Leverage component-based architecture in Web Dynpro Java business applications

Part 2 — Component models

by Bertram Ganz and Richard Tucker

Bertram Ganz
Senior Product Specialist,
SAP AG

Richard Tucker
Principal Web
Development Architect,
Atos Origin UK

(Full bios appear on page 80.)

This is the second installation in our three-article series on the componentization of Web Dynpro Java business applications. In the first article, published in the May/June 2008 issue of *SAP Professional Journal*, we looked at the general concepts, principles, and benefits of a clearly component-based application design. The Web Dynpro Java application developed at a UK-based company showcased these aspects in a real-life example without going into technical or implementation- and declaration-specific details.

In this second article, we primarily focus on the beginning aspects of implementing a component-based architecture in Web Dynpro Java using SAP NetWeaver Development Infrastructure (NWDI). We illustrate this architecture by exploring the two independent component models that exist in the Web Dynpro Java application development context. The first and most central component model is the *Web Dynpro component model*, which includes the Web Dynpro component as the main building block in the Web Dynpro Java programming model in which the controller code and a user interface (UI) are implemented. The second component model is defined by the NWDI and includes the *Web Dynpro development component*, which packages, builds, and deploys Web Dynpro components and other Web Dynpro development objects.

We begin with a detailed description of the Web Dynpro component model including its technical parts, such as the component anatomy specified by the outer component interfaces and the inner component implementation, and the integral component usage entity for assembling multiple Web Dynpro components. We then introduce the component interface definition concept, which loosely couples Web Dynpro component implementations at design time and presents the Web Dynpro component diagrams drawn in this article. Based on these concepts, we then describe the most important Web Dynpro componentization

techniques and show how the case study about the Web Dynpro Java application applied them. We cover declaring component usages, invoking component interfaces, firing and catching events across component borders, sharing data between components with interface context mapping, and embedding visual components using component interface views.

Next, we describe the Web Dynpro development component model and the techniques related to NWDI. These techniques are used to separately store Web Dynpro components and other Web Dynpro entities within Web Dynpro development components. Moving beyond the elementary concepts of sharing Web Dynpro entities across several Web Dynpro development components by defining public parts and their related usage dependencies, you learn how to use the same external Java Archive file (JAR file), which you used in the first article, in different Web Dynpro development components and how to deploy it to SAP NetWeaver Application Server (AS) Java.

Finally, we conclude this article with an explanation of what's new in SAP NetWeaver Composition Environment (SAP NetWeaver CE) 7.1: a streamlined Web Dynpro component model and enhanced functions not available in SAP NetWeaver 2004 and 7.0.

In the third article, we'll present a practical description of Web Dynpro component implementation and packaging techniques. We'll explore Web Dynpro componentization patterns in more detail (some of which are covered in this article). We'll also cover some tips and tricks to help you work more efficiently with Web Dynpro development components inside NWDI. We'll explain how to optimize productivity and maintainability by finding the right granularity of Web Dynpro development components and by packaging Web Dynpro models into Web Dynpro development components: This includes putting into practice our naming convention recommendations for Web Dynpro components and packages.

Let's begin with an introduction of the Web Dynpro Java component model.

Web Dynpro Java component model

The development of a component-based Web Dynpro Java application takes place on two different component levels, as illustrated in **Figure 1**.

You can separate a complete Web Dynpro business application into several Web Dynpro components on architectural and implementation-related levels; this yields a clear separation of concerns and the ability to reuse functionality and UI parts (see the left side of **Figure 1**).

On an infrastructure level, you can separate all of the development objects of the whole Web Dynpro business application containing Web Dynpro components or Web Dynpro models into several Web Dynpro development components (see **Figure 1**, right side). This yields the advantages of having separate development, build, and deploy units such as versioning, team development, ease of maintenance, and reuse of centrally implemented functionality. With the Web Dynpro components, models, or other entities added to a public part of the corresponding Web Dynpro development components, they can be easily used inside other development components. (We'll cover the Web Dynpro development component separation topic in the section "SAP NetWeaver development component model" later in the article.)

To successfully apply this component-separation technique in your own Web Dynpro Java business application, you need to clearly understand the basic principles and the technical structure, or anatomy, of a Web Dynpro component. You also have to understand its outer interface parts, its inner implementation, and the special Web Dynpro component usage concept on which every relation between a Web Dynpro child component and its parent component is based. The Web Dynpro Java component model and the most important componentization techniques presented in this section are illustrated with concrete examples taken from the Web Dynpro Java applications that the case study uses.

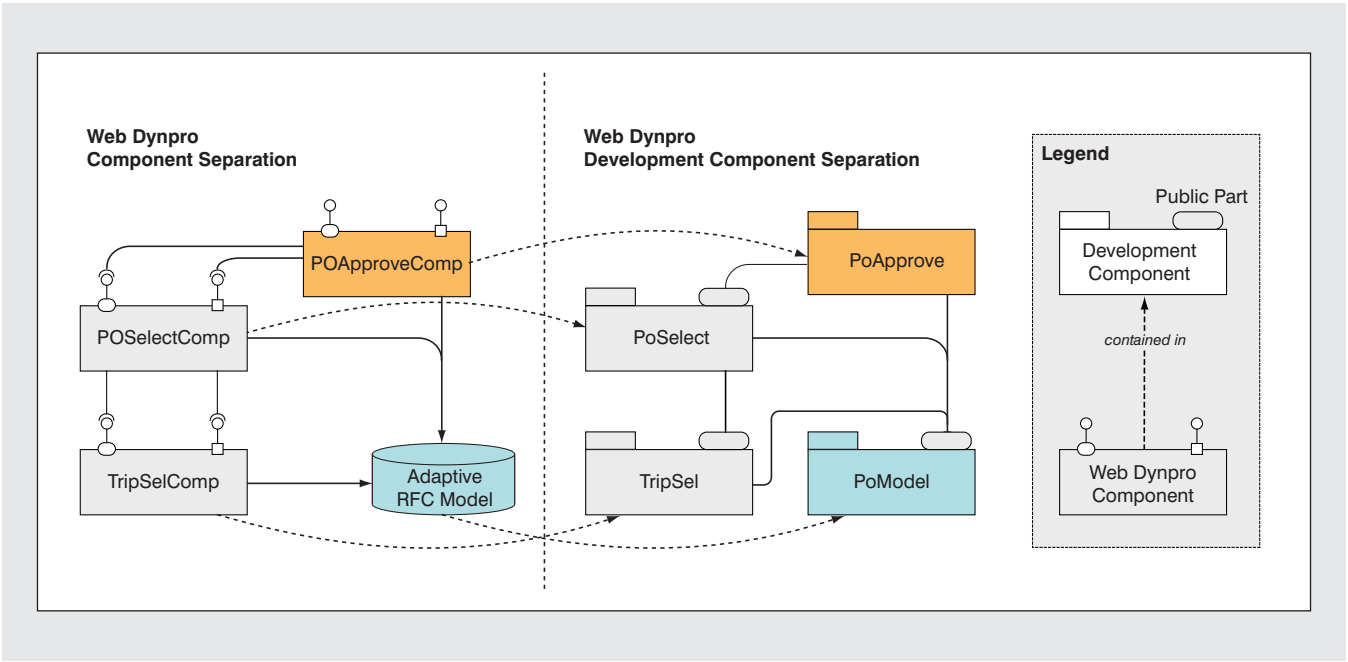


Figure 1 Two types of component separation — Web Dynpro components and development components

Before looking at the technical details of the Web Dynpro component anatomy, first we'll discuss the basic principles.

Web Dynpro component principles

Web Dynpro defines five independent development entities: application, component, component interface definition, model, and dictionary. Of these, the Web Dynpro component is considered as the most important, integral building block for developing Web Dynpro business applications. Web Dynpro components are the only places into which you can put Web Dynpro functionality. For example, to design a UI to handle events that an user triggers or to retrieve back-end data from a Web Dynpro model, you need to implement a Web Dynpro component.

Users cannot directly access the components themselves. If your users need access to a specific Web Dynpro component, you must create a *Web Dynpro application* that refers either to this Web Dynpro component directly or to another component that contains it (see the sidebar on the next page for definitions of some additional componentization-related terms). To use a Web Dynpro component, you must create at least one Web Dynpro application entity as an entry point for the user and one Web Dynpro component as a functionality provider. At a minimum, this single Web Dynpro component implements the UI and the application logic defined in the comprised controller classes. In a simple "Hello World" Web Dynpro component, the UI consists of a Web Dynpro view layout with a simple input form. The application logic implemented in the related view controller class handles the user input and returns some message text to the UI. Although you can build a simple Web Dynpro application with only one component, an actual Web Dynpro application is typically based on multiple components that connect to each other (see **Figure 2** on the next page).

Web Dynpro components are the means by which Web Dynpro provides reuse on the dialog control level, whereas Web Dynpro models are for reuse on the back-end integration level (see the terminology sidebar on the next page). You can only reuse a component as a whole; you cannot reuse contained parts of the component, such as views or controllers,

New Web Dynpro terminology

In addition to the terms defined in the article, here are others that you will encounter:

- **Web Dynpro application:** This special entity in the Web Dynpro programming model defines which visual interface of a Web Dynpro component the application's startup phase addresses. You can start the application via a URL in a Web Dynpro client (e.g., in the Web Dynpro HTML client). Technically, a Web Dynpro application points to a startup navigation plug of a component interface view that the Web Dynpro root component exposes. The *Web Dynpro root component* is the root of a Web Dynpro component hierarchy.
- **Web Dynpro model:** This technical entity in the Web Dynpro programming model includes an aggregation of model classes needed to connect a Web Dynpro application to the back end. Importing an Adaptive Remote Function Call (RFC) model (based on an RFC module) or an Adaptive Web service model (based on a Web service definition) automatically generates these model classes. The application developer must manually implement a JavaBean model to connect to Enterprise JavaBeans (EJB) in SAP NetWeaver 2004 and 7.0. However, SAP NetWeaver CE 7.1 automatically generates the new Web Dynpro EJB 3.0 model.
- **Separation of concerns (SoC):** This process breaks down a computer program into distinct features that overlap in functionality as little as possible. Any piece of interest or focus in a program is a concern. Typically, concerns are synonymous with features or behaviors. Traditionally, modularity and encapsulation have been used to progress toward SoC with the help of information-hiding (see Wikipedia at http://en.wikipedia.org/). In Web Dynpro, modularity and encasulation are achieved with Web Dynpro components and information-hiding with their visual and programmatic interfaces.

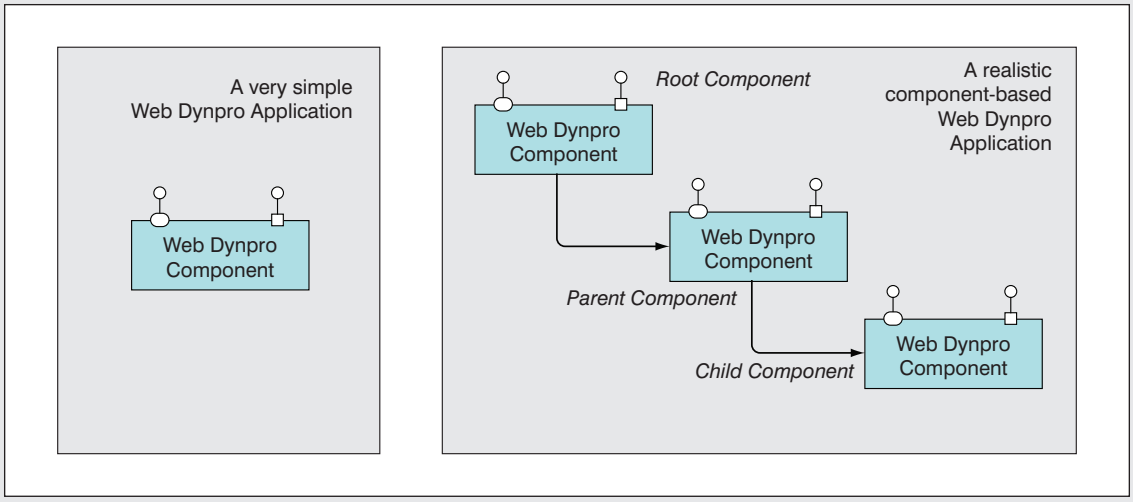


Figure 2 Every Web Dynpro application has at least one component

- The name **component interface view** is motivated by the fact that you can use the visual interface of a Web Dynpro component like a normal view (defined inside a component) when modeling the view hierarchy and the navigation structure inside a window. As with normal views, you can embed component interface views into windows, view sets, and ViewContainer UI elements.
- Interface **context mapping** in general means the Web Dynpro context mapping of a controller context in the parent component to the context of the child component's interface controller, or vice versa, the context mapping of the child component's interface controller context to a controller context in the embedding component (named external interface context mapping).
- The **Model-View-Controller** is a well-known architectural pattern in computer software development first described by Trygve Reenskaug in his article: "Thing-Model-View-Editor—An Example from a Planning System." Technical note, Xerox PARC, (May 1979). Meanwhile, this pattern is the de facto architectural standard adopted by many existing Web programming frameworks in different flavors. One flavor is the Web Dynpro Java programming model based on components, views, controllers, and models.
- The **data context** is storing the original context data. It is not mapped to another context.
- **Java dictionary types** are simple types and structure types defined independently either in a Local Dictionary or in combination with an Adaptive RFC Model where all ABAP Dictionary types, used by the underlying Remote Function Modules (RFMs), are mirrored in a corresponding Logical Dictionary used for declaration purposes at design time. The Logical Dictionary is a design-time snapshot of the ABAP Dictionary types created during an Adaptive RFC Model import. At runtime the dictionary type metadata is retrieved from the ABAP Dictionary of the connected SAP system.

outside the component. Access to parts of a component, apart from its interfaces, isn't possible, and embedded Web Dynpro components cannot make any assumptions about their embedder. This is called the *black-box principle* (see the sidebar on the next page for more information).

