields listed above.

4. Copy this context node to the context of view CUSTOMER_DATA_VIEW and map the view node the component controller node.
   a) Perform this step as in previous exercises.
5. Display the customer data in the layout of the view.
   a) Change the Layout of the ROOTUIELEMENTCONTAINER to MatrixLayout.
   b) Use the Web Dynpro Code Wizard to create a form for all attributes of the context node containing the customer information.
   c) For the generated GROUP UI element, change the property LayoutData to MatrixHeadData.
   d) Change the property readOnly for all input fields by selecting the corresponding checkbox.
   e) Define a text for the group's caption.

6. Go back to the method defined in Step 1 of this task. Read the customer data for the customer ID passed to this method via the interface parameter. Use the static method CL_VCT_FLIGHTMODEL=>READ_CUSTOMER(). Store the data in the controller context.
   a) Define a local variable of type SCUSTOM (suggested name: LS_CUSTOMER_DATA).
   b) Use the pattern for calling methods to generate the statement for calling the static method. Fill the variable created above (LS_CUSTOMER_DATA).
   c) Define a reference variable of type IF_WD_CONTEXT_NODE (suggested name: LO_ND_CUSTOMER_DATA).
   d) Get the reference to the context node that will hold the customer data in LO_ND_CUSTOMER_DATA.
   e) Use the method BIND_STRUCTURE() for this reference to pass the customer data to the context node.

Task 4:
Use the component defined in the last task to display details of a customer in the consumer component ZVCT_COMP1_##. The customer details should be displayed below the booking table on view OUTPUT_VIEW. The customer details should always fit to the booking highlighted in the booking table on this view.

1. Edit the component ZVCT_COMP1_##. Define a usage of the component ZVCT_COMP2_## (suggested name: CUSTOMER_COMP_USAGE).
   a) On the Used Components tab, enter ZVCT_COMP2_## in the Component column.
   b) Enter the usage name (suggested name: CUSTOMER_COMP_USAGE) in the Component Usage column.

2. Edit the component controller. Create a new method (suggested name: CUSTOMER_READ). This method will encapsulate the code to instantiate the used component and to call the used component's interface method.
   a) Perform this step as in previous exercises.

3. Declare the usage of the interface controller of component usage CUSTOMER_COMP_USAGE.
   a) On the Properties tab, choose Create Controller Usage.
   b) Select the entry for the interface controller of the component usage CUSTOMER_COMP_USAGE.
4. Go back to the method you created in step 2. Implement the method: Read the customer id from the context. If necessary instantiate the used component. Finally call the interface method of the used component and pass the customer id to this method.

   a) Use the Web Dynpro Code Wizard to read the value of the attribute CUSTOMID from the context. Remember that selecting a booking in the table sets the lead selection in the node BOOKINGTAB.

   b) The code for instantiating the used component and for calling the interface method SHOWCUSTOMER can also be created by means of the Web Dynpro Code Wizard. You only have to pass the parameter containing the customer id to the method.

5. Edit the layout of view OUTPUT_VIEW. Between the booking table and the button, add a ViewContainerUIElement. Change the LayoutData property to MatrixHeadData.

   a) Perform this step as in previous exercises.

6. In order to see the interface view of the used component, you have to embed it in the view container of the view OUTPUT_VIEW.

   a) Edit the window of the consumer component. Select the view container of the view OUTPUT_VIEW.

   b) From the context menu of the view container, choose Embed View.

   c) Choose the interface view of the component usage.

7. Go back to the view OUTPUT_VIEW. The customer information has to be read if the user navigates from view INPUT_VIEW to view OUTPUT_VIEW, if a new flight is selected, or if a new booking is selected. Thus call the component controller method CUSTOMER_READ for these three situations.

   a) Select the tab Actions. Create a new action (suggested name: SHOW_CUSTOMER).

   b) Implement the action handler method. Call the method CUSTOMER_READ of the component controller (use Web Dynpro Code Wizard).

   c) Select the tab Layout. Assign the action SHOW_CUSTOMER to the event onLeadSelect of the flights table and of the booking table, respectively.

   d) Select the tab Inbound Plugs. Double click the plug's name to navigate to the source code of the related method. Call the method CUSTOMER_READ of the component controller (use Web Dynpro Code Wizard) after having read the flights.

Continued on next page
8. If time is remaining, you may translate the texts that are used on the layout of both views.
   a) Perform this step as in previous exercises.

9. Activate and test your consumer component.
   a) Perform this step as in previous exercises.

Result

Model Solution: VCT_COMP_S2, Method SHOWCUSTOMER

METHOD showcustomer.

DATA ls_customer_data TYPE wd_this->element_customer_data.
DATA lo_nd_customer_data TYPE REF TO if_wd_context_node.

cl_VCT_flightmodel=>read_customer(
  EXPORTING
  i_customid = iv_customer_id
  IMPORTING
  e_scustom = ls_customer_data).

lo_nd_customer_data = wd_context->get_child_node(
  wd_this->wdctx_customer_data).
lo_nd_customer_data->bind_structure( ls_customer_data ).

ENDMETHOD.

Model Solution: VCT_COMP_S1, Method CUSTOMER_READ

METHOD onactionshow_customer.

DATA lo_nd_flighttab TYPE REF TO if_wd_context_node.
DATA lo_nd_bookingtab TYPE REF TO if_wd_context_node.
DATA lo_el_bookingtab TYPE REF TO if_wd_context_element.
DATA ls_bookingtab TYPE wd_this->element_bookingtab.
DATA lv_customid LIKE ls_bookingtab-customid.
DATA lo_cmp_usage TYPE REF TO if_wd_component_usage.
DATA lo_interfacecontroller TYPE REF TO iwci_VCT_comp_s2.

Continued on next page
* read value of attribute CUSTOMID from context
lo_nd_flighttab = wd_context->get_child_node( name = wd_this->wdctx_flighttab ).
lo_nd_bookingtab = lo_nd_flighttab->get_child_node( name = wd_this->wdctx_bookingtab ).
lo_el_bookingtab = lo_nd_bookingtab->get_element( ).
lo_el_bookingtab->get_attribute( EXPORTING name = 'CUSTOMID'
IMPORTING value = lv_customid ).

* Instantiate component CUSTOMER_COMP_USAGE if necessary
lo_cmp_usage = wd_this->wd_cpuse_customer_comp_usage( ).
IF lo_cmp_usage->has_active_component( ) IS INITIAL.
lo_cmp_usage->create_component( ).
ENDIF.

* call interface method SHOWCUSTOMER passing customer number
lo_interfacecontroller = wd_this->wd_cpifc_customer_comp_usage( ).
lo_interfacecontroller->showcustomer( iv_customer_id = lv_customid ).

ENDMETHOD.
Model Solution: VCT_COMP_S1, Method
HANDLEFROM_INPUT_VIEW

METHOD handlefrom_input_view .

wd_comp_controller->flighttab_fill( ).
wd_comp_controller->customer_read( ).

ENDMETHOD.
Model Solution: VCT_COMP_S1, Method
ONACTIONSHOWCUSTOMER

METHOD onactionshow_customer .
wd_comp_controller->customer_read( ).
ENDMETHOD.