The functionality of componentization allows you to divide a large-scale business application into smaller, more manageable parts, so that different concerns are separated into logical units, also known as the separation of concerns (see the sidebar on page 52). For example, you can divide a Web Dynpro application into faceless components implementing back-end access, several visual components executing different parts of the application UI (such as a search form component or a data display component), and one root component bundling together all other components.

Some of the essential features of the Web Dynpro component model, such as its interface concept, the ability to abstractly define component interfaces, the modularization of large applications, and the reusability of Web Dynpro components make componentization possible. The Web Dynpro application developed at the UK-based facilities management company (described in the first article) applied Web Dynpro component reusability. If you recall, we used Web Dynpro components to provide central functionality, such as selecting purchase orders (PoSelect component) or selecting sites (TripSel component) to other higher-level components such as changing purchase orders (PoChange component) and approving purchase orders (PoApproval component). Once you implement functionalities like these, you can reuse them in other applications. Web Dynpro components concentrate reuse in one building block, because you cannot reuse the entities inside a

Web Dynpro components adhere to the black-box principle

Web Dynpro components follow the *black-box principle,* which is characterized by the following criteria:

- Access to inner parts of components, such as views or controllers, apart from their interfaces is *not* possible.
- Access to a component is only possible by using its visual and programmatic interfaces. *Component interface views* are the visual interfaces; the *component interface controller* is the programmatic interface for a Web Dynpro component.
- Embedded Web Dynpro components cannot make any assumptions about the components in which they are embedded. This means a Web Dynpro child component does not know its parent component.

component (such as views or controller classes) as standalones. This approach yields a more efficient development process by minimizing redundancies on the implementation and declaration levels.

Anatomy of a Web Dynpro component

In this section we focus on the Web Dynpro component level and examine the fundamental concepts of a component-oriented Web Dynpro programming approach: the Web Dynpro component anatomy with the outer component interfaces and the inner parts of a component implementing these interfaces, such as views and controllers.

First, we look at the outer component interfaces of the Web Dynpro component anatomy, in which a Web Dynpro component is seen as a "black box" without knowing the inner implementation details, such as:

- What are the visual and programmatic interfaces of a Web Dynpro component that you can apply in a Web Dynpro parent component?
- How can you visually embed another Web Dynpro component into the UI of an embedding component?
- How can you programmatically access a Web Dynpro component for the purpose of calling controller methods, transferring data, or subscribing to controller events?

A component user who wants to apply or reuse the functionality of another component by using its interfaces without knowing any implementation details needs the external component interfaces of a Web Dynpro component.

So, let's dive into the inner parts of the Web Dynpro component anatomy and take a closer look at the implementation details, such as:

- How are the visual interfaces (i.e., component interface views) implemented inside a Web Dynpro component and how do they expose its UI and navigation plugs to other components?
- How are the programmatic interfaces put into practice inside a Web Dynpro component, and how do they expose controller methods to fire events across component borders or to share data between components with interface context mapping?
- What are the visual and programmatic entities inside a Web Dynpro component implementation (windows, view layouts, or different controllers), and how do they relate to component interfaces?

Interfaces of a Web Dynpro component

Web Dynpro components have specific interfaces for fulfilling visual, programmatic, and data exchange-specific needs. A parent component uses these interfaces to visually embed a child component, to

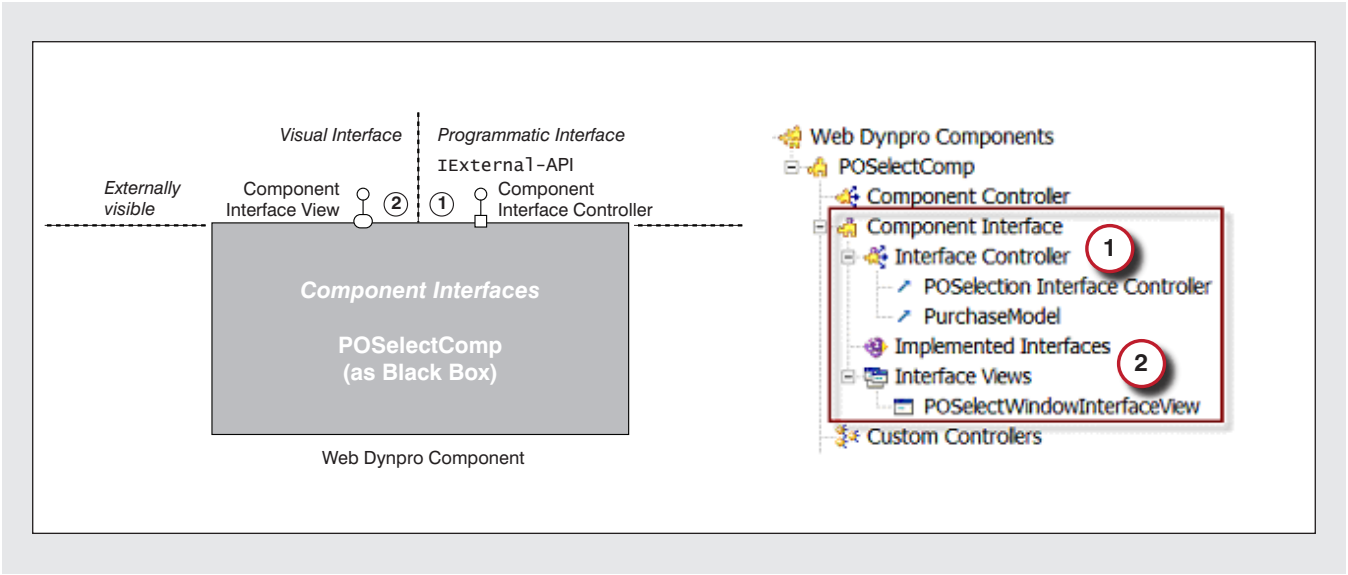


Figure 3 The visual and programmatic interfaces of POSelectComp

retrieve some structured data from the child component, to subscribe to the child component's events, or to delegate some logic to the child component by invoking an interface method. Based on the black-box principle, the parent component exclusively connects to a child component's interfaces, without knowing any inner implementation details inside the child component, such as the controller implementation or the internal structure of the UI.

Figure 3 illustrates the visual and programmatic interfaces of the POSelect Web Dynpro component taken from the case study application. To adhere to the Web Dynpro component naming conventions described in our next article, we added the suffix *Comp* to the component name *POSelectComp*. On the left side of the figure, you can see the Web Dynpro component diagram with two component interface port symbols representing the visual (②) and programmatic (①) component interfaces (for more information on the diagrams used in this article series, see the sidebar on page 52). On the right side of the figure, you can see the corresponding nodes shown in Web Dynpro Explorer. The developer of the component POSelectComp selects these nodes to declare its interface controller (①) by adding context elements, methods, or events and its component interface view POSelectWindowInterfaceView (②) by adding navigation plugs.

A Web Dynpro component provides two different types of interfaces to an external Web Dynpro entity, which can be another Web Dynpro component or a *Web Dynpro application:*

- **Component interface view:** You can use a visual interface of a Web Dynpro component, called a component interface view, to build modularized UIs: You split the application UI into separate Web Dynpro UI components, each implementing a specific part of the application UI (see the terminology sidebar on page 52). The purpose of a component interface view is to expose the application UI implemented inside a visual Web Dynpro component via the UI for external use. In **Figure 4** (on the next page) you see the component diagram for the Purchase Order Approve application (POApproveApp), described in the first article. The root or application component (with technical name POApproveComp) is used by the Web Dynpro application entity POApproveApp (①), which the application's user can address via URL. The application component POApproveComp (②) uses the UI component POSelectComp (③), which uses the search

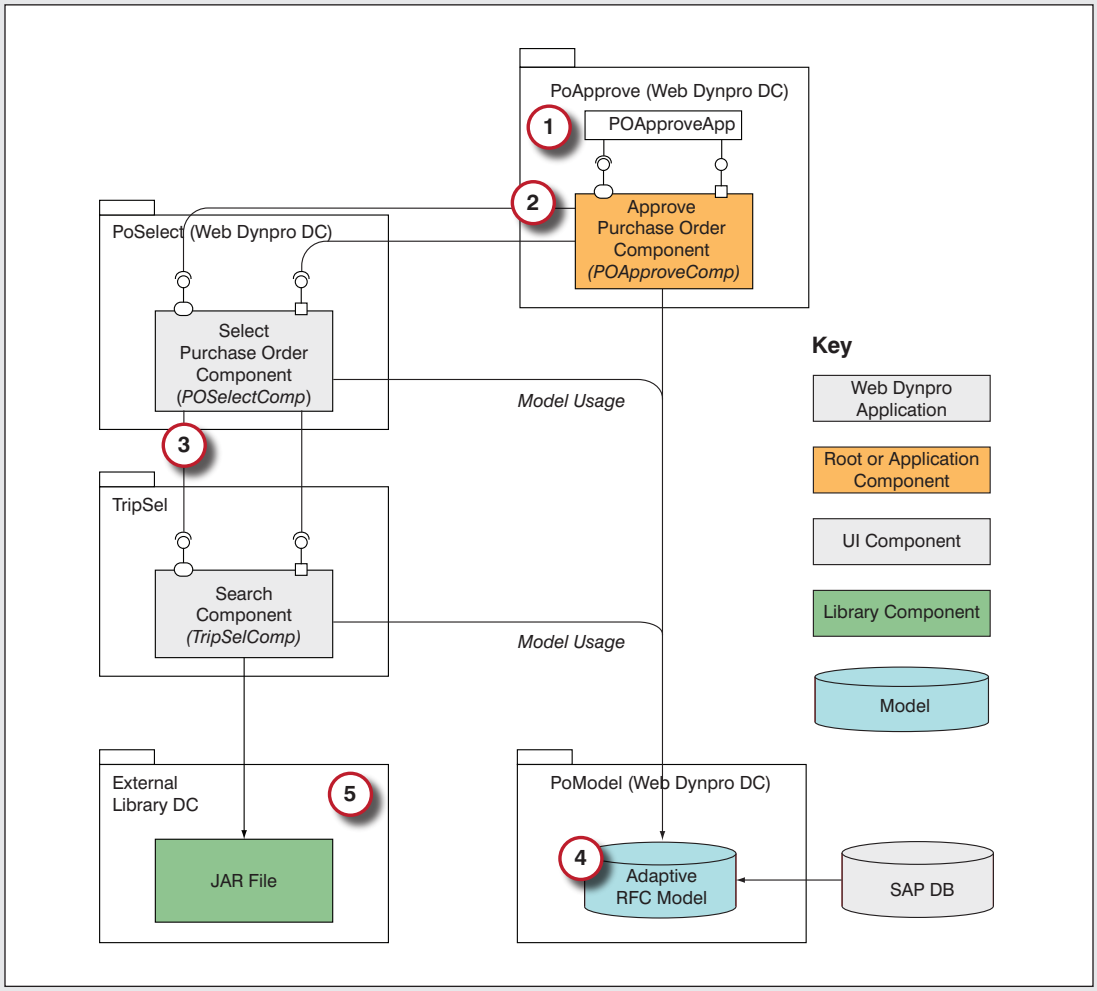


Figure 4 Web Dynpro component architecture for Purchase Order Approve application

component TripSelComp. Remember that the component POSelectComp is reused by additional application components: the purchase order change component (POChangeComp) and the purchase order goods receipt component (POGoodReceiptsComp).

A Web Dynpro parent component uses the component interface view of its visual child component by embedding it into the parent component's own UI implementation. The parent component can use the component interface views of the visual child component, such as its Web Dynpro views. You can accomplish this by embedding component interface views into the special ViewContainer UI element (defined in a Web Dynpro view layout) or into a view area (defined in a view set of a Web Dynpro window). Look ahead to **Figure 6** (page 58), which illustrates the embedding of component interface views. You also can define the navigation plugs for interface views to start, exit, suspend, or resume a Web Dynpro application or to navigate between Web Dynpro views and other component interface views. The inner visual

parts in Web Dynpro components, including Web Dynpro windows, Web Dynpro views, and UI elements, are described in more detail later in this article.

A Web Dynpro component can expose one or multiple component interface views. A Web Dynpro component that does not expose a component interface view is called a *faceless component*.

- **Component interface controller:** You can use a programmatic, therefore non-visual, interface of a Web Dynpro component to implement modularized controller logic and to transfer structured data across component borders. You specify a component interface controller by its methods, events, and context as the Web Dynpro-specific hierarchically structured storage location, which all Web Dynpro controllers can define within a controller. The component interface controller enables an external Web Dynpro component to interact with the embedded Web Dynpro component via method calls, the reaction to events, or the exchange of context data using interface context-mapping (see the terminology sidebar on page 52). The component interface controller exposes an automatically generated Java interface with the name IExternal<*component name*>, which controllers of the embedded component invoke. This Java interface reflects all of the methods and events that you have declared in the component interface controller (see **Figure 5**), and it provides access to the generic controller IWDController[1] API, common to all Web Dynpro controllers. The exception is that it's not possible to access the inner parts of a Web Dynpro component (black-box principle) via the component interface controller.

To illustrate the practical use of the two visual and programmatic Web Dynpro component interface types, let's take a closer look at the case study Web Dynpro application developed for the UK-based facilities management company.

[1] For more information go to SDN at https://help.sap.com/javadocs/NW04S/current/wd/index.html.

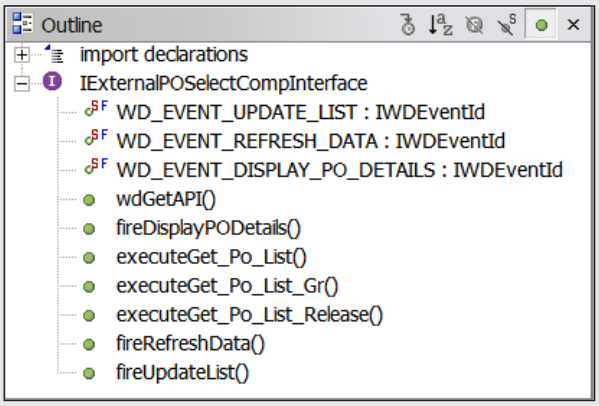


Figure 5 Generated IExternal API of the POSelectComp component's interface controller

Figure 6 on the next page is an architectural representation of the completed PoApprove application UI.

- **Embedding component interface views:** Based on a nested Web Dynpro component architecture, the root component POApproveComp uses the visual component interface of its child component POSelectComp to embed the POSelectComp implemented UI into its own UI. This is done by embedding the component interface view with the name POSelectWindowInterfaceView into cell[1,1] of the previously defined view set selectionViewSet. Two other cells are filled with the views ApproveView and MessageView, which are part of the parent component. In the same way, the component POSelectComp embeds the component TripSelCompInterfaceView of the child component TripSelComp into cell[1,1] of its own view set. In the Web Dynpro component diagram, the embedding of the component interface views is visualized with a connector between the parent component and the visual port symbol (the component interface view) of the used child component.

- **Using component interface controllers:** Component POApproveComp must be able to

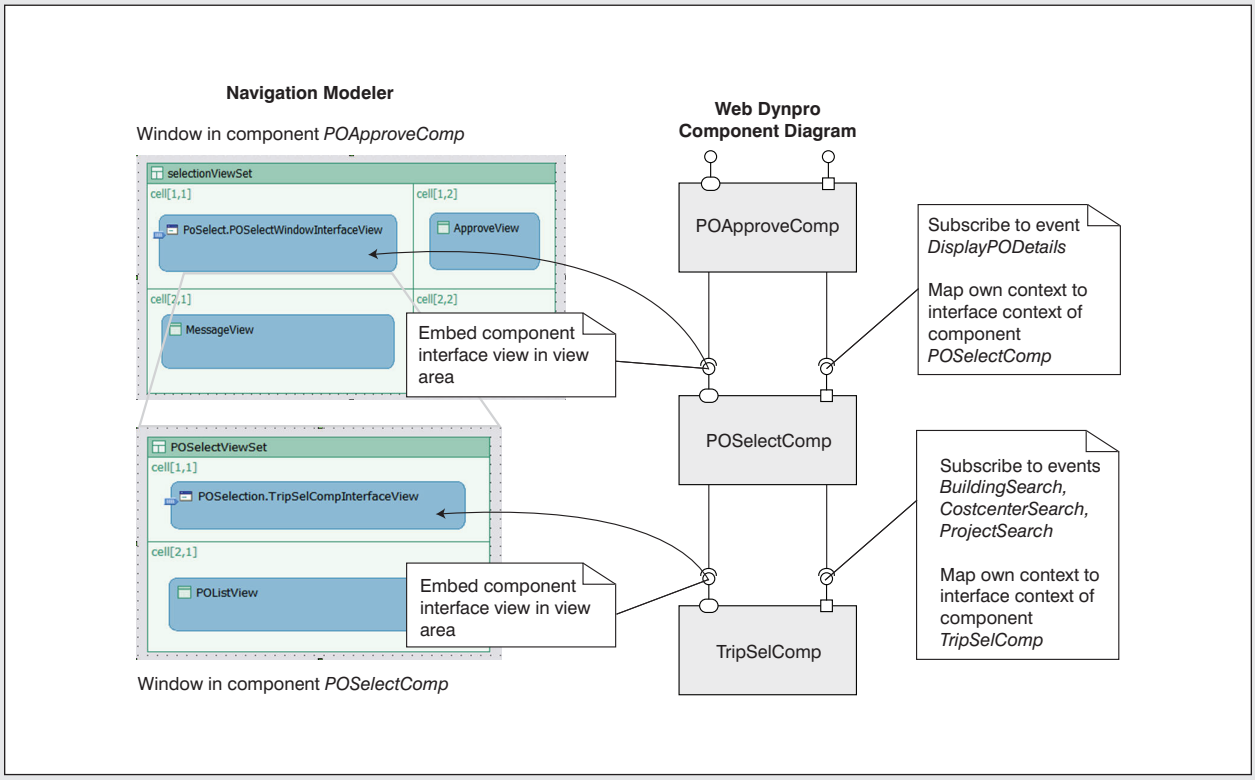


Figure 6 Using Web Dynpro component interfaces in the Purchase Order Approval application

react on events fired in its child component POSelectComp, which itself must react on events fired in the component TripSelComp on the deepest level of the component hierarchy. When the user selects a purchase order from the table in child component POSelectComp, the parent component POApproveComp must validate the purchase order details and then display them on the screen in its own view. To make this happen, the POApproveComp component controller subscribes an event handler to the event DisplayPODetails exposed by interface controller of component POSelectComp.

In addition, the component POApproveComp must be able to access the data stored in its embedded component. This is done by *interface context mapping* — mapping its own controller context to the context defined in the interface controller of the used POSelectComp component. With this kind of mapping, it is possible to reference structured data stored in a used component, such as the purchase order selected in component POSelectComp to be approved in component POApproveComp.

Web Dynpro component interfaces vs. Java interfaces

For a Java developer, an interface is directly associated with a programmatic Java interface. In Java, an interface is a type, just as a class is a type. Like a class, an interface defines methods, but never implements them. A Java interface is an abstraction of a class independent from its implementation.

Web Dynpro component diagrams

The Web Dynpro component diagrams used in this article series are based on the general component diagrams that were introduced in Unified Modeling Language (UML) 2.0 (see http://www.uml.org/ for more information). We made small adaptations to the standard presentation to tailor the diagrams to the special architecture of Web Dynpro components and, therefore, simplify the interpretation of the diagrams.

Ports for presenting Web Dynpro component interfaces

A Web Dynpro component has special entry points called ports in UML 2.0, which connect the outer environment to the inside. In the outer environment, other Web Dynpro components and Web Dynpro applications represent the possible entities using a Web Dynpro component. You can regard a port as the specification of an interaction point or as an interface on the Web Dynpro component's shell. As shown in the top illustration on the next page, a Web Dynpro component has exactly two port types:

- **User interface port equals component interface view:** Component interface views are the visual interfaces of a Web Dynpro component for the modular build of Web Dynpro user interfaces. They are also referred to as user interface ports. A Web Dynpro component can expose several component interface views to the outside, where every window inside a component has exactly one corresponding component interface view. It is also possible not to define a component interface view, which is the case with non-visual components (i.e., faceless Web Dynpro components, in contrast to the visual Web Dynpro UI components). In addition, component interface views have diverse navigation plugs (and their parameters) for defining application startup and exit navigation (startup and exit plugs), navigation transitions from and to component interface views (inbound and outbound plugs, respectively) inside a running Web Dynpro application, and navigation transitions to and from external applications while a Web Dynpro application keeps running (suspend and resume plugs, respectively).
- **Controller port equals component interface controller:** At the controller level, the component interface controller of a Web Dynpro component represents the second port type. Every Web Dynpro component has exactly one component interface controller that exposes the context, methods, and events to the outer using components (other Web Dynpro components).

To distinguish between user interface ports and controller ports, user interface ports (component interface views) are identified with rounded corners, and controller ports (component interface controller) are represented by squares as defined in UML 2.0.

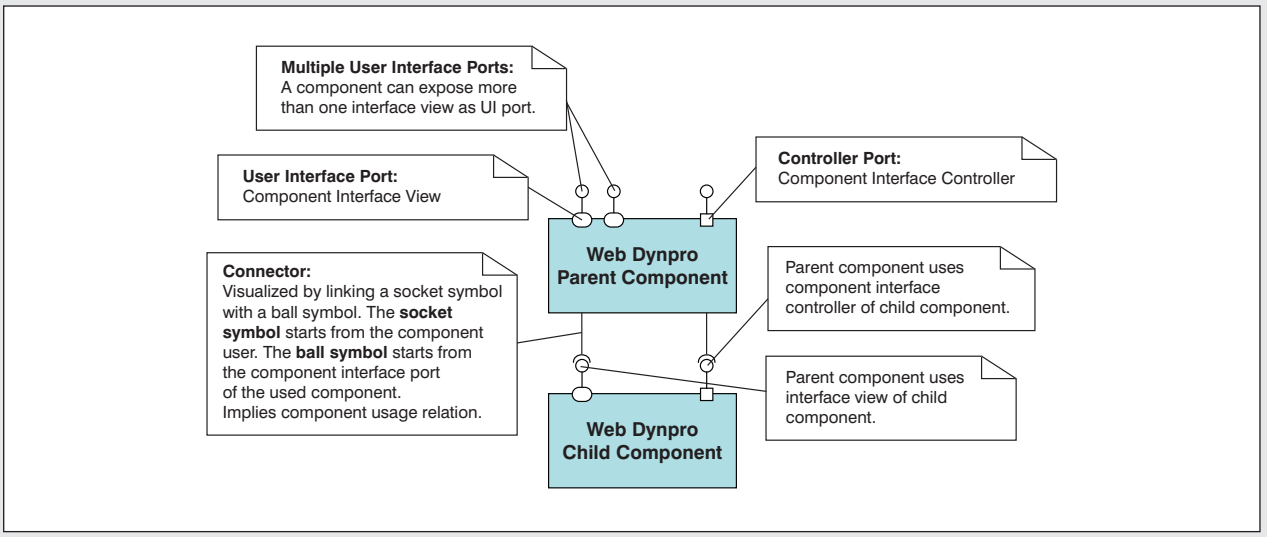
Connectors and usage dependencies between Web Dynpro components

A connector represents the connection of a Web Dynpro component to the port of another component. The connector is represented by linking a socket icon to a ball icon. The socket icon starts at the component that is using the other component while the ball icon starts at the component that is being used.

The diagram also illustrates the usage dependency of a parent component with its child component. The parent component embeds the component interface view of the child component in its own user interface; this is represented by a connector between the parent component and the user interface port of the child component. The parent component uses or requires the component interface controller of the child

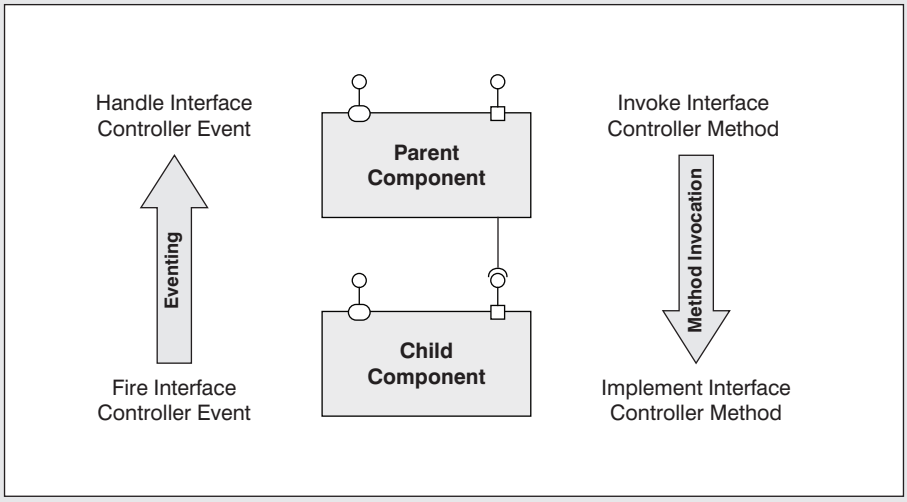
component, which is represented by a second connector between the parent component and the controller port of the child component.

Because a connector between two Web Dynpro components implies usage dependency, you can omit the additional arrow from the parent to the child component.



Component interface controller eventing vs. method invocation

There are two ways of communicating between a parent component and its embedded child component. Component interface controller eventing is applied by a child component to communicate with its parent component. Component interface controller method invocation is applied by a parent component to communicate with its embedded child component (as depicted in the illustration below).



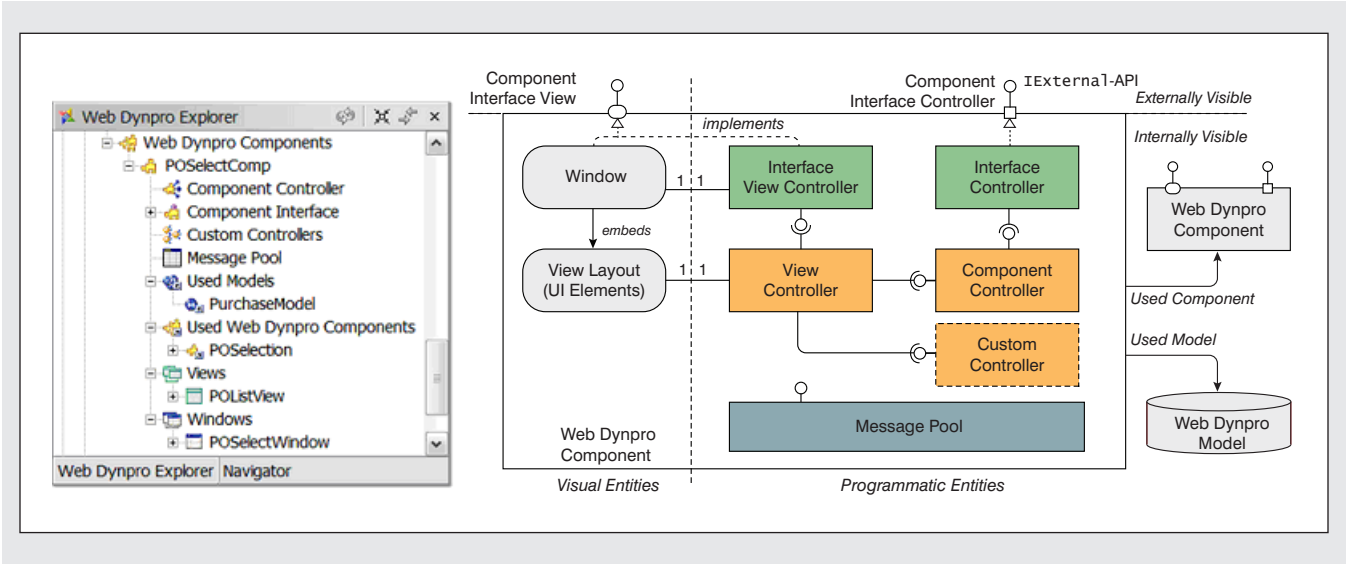


Figure 7 The inner anatomy of Web Dynpro component POSelectComp displayed in Web Dynpro Explorer

It's important to understand how the Web Dynpro component model applies the interface concept. Principally, you need to see Web Dynpro component interfaces as *meta-model interfaces*, not as conventional Java interfaces. This means that the two Web Dynpro component interface flavors *interface view* and *interface controller* primarily expose *meta-model information* on the functionality provided by a Web Dynpro component implementation, to be used by the embedding component:

- **Component interface view:** The meta-model information is the existence of the defined component interface views, the navigation plugs exposed by interface views, and the types of these plugs (startup, exit, inbound, outbound, suspend, resume). With this meta-model information, a parent component can then embed the component interface view into its own UI and navigation schema.
- **Component interface controller:** The meta-model information is the interface context structure needed to declare context mapping relationships between a parent component and its used component and the events to which a controller inside the parent component can subscribe. For this event-subscription relationship to exist, the Web Dynpro tools (the Web Dynpro design-time environment inside SAP NetWeaver Developer Studio) need the meta-model information that describes the component interface controller of the used component.

The generated, typed IExternal<*component name*> API of a Web Dynpro component interface controller is the only Java interface related to the Web Dynpro component interface concept. The Web Dynpro tools automatically generate this Java interface, which reflects the events and public methods declared in a component interface controller for programmatic access within your controller code (**Figure 5**). (You'll learn more about the IExternal<*component name*> API of a Web Dynpro component interface controller in the third article.)

Next, let's take a closer look at the internal anatomy of the Web Dynpro component and its implementation

Internal implementation of Web Dynpro component interfaces

To implement a Web Dynpro component and its exposed component interfaces, you need to understand the inner parts of the component and how they relate to each other. **Figure 7** shows the anatomy of Web

MVC model	Model	View	Controller
External interface	Component interface controller (context)	Component interface view (navigation plugs)	Component interface controller (meta-model information on declared methods and events) with its IExternal<*component name*> API
Internal interface implementation	Component interface controller (context)	Window	Component interface controller (class implementing declared methods and events); component interface view controller (class implementing event handlers for declared navigation plugs)
Internal implementation	Component, custom, view controller (context)	View (UI elements in view layout)	Component, custom, and view controllers (methods, events), message pool
Used entities	Web Dynpro model, Web Dynpro component (interface controller: context)	Web Dynpro component (interface view)	Web Dynpro component (interface controller: declared methods, events)

Figure 8 MVC classification of the Web Dynpro component anatomy

Dynpro component POSelectComp displayed in Web Dynpro Explorer.

A Web Dynpro component mainly consists of different programmatic controller entities that implement the application and navigation logic (such as interface, component, custom, and view controllers) and visual entities that implement the UI part (such as interface views, windows, view layouts, and UI elements). In addition, the inner part of a Web Dynpro component implements the visual and programmatic interfaces exposed to other outside components.

The architecture of a Web Dynpro component is based on the Model-View-Controller (MVC) model (see the terminology sidebar on page 52), which establishes a clear separation between the business data model, the user interface, and the controller implementation or application logic. This MVC-based approach helps to differentiate among the following component-related entities:

- **External interfaces:** Visual and programmatic component interfaces that a Web Dynpro component exposes
- **Internal interface implementation:** Inner implementation of the exposed Web Dynpro component interfaces
- **Internal implementation:** Additional inner entities of a Web Dynpro component
- **Used entities:** Outer entities that a Web Dynpro component can use

Figure 8 defines the component-specific entities in relation to the three underlying parts of the MVC model.

From a multi-component-based development perspective, it's important to understand the internal implementation of the two interfaces of a Web Dynpro component:

- **Implementation of the component interface controller:** The component interface controller class (or simply the interface controller class) implements the component interface controller that a Web Dynpro component exposes. The interface controller class implements all of the methods exposed in the IExternal<*component name*> API and allows you to fire the declared events. The context of the interface controller class transfers hierarchically structured data like the data shown in a master-detail table or UI state information such as "is save button visible," "is tree node expanded," or "number of visible table rows," Java objects, and model data across component

borders. The interface controller context can expose context data stored inside a Web Dynpro component for external components, or vice versa, so that the data context resides outside the component. In the latter case, you must externally map the interface controller context to the data context. Because the component interface controller no longer has an associated component interface controller class in future SAP NetWeaver releases, it's advisable to delegate all implementation parts in the component interface controller class to appropriate methods in the component controller class (see the terminology sidebar on page 52).

- **Implementation of component interface views:** Inside a Web Dynpro component, the component interface views are implemented in two parts. First, the interfaces for the user interface of a component that are exposed to the outside are implemented within the component via their windows. A *window* contains the definition of a *view composition* that is defined by the connection of views or component interface views of embedded components to navigation links. To the outside — that is, for visually embedding a Web Dynpro component in the view composition of another component — windows are represented by appropriate component interface views that themselves have inbound and outbound plugs. The second part of the implementation takes place in the component interface view controllers that carry out the event handling of navigation plugs (inbound, startup, resume) to enter a component interface view and the triggering of other navigation plugs (outbound, suspend, exit) to leave a component interface view.

The remaining inside parts of a Web Dynpro component are not directly associated with the externally visible component interfaces:

- **Component controller:** This part acts as the main controller class inside a Web Dynpro component centrally implementing component-relevant controller logic and centrally storing context data that other controllers can reference via context mapping. You should implement the component interface controller inside the component controller (by applying method delegation and context mapping).
- **Custom controller:** This optional controller class inside a Web Dynpro component centrally implements a specific component-relevant part of the controller logic following the separation of concerns design principle. You can map or refer to its context by other controller contexts inside the same component.
- **View layout:** The Web Dynpro view layout provides a visual entity for defining a specific user interface part by arranging Web Dynpro UI elements. You can view a layout in either a window or another view layout using the ViewContainer UI element. Web Dynpro windows provide the possibility of merging several views with an entire user interface. By connecting inbound and outbound plugs at the view level, you define the navigation schema in a window.
- **View controller:** Every view layout is associated with a corresponding view controller. The view controller implements view-specific controller logic such as UI element state management, user action event-handling, dynamic view modification, or navigation logic. It delegates all non-view-specific controller logic to the component controller or an optional custom controller. Data stored in the view controller context is automatically transferred to and from the UI (view layout) based on a context-binding relationship between UI element properties and context elements.
- **Message pool:** You use the message pool to define texts and messages of different types, such as errors, warnings, or success messages, which all controllers in a Web Dynpro component can access. You can define these messages and texts with placeholders to be filled at runtime.
- **Model usages:** You use models to define the integration of a back-end layer into the Web Dynpro component (e.g., an Adaptive RFC model or an Adaptive Web service model). You need to use a model to bind a component or custom controller context in a Web Dynpro component to an imported Web Dynpro model. Based on this

context-to-model binding relationship, model objects are automatically transferred between a Web Dynpro component and a Web Dynpro model at runtime.

Component interface definitions for loosely coupled Web Dynpro components

Another important aspect of the Web Dynpro component interface concept is the two flavors of component interface abstraction. The term *component interface abstraction* means separating an abstract component interface from its concrete component implementation. The Web Dynpro component model has two different abstraction levels for its component interfaces:

- **Web Dynpro component interface:** The component implementation itself defines the component interface (interface view, interface controller). A component interface is implemented in the component that defines it. The interface view controller, its associated window (the defined component interface view), and the component interface controller accomplish the implementation together.
- **Standalone Web Dynpro component interface definition:** You can define the component interface as *standalone*, or *outside* the concrete Web Dynpro component implementation. This special Web Dynpro entity can loosely couple Web Dynpro components without using concrete Web Dynpro components to implement the standalone component interface (*strategy pattern*). The UK case study application did not apply the Web Dynpro component interface definitions. (We'll provide more details in the next article.)

Defining Web Dynpro component usage dependencies

After looking more closely at a Web Dynpro component with its exposed interfaces, we now come to the most important Web Dynpro entity with respect to component-based application architecture: the *Web Dynpro component usage*. Understanding this Web Dynpro-specific entity is essential for answering the questions presented in **Figure 9**.

Understanding the Web Dynpro component usage entity — the key to success

Whenever you want to use another child Web Dynpro component inside an embedding parent Web Dynpro component for different purposes, such as method invocation, event subscription, context-mapping for data transfer, or UI embedding, you must explicitly declare a corresponding Web Dynpro component usage entity inside the using component at design-time (Web Dynpro tools design time environment). Note that in SAP NetWeaver 7.0 a component usage can only be created at runtime using controller code, by copying a component usage defined at design time. However, in SAP NetWeaver CE 7.1, the Web Dynpro Runtime API IWDComponent will expose methods to dynamically create *typed* and *untyped* component usage instances at runtime "from scratch" without copying a component usage defined at design time.

In the Web Dynpro programming model, the component usage is a variable with which the embedding component defines the usage of another component. At design time, you can use a component usage entity as a placeholder for a used component; at runtime, you need to associate the component usage entity with a corresponding component instance. If you want to use a component more than once (e.g., to display the same component twice on the UI), you need to define two separate component usages. At runtime, the embedded component would then have two component instances of the same type.

Every component usage dependency declared in the using component is displayed in Web Dynpro Explorer under the node Web Dynpro Components – <*component name*> – Used Web Dynpro Components. In the Purchase Order Approval application presented in our first article, we defined two component usages to declare usage dependency relationships between the three separate Web Dynpro components: POApproveComp for the purchase order approval component, POSelectComp for the purchase order

Question	Sample answer
How can you implement Web Dynpro component call functions in another component?	The Web Dynpro component POApproveComp invokes a method of its child component interface controller to refresh the displayed business data based on pending user input.
How can a used Web Dynpro child component interact with its embedding parent component?	The Web Dynpro component TripSelComp fires an interface event so that its parent component can react to it after defining a corresponding event subscription relationship.
How can a Web Dynpro component consume context data stored in another component?	The Web Dynpro component POApproveComp maps its context to the interface context of the POSelectComp purchase order selection component exposing the data of all of the purchase orders to be approved.
How can a Web Dynpro component visually embed the UI implemented in another component?	The Web Dynpro component POSelectComp embeds a component interface view of the TripSelComp search component on top of the displayed search results.

Figure 9 Understanding this Web Dynpro-specific entity

selection component, and TripSelComp for searching for cost centers, buildings, or projects:

- **Component usage with name *PoSelect*:** Defines component usage for the UI, functions, and data that the component POSelectComp provides in the component POApproveComp
- **Component usage with name *Search*:** Defines component usage for the search functionality that the component TripSelComp implements in the component POSelectComp

The first component usage dependency between the two Web Dynpro components, POApproveComp and POSelectComp, is illustrated in **Figure 10** on the next page.

On the right side of the diagram is a screenshot of the Web Dynpro Explorer at design time. Node PoSelect (①) represents the component usage entity with which component POApproveComp defines its usage of Web Dynpro component POSelectComp. We recommend naming a component usage with the name of the used Web Dynpro component omitting the suffix *Comp*. Based on this declared Web Dynpro component usage (PoSelect), the parent component POApproveComp can now "use" functions that its child component POSelectComp exposes and implements:

- **Component interface controller method invocation:** To refresh the purchase order data after approval, the view controller class ApproveView.java for the ApproveView (②) in

Note!

The programming model of Web Dynpro is designed in a way that relieves the application programmer or developer of instance management as much as possible. This model allows only single instances of types, such as only one view instance for one view type. However, Web Dynpro components are intended to reach two aims: reuse and multiple instances. Because Web Dynpro has a declarative programming model, you must introduce high-level variables, also called *component usage*, to handle multiple instances of components. During runtime, the component usage provides access to the component life-cycle management so that a component usage instance acts as a component factory.

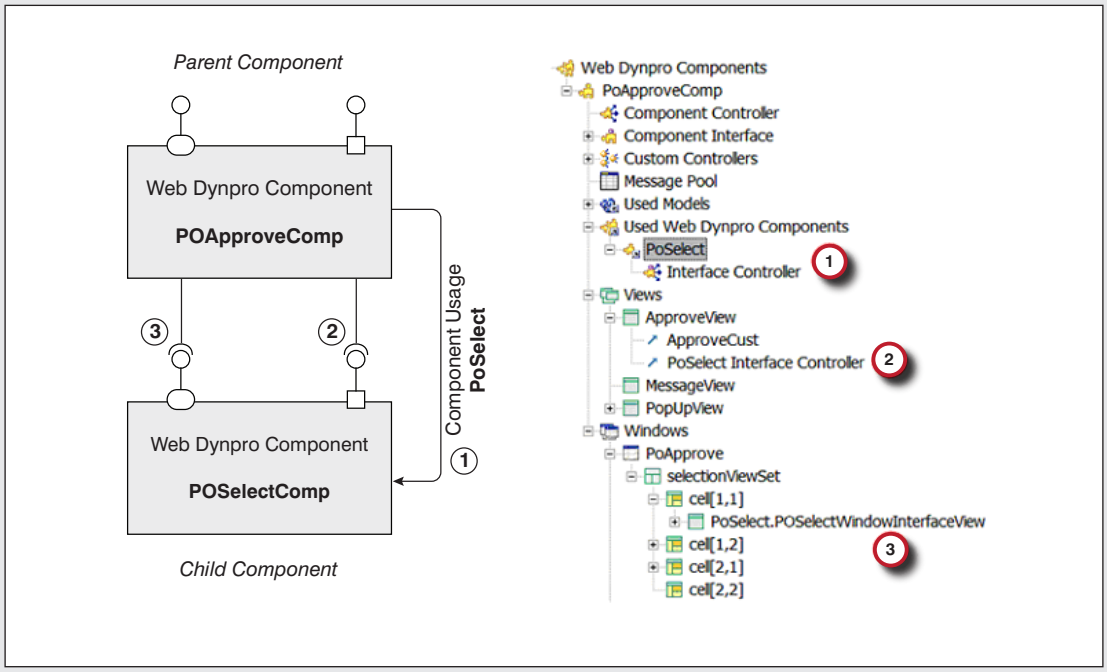


Figure 10 Embedding a Web Dynpro child component in a parent component

component POApproveComp invokes the `refreshData()` method exposed by the interface controller of the used component POSelectComp.

- **Component interface view embedding:** To use its UI, you visually embed the component interface view POSelectWindowInterfaceView in the PoApprove window of the parent component (③) by placing it inside cell[1,1] of the view set selectionViewSet.

On the left side is a Web Dynpro component diagram visualizing the component usage dependency between the two components POApproveComp and POSelectComp. The component usage PoSelect is explicitly drawn with an arrow (①). The invocation of a component interface controller method is drawn with a connector between the parent component and the component interface controller port symbol of the used child component (②), linking a socket symbol with a ball symbol. The embedding of a component interface view is drawn with a connector between the parent component and the component interface view port symbol of the used child component (③), again linking a socket symbol with a ball symbol.

Differentiating a Web Dynpro component usage from its associated component instance

To understand the Web Dynpro component usage entity, you need to distinguish between the entity of the component usage itself and the component to which it refers. More generally, you need to distinguish between Web Dynpro components and Web Dynpro component usages declared at design time and their instance counterparts at runtime.

At design time, the definition of a Web Dynpro component usage entity represents a *usage dependency* between a parent component and its used child component. You define this component usage dependency on a component type or on the variable level by associating the Web Dynpro parent compo-

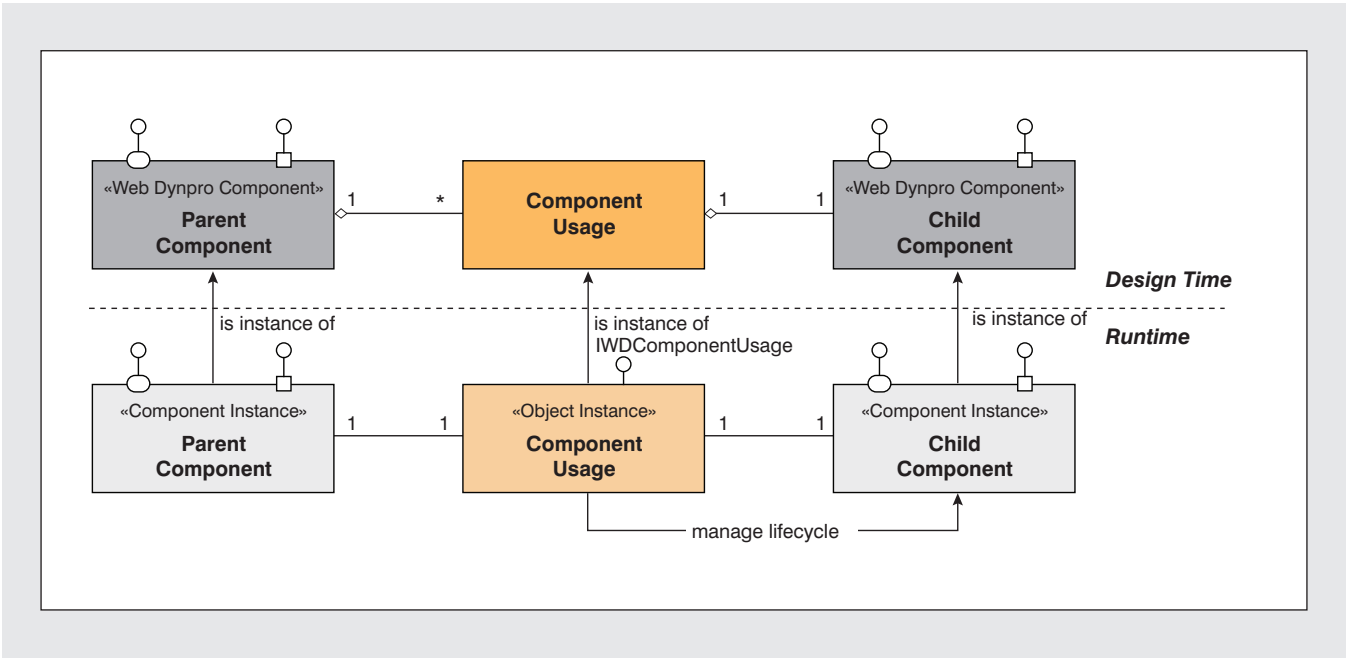


Figure 11 Web Dynpro component and component usage – runtime vs. design time

nent of a specific type with a Web Dynpro child component of a specific type. (It's also possible that a Web Dynpro component usage points to an abstract Web Dynpro component interface definition that has no component implementation inside. In this way a parent component can be loosely coupled with its child component implementations.) In parallel, a variable in Java represents the usage of another object instance. The entity of the component usage is not identical to the actual component instance.

At runtime you must switch from the component type or the variable level to the component *instance* level and from the component *usage entity* level to the component usage *object instance* level, visualized in **Figure 11**:

- **Component instance at runtime:** At runtime, you represent a declared Web Dynpro component with a corresponding Web Dynpro component instance. For root components and child components used with the component usage property Lifecycle=createOnDemand, the Web Dynpro Java runtime environment automatically creates this component instance. Otherwise, you must program it in the controller coding by invoking the IWDComponentUsage API.
- **Component usage instance at runtime:** At runtime, you represent a declared Web Dynpro component usage with an object instance of type IWDComponentUsage.[2] This component usage object points or refers to its associated used child component instance, making it clear that the entity of component usage is not identical to the actual component instance. The component usage object manages the life cycle (creation and destruction) of its related Web Dynpro component instance. The Web Dynpro framework deletes all embedded components when the embedding component is deleted. Depending on the defined value of a component usage's Lifecycle property, either the Web Dynpro Java runtime environment itself (Lifecycle=createOnDemand) manages the component instance creation and destruction or you must do it manually using the application

[2] See JavaDoc of Web Dynpro Java Runtime API com.sap.tc.webdynpro.progmodel.api.IWDComponentUsage (SDN link: https://help.sap.com/javadocs/NW04S/current/wd/index.html).

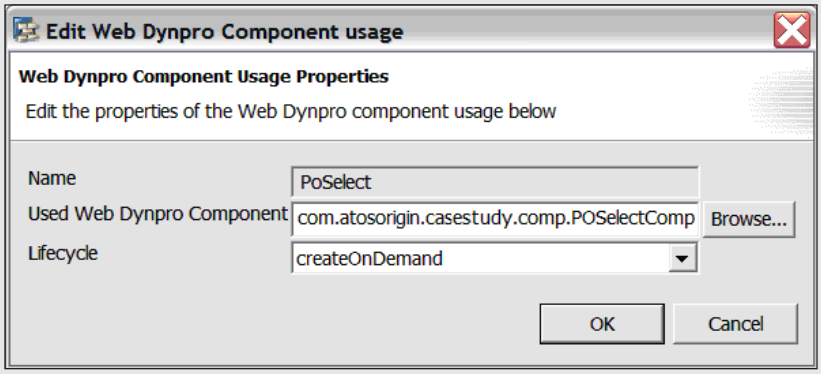


Figure 12 Definition of the Lifecycle property of a component usage

logic that invokes the corresponding `IWDComponent.createComponent()` and `–destroyComponent()–` methods in the controller code. In this way, a component usage instance can be seen as a component instance factory.

Understanding the Lifecycle property of a component usage

When you add a new component usage entity inside an existing Web Dynpro component, the system asks you to declare a value for the Lifecycle property. At design time, you define the Lifecycle property for component usage to specify how to control the creation and destruction of the associated component instance at runtime. An instance of the component usage of the type IWDComponentUsage points to the instance of the used Web Dynpro component at runtime.

You can select between two types of component lifecycle management (see **Figure 12**):

- **createOnDemand:** The Web Dynpro runtime environment itself performs the life-cycle management of the component instance belonging to the component usage. On demand, the Web Dynpro runtime environment creates the component instance automatically and destroys it, if necessary. For example, it creates a UI component automatically as soon as its component interface view is contained in a view assembly for the first time or displayed in the UI.
- **manual:** The life-cycle management of the component instance belonging to the component usage takes place explicitly by implementing it in the application code. This manual life cycle is exclusively defined for a Web Dynpro component usage that points to a Web Dynpro component interface definition, which cannot be instantiated. Instead, the controller logic must create a component instance for a component usage pointing to an abstract component (i.e., a component interface definition) by passing the fully qualified component name and the name of the Web Dynpro development component that contains it to the IWDComponentUsage API. With its IWDComponentUsage interface, the component usage gives a controller the option to control the life cycle of the associated component instance. That is, the controller can create it via `IWDComponentUsage.createComponent()`, if needed, or destroy it via `IWDComponentUsage.deleteComponent()`.

SAP NetWeaver development component model

In this section we move from the first component

model defined by the Web Dynpro Java UI framework to the second important component model defined by the NWDI. With the NWDI component model, you can organize the development entities comprised in your Web Dynpro Java application project, such as reusable Web Dynpro components or Web Dynpro models, into separate development and build units named Web Dynpro development components.

After a short description of the NWDI with its comprised parts, the definition of software components and development components, we introduce the Web Dynpro development component and distinguish it from the Web Dynpro component. We then describe the basic concepts of sharing Web Dynpro entities across several Web Dynpro development components by publishing them in public development component interfaces called *public parts* and by defining the related usage dependencies.

Finally, we'll show you how to use the same external Java Archive file (JAR file) in different Web Dynpro development components and how to deploy it to the SAP NetWeaver AS Java by using a development component of type J2EE Server Component/Library.

NWDI

The development of Web Dynpro business applications for SAP NetWeaver takes place within NWDI. NWDI combines the properties and benefits of a local development environment with a server-based development landscape. It provides a central and consistent development environment to developer teams and supports the entire development process of a software product. It is composed of the following:

- **Design Time Repository:** In an NWDI-based development scenario, you store the development objects in the Design Time Repository (DTR), which is NWDI's central version management system. In Web Dynpro, these objects are the various metadata files for describing a Web Dynpro project or a Web Dynpro development component. Using the DTR, several developers can create component-based Web Dynpro applications in a distributed way, working collaboratively across many locations. Development configurations define different developer-specific views of the development infrastructure. These views include the code lines relevant to a developer or a project, the versions of other used components, the versions of external third-party components, and the addresses of different systems in the NWDI development landscape.
- **Component Build Service:** The central build process is no longer based on command-line tools and Makefiles; it now uses the Component Build Service (CBS) of NWDI. The CBS automatically builds development components and the components depending on them on-demand and incrementally. That is, only the changed files and those files that have a dependency on them are built, instead of building all sources from scratch. The CBS also creates libraries and deployable units for developers and runtime systems.
- **Change Management Service:** The Change Management Service (CMS) transports software components, including the source code and libraries, within an NWDI development landscape. In addition, it supports the automatic deployment of executable software units in central test and production systems.
- **System Landscape Directory:** The System Landscape Directory (SLD) provides services for the administration of system landscapes. These usually include several hardware and software components that depend on each other regarding installation, software updates, and interfaces.
- **SAP NetWeaver Developer Studio:** The SAP NetWeaver Developer Studio supports the development and build processes of DCs and those of Web Dynpro DCs in particular. You can thus deploy the locally built components in a local runtime system for testing.

Component model of NWDI

The NWDI component model is the basis of an efficient development of component-based Web Dynpro applications and enables you to structure

applications as reusable development components. The NWDI component model doesn't change known development objects, such as Java, J2EE, or Web Dynpro entities, but supplements them with special metadata that defines the encapsulation of these objects and interfaces. It provides special development units referred to as software components (SCs) and development components (DCs):

- **Software components (SCs):** Software components are development units that assemble several DCs to form larger, deliverable, and deployable components.

- **Development components (DCs):** DCs are development and build units containing the actual development objects (such as Web Dynpro components, models, component interfaces definitions, applications, or dictionaries). They combine functionally related development objects with reusable and non-overlapping units. Between the individual DCs, you can define usage dependencies where no cycles may occur. The visibility concept applied for DCs is based on the black-box principle in which only the DC parts exposed via public parts can be used by other DCs. You can limit the range of SCs or DCs using a public part by using additional access lists. Every DC has a unique type that determines the technology on which it is based, which development tools to use, and how to compile and archive the DC.

  A DC can only use the development component of another software component if its own software component has defined the usage. Lastly, development components can specify or restrict their visibility for specific software components by defining Access Control Lists (ACLs).

Differentiating Web Dynpro components from Web Dynpro development components

When dealing with componentization in the Web Dynpro for Java context, you will soon recognize that the term *component* occurs in different flavors. Because the term component is used both in NWDI and in the Web Dynpro programming model, we would like to clearly point out the difference between these two components types. It is essential to differentiate between *Web Dynpro development components* and *Web Dynpro components.* Although these two terms are fairly similar they refer to two entities that are technically completely different. They are only related to each other in that Web Dynpro development components are those development components of NWDI that are provided for storing Web Dynpro components and other Web Dynpro entities.

You need to be able to distinguish between the following three component types:

- **Web Dynpro component:** A Web Dynpro-specific component entity representing the main building block in the Web Dynpro Java programming model where UI and controller logic is implemented based on the MVC design principle. In the Web Dynpro programming model, the entity of the Web Dynpro component is the central module for building modular Web Dynpro applications. To better differentiate between functional component objectives, subclasses of Web Dynpro components exist, such as Web Dynpro UI components, implementing user interfaces or faceless Web Dynpro model components implementing the back-end access.

- **Development component:** An NWDI-specific component entity representing a software unit of functionally related development objects of general types for the purpose of versioned storage, deployment and change management. The *development objects* created within an embedding development component are centrally stored as versioned files in the DTR, which acts as a Concurrent Versioning System (CVS) of NWDI. Development component is an umbrella term for several content-specific development component types such as Web Dynpro development components, External Library development components, Java development components, or J2EE development components.

- **Web Dynpro development component:** A special development component type in the component model of the NWDI used to develop,

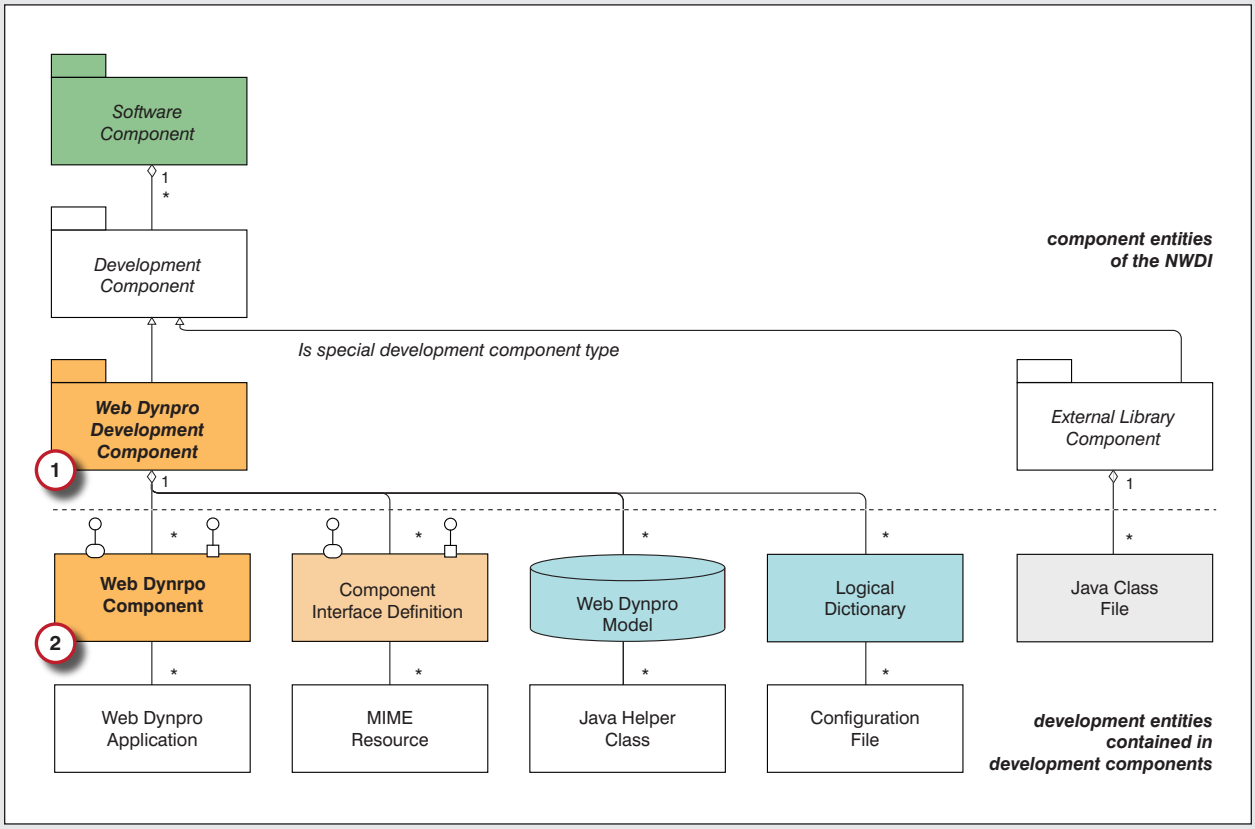


Figure 13 Associations between different components and development objects in the NWDI component model

store, version, build, and deploy Web Dynpro-specific development objects or entities such as Web Dynpro components, Web Dynpro applications, component interface definitions, models, or Java dictionary types in a modular way. Outside of NWDI, the Web Dynpro development component corresponds to a Web Dynpro project with the essential difference that in contrast to Web Dynpro development components, no usage dependencies can be defined between Web Dynpro projects. This is why Web Dynpro projects quickly reach their limits in the distributed development of larger Web Dynpro applications outside of NWDI.

Figure 13 illustrates the relational connection between the different types of software components, Web Dynpro development component (①) and Web Dynpro component (②). A software component can contain several Web Dynpro development components that in turn can store several Web Dynpro components. It is also possible that a Web Dynpro development component does not contain any Web Dynpro components, but only other Web Dynpro entities such as a Web Dynpro models or component interface definitions.

Figure 4 on page 56 shows how the development objects of the case study application are packaged inside different Web Dynpro development components and an External Library DC (**Figure 4**, ⑤). The three Web Dynpro components POApproveComp,

POSelectComp, and TripSelComp are stored inside different Web Dynpro development components. The PoModel component comprises one Adaptive RFC model connecting to an SAP system (**Figure 4,** ④).

Public parts of a development components

Large Web Dynpro applications with a component-based architecture are composed of several Web Dynpro development components. By using several development components, you can split large Web Dynpro applications into clearly structured and, if necessary, reusable parts, and therefore a development team can build one application. Between development components, you can define different usage dependencies that are based on a special visibility concept. *Public parts* represent the development component interfaces that are visible to the outside. Via additional ACLs, you can define which development or software components are allowed to use a public part. Without the definition of an ACL, a public part is visible to all external using components.

Because development components are containers of development objects, you can also define the purpose of the public parts (*public part purpose*). In the NWDI component model, you need to generally distinguish between the two usage types of a public part definition — compilation and assembly:

- **Compilation:** The definition of a public part of the compilation type means that the development objects contained in the public part (also called *public part entities*) are required for compiling, or for the build process of a Web Dynpro development component using this public part. For this purpose, the development component inserts the public part archive in its own Java build path. The public part archive contains all Web Dynpro metadata files and class files that are required for generating its own development component archive. The development component archive of the used Web Dynpro development component must still be created and deployed separately, though, because the usage of a public part of the compilation type does not involve the extension of the development component archive.
- **Assembly:** You must define public parts of the assembly type when the development objects contained therein need to be grouped together, or assembled, to a larger unit at a higher level, that is, by using a development component. You cannot use a public part of this type to compile the parent development component, though. To achieve this, you need to define a second public part of the compilation type.

> ***Tip!***
>
> For Web Dynpro development components, you can always select the compilation type when defining the public part. The second usage type, assembly, is required for embedding non-deployable development component types, such as External Library development components, or in deployable development components, such as J2EE Server Component / Library development components or Web Dynpro development components. Web Dynpro development components, however, can be deployed directly so that they do not need public parts of the assembly type.

To understand the public part definition, you should know that only the compilation usage type is relevant for Web Dynpro development components. Public parts of the assembly type are required for wrapping non-deployable development component types such as Java development components or External Library development components in deployable development components. Because Web Dynpro development components are deployable themselves, public parts of the assembly type are not relevant. For more information on exposing a Web Dynpro model, see the sidebar on the next page.

How to expose a Web Dynpro model in a public part

To expose a Web Dynpro Adaptive RFC model stored in the Web Dynpro development component POModel, the application developer made the following declaration steps:

- Select node *<node for Web Dynpro DC PoModel> – DC MetaData – Public Parts*.
- Use the context menu option New Public Part.
- In the Add to Public Part display frame, enter the public part name *PurchaseModelPP*.
- In the same window, select the Provides an API for developing/compiling other DCs button in The exposed items can be used as a library that group box.
- Enter descriptive information in the Caption field and Description field (optional).
- To add the comprised Adaptive RFC model with name PurchaseModel to the new public part, select the entity type Common Model. Then select the /com/atosorigin/casestudy/model/PurchaseModel checkbox, and click on Finish.

The Adaptive RFC model named PurchaseModel is now added to the new public part PurchaseModelPP and gets displayed as node DC Metadata – Public Parts – PurchaseModelPP – Entities – PurchaseModel. This public part can then be used by other Web Dynpro development components so that their contained Web Dynpro components can use the exposed model (public part entity).

Using public parts of other development components

To use a development object stored in a (Web Dynpro) development component (that acts as a service development component), you must first declare a corresponding public part usage in relationship to the other (Web Dynpro) development component (which acts as a client development component). Naturally the used public part defined by the service (Web Dynpro) development component must comprise the development object to be used as a public part entity. **Figure 14** (on the next page) illustrates this relationship based on the Web Dynpro case study application.

The search component TripSelComp, which is stored in Web Dynpro development component TripSel, needs to use the Adaptive RFC model PurchaseModel stored in another Web Dynpro DC PoModel. In a first step this model was added to the public part PurchaseModelPP exposed by the Web Dynpro development component PoModel as a public part entity (①).

To use PoModel in the Web Dynpro search component, you need to complete the following steps:

- In the Web Dynpro Explorer, first you have to open the nodes *<Web Dynpro development component TripSel> – DC MetaData – DC Definition – Used DCs* to select its context menu option Add Used DC.
- In the Add Dependency display frame, you select the public part entity of the PurchaseModel. Open the nodes *< Web Dynpro development component TripSel > – DC MetaData – Public Parts – PurchaseModelPP – Entities – PurchaseModel*.
- As a dependency type, enable the Build Time (needed for compilation) checkbox and then complete the definition of the public part purpose by clicking on Finish.

Because all development components can use

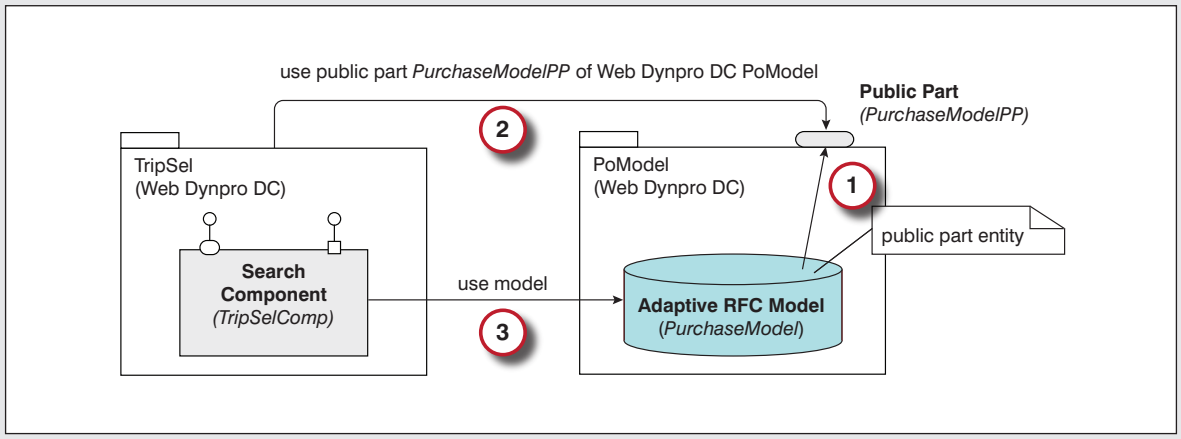


Figure 14 Using a model as a public part entity of another Web Dynpro development component

each other in the same way, they are dependant on each other. When defining a public part purpose, you must specify the dependency type that needs to exist between the using development component and the development component providing the public part. Because the public part entity PurchaseModel (Web Dynpro model) is required in the Web Dynpro components TripSelComp, POSelectComp, and POApproveComp for both declaration and compilation, or for building the Web Dynpro development component archive, respectively, we will select the *Build Time* dependency type.

When adding a new public-part dependency to a Web Dynpro development component, you can apply the following rules to correctly define the dependency type:

- **Build Time:** As a general rule, you need to specify the Build Time type as the public part dependency when using Web Dynpro development components.
- **Deploy Time:** You select the Deploy Time dependency type so the deployment of a Web Dynpro development component will automatically be cancelled when one of its used development components is missing in the runtime system.
- **Run Time:** You select the Run Time dependency type when the public part of assembly type contains a JAR file that must be visible in the using development components at runtime for class loading or instantiation. However, if the used public part contains Web Dynpro entities, the required runtime dependencies are automatically entered in the development component archive by the Web Dynpro tools, even without the Run Time dependency type being set. Still, you should always select the Run Time dependency type whenever you use public parts of other Web Dynpro development components.

Using the same JAR file in different Web Dynpro development components

For the case study application, we needed a component to hold all of the external libraries that we wanted to use but that were not part of the standard Web Dynpro build or SAP NetWeaver AS. For example, we added an external library that accessed the UME on the portal into the library development component. SAP provides the UME external library in the form of a JAR file in SAP NetWeaver 7.0.[3] All the

[3] In SAP NetWeaver CE 7.1 the UME library is exposed by the existing J2EE Server Component Library development component tc/je/user-managment/api of software component ENGFACAD (engine façade). You can then use this development component to compile your controller code accessing the UME library at design time. The External Library development component used for compilation is then no longer needed.

components that needed to access the UME then referred to this single version of the JAR file. It saved adding the JAR file to a large number of individual applications or development components. Naturally, an external JAR file comprises compiled Java classes without sources and it is not already deployed on SAP NetWeaver AS. Typical examples are external JAR files provided by an open source library used to read, write, or modify Excel files.

Here is a solution for using the same external JAR file in different Web Dynpro development components fulfilling three main requirements:

- **Central storage place:** The JAR file is stored in a central development component. To keep this JAR file independent from the runtime environment, use a non-deployable External Library development component. With this single-source approach you can avoid the same JAR file being physically stored in different development components.
- **Compilation at design time:** At design time, the JAR file is needed for compilation by several Web Dynpro development components. The public part of the External Library development component must, therefore, be of type compilation. Any Web Dynpro development components that require the JAR file could be called Web Dynpro *client* development components.
- **Classloading at runtime:** At runtime, the JAR file must be deployed on SAP NetWeaver AS Java so that the Web Dynpro Java runtime environment can load the JAR file's classes. To deploy the JAR file to the server, you have to wrap its embedding (non-deployable) External Library development component in a separate deployable development component of type J2EE Server Component/Library. The External Library development component must therefore expose its comprised JAR file in another public part of assembly type. In this case, the JAR file is added to the deployable archive of its deployable wrapperdevelopment component (using the second public part). By deploying the wrapper development component to the SAP NetWeaver AS Java, the JAR file also gets deployed to the server to be found (and is found) by the class loader at runtime.

  To load the JAR file's classes at runtime inside the Web Dynpro client development component, the Web Dynpro Java runtime must know the fully qualified name of the development component comprising the JAR file. You need to define a runtime dependency between the Web Dynpro client development component where the JAR file's classes are invoked and the J2EE Server Component/Library development component comprising them. In SAP NetWeaver 7.0 this runtime dependency is defined as Library reference in the properties of a Web Dynpro development component. The Library reference entry must be the technical name of the J2EE Server Component/Library development component. For example, *vendor.org/jsclib~ext~exlexp* is the technical name of a J2EE Server Component/Library development component with name *vendor.org/jsclib/ext/exlexp*.

See the sidebar on the next page for more information on how to fulfill these requirements.

The solution in SAP NetWeaver CE 7.1 is slightly different. The development component type J2EE Server Component/library is depreciated and replaced

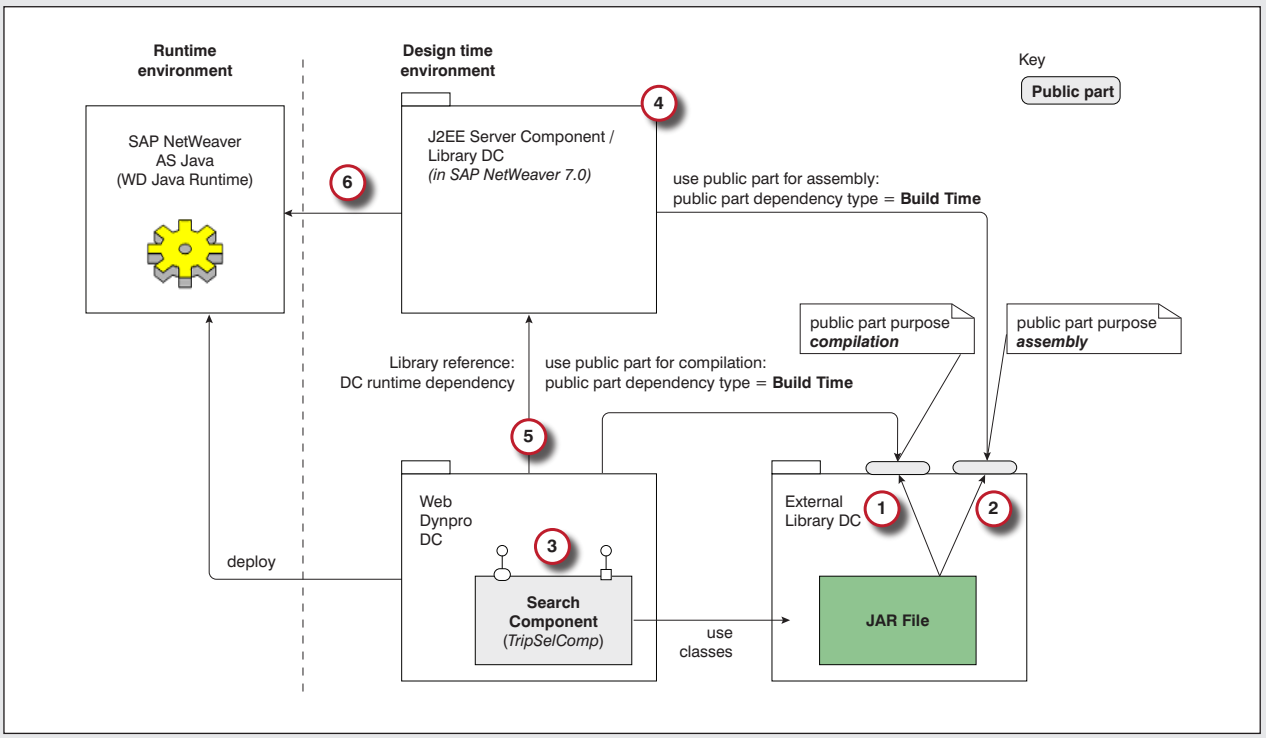
Tip!

To implement and reuse your own Web Dynpro Java-specific Java utility or helper classes such as classes for sorting or filtering tables to be invoked in your Web Dynpro controllers, you do not need to create and deploy a corresponding JAR file. Simply implement these helper classes in folder *src* of a Web Dynpro development component, build and deploy it to the SAP NetWeaver AS and expose the classes to other Web Dynpro development components in a public part of type compilation.

Fulfilling SAP NetWeaver 7.0 requirements for this solution

The solution fulfilling all these requirements in SAP NetWeaver 7.0 includes the following tasks, as illustrated below:

- Store JAR file in the libraries folder of an External Library development component.
- Expose the same JAR file in two separate public parts, one for assembly (①) and another one for compilation (②).
- Use the public part with purpose compilation in the Web Dynpro development component where you need the JAR file to implement your Java code, for example, inside a Web Dynpro component controller class (③).
- Add a development component of type J2EE Server Component/Library, and use the second public part (assembly type) of the External Library development component there (④).
- Define a Library reference to the J2EE Server Component/Library development component in the project properties of the Web Dynpro client development component, where classes and interfaces of the external JAR file are invoked (⑤). In Web Dynpro Explorer, open the context menu of the node *<Web Dynpro client development component name>* and select the Properties item. In the properties window select Web Dynpro References – Library references and enter the technical development component name, e.g., vendor.org/jsclib~ext~exlexp.
- Deploy the development component of type J2EE Server Component/Library wrapping the External Library development component to SAP NetWeaver AS Java (⑥).



by the new development component type Java EE / Enterprise Application. This is used to assemble and deploy a JAR file in an External Library development component. In addition, the Library reference defined on the project level is replaced with a runtime usage dependency to the Java EE Component/Enterprise Application development component defined on development component metadata level. To define this development component usage dependency, you no longer need to manually enter the technical name of the used development component.

Quick development component build process

If a new public part is defined for a Web Dynpro development component, this means that every triggered development component build process involves the creation of the associated public part archive. You can trigger a development component build process with context menu item <*Web Dynpro development component node name*> – Development Component – Build in the Web Dynpro Explorer.

The definition of several public parts with numerous development objects can cause a noticeable slowdown of the development process. You should therefore definitely consider the following tip for accelerating the process.

> ***Tip!***
>
> Public part archives are newly created only if a development component build process is triggered. If the development objects contained in the public part do not change, it is sufficient to run the normal and faster project build process with context menu item <*Web Dynpro development component node name*> - Rebuild Project in the Web Dynpro Explorer. Particularly for larger Web Dynpro development components, this procedure can considerably accelerate the development process for repeated testing.

What's New in SAP NetWeaver CE 7.1

In SAP NetWeaver CE 7.1, the Web Dynpro component model is further streamlined and enhanced with new, interesting functions not available in SAP NetWeaver 2004 and 7.0 (see **Figure 15** on the next page). You will now be able to migrate existing Web Dynpro components to the new component model if you want to apply any of the new component functions. The old component model will be supported in SAP NetWeaver CE 7.1 for downward-compatibility reasons with prior SAP NetWeaver releases.

So what can you expect?

- **New window controller replaces interface view controller class:** Web Dynpro windows will have their own new *window controller* class. The new window controller will replace the existing component interface view controller class, which will no longer exist.[4] This change extends the already given implementation relation between a component interface view and its related window (which has a 1:1 relation in SAP NetWeaver 2004 and 7.0) to the controller level.
- **New window plugs simplify cross-component navigation:** You can define inbound and outbound plugs inside a window. Before SAP NetWeaver CE 7.1, you had to create navigation plugs in views. With defined window plugs, developers will be able to easily implement navigation to a specific view inside a window when a component interface view (implemented by the window) is reached via navigation link. You just have to define a navigation link from the window's outbound plug to an inbound plug of an embedded view. The window's outbound plug is fired within the window controller class. In previous SAP NetWeaver releases, such a cross-component navigation scenario had to be resolved with server-side eventing and an additional navigation dispatcher view, which is now no longer needed.

[4] Note that component interface view controller classes, *not* component interface views, will no longer exist in newly created Web Dynpro component. Existing Web Dynpro components comprising component interface view controllers that are not migrated will still be supported.

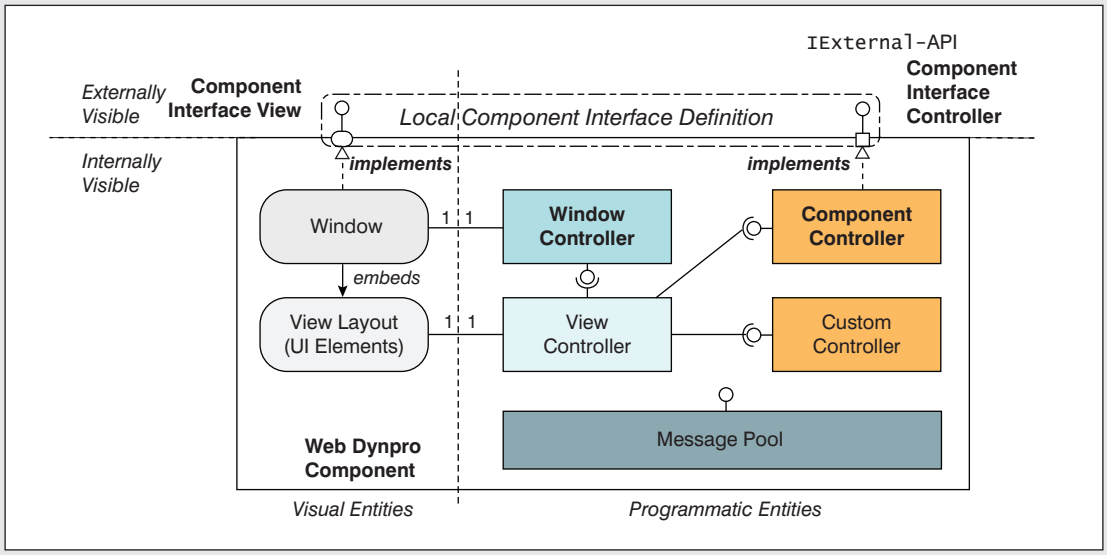


Figure 15 Enhanced Web Dynpro component anatomy in SAP NetWeaver CE 7.1

- **Window and component controllers implement component interface definitions:** On the interface controller level, the component interface controller class will no longer exist. Instead, the existing component controller class will act as an implementer for all component interface controllers that are defined in the local and/or one or more standalone component interface definitions.

  In the same way, the newly introduced window controller class will act as an implementer of one or more component interface views that are defined in the local and/or one or more external component interface definitions.[5] A window controller implements a component interface view with event handlers for the defined inbound, startup, and resume navigation plugs and with generated wdFirePlug<*plug name*>() methods to fire outbound, exit, and suspend navigation plugs. These methods can be invoked inside a window controller by accessing the controller's generated IPrivate<*view name*> API and outside (from view, custom, or component controllers) by accessing the controller's IPublic<*view name*> API.

- **Local component interface definition:** In the new SAP NetWeaver CE 7.1 Web Dynpro component model, the component interface view and the component interface controller are no longer associated with a corresponding component interface view controller class and a component interface controller class. The new window controller class now implements the code formerly comprised in the component interface view controller class, and the component controller class now implements the code formerly comprised in the component interface controller class. In this way, the previous component interface of a Web Dynpro component implementation becomes the *local component interface definition* defined inside a Web Dynpro component according to the *external* or *standalone component interface definition*, which already existed in SAP NetWeaver 2004 and 7.0. A local component interface definition can be seen as a self-defined component interface definition inside a Web

[5] Every component interface view defined in either the local or a standalone component interface definition must have a unique implementing window inside the component. In contrast, a window can implement more than one component interface view, which was not possible in the SAP NetWeaver 2004 and 7.0 component model.

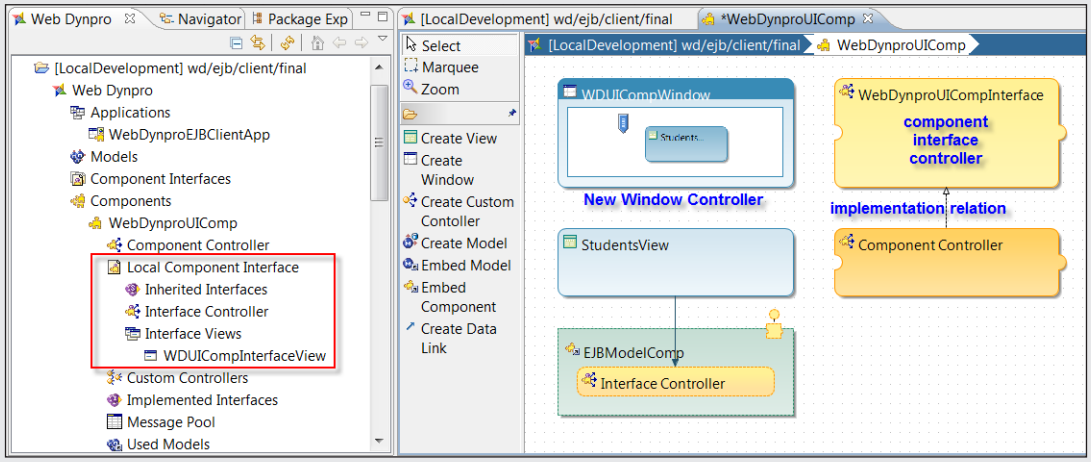


Figure 16 New Web Dynpro component displayed in SAP NetWeaver CE 7.1 Developer Studio

Dynpro component implementation. An embedding parent component that directly uses a Web Dynpro component implementation uses its exposed local component interface definition. See **Figure 16**.

The local component interface definition remains unaffected when its Web Dynpro component implements standalone component interface definitions. This wasn't possible in the former component model in which all implemented standalone component interface definitions were reflected in the component interface and its implementing classes (component interface view controller class and component interface controller class).

- **Component interface definition inheritance:** Local and standalone component interface definitions can extend other standalone interface definitions on the meta-model level. In this case, a Web Dynpro component must implement all parts defined in the implemented component interface definition and its inherited component interface definitions. With native interface inheritance set, the IExternal<*component interface definition name*> API generated for every local or standalone component interface definition extends the IExternal<*component interface definition name*> APIs of all inherited component interface definitions.

- **New component interface view containers:** In the new Web Dynpro component model, you will be able to declare view containers inside a component interface view (part of a local or standalone component interface definition). A parent component can then externally inject or embed its own view or a component interface view of another user interface component into the exposed interface view container. The user interface component implements its defined interface view containers with ViewContainerUIElements on the window level. For this, every interface view container must be implemented by a unique ViewContainerUIElement embedded into the window (implementing the interface view in which the interface view containers are defined).

  With this technique, you can develop Web Dynpro layout components that implement the position or layout of the views and interface views maintained by the parent component outside. The layout component must only expose its interface view containers as placeholders so that the parent component can embed other component interface views; the layout component must not use or embed these other user interface components.

- **Use of visual components in referencing mode:** In SAP NetWeaver 2004 and 7.0, the component usage referencing mode was only supported for referencing faceless Web Dynpro components inside other, mostly visual, Web Dynpro components (see the section "Component usage referencing pattern" in the next article). This restriction is no longer given in SAP NetWeaver CE 7.1 so you can now enter another component usage that points to a user interface component in referencing mode. In this way a Web Dynpro component can display another component interface view without managing its component instance lifecycle. The embedded user interface component can be managed (created, destroyed) by the parent component outside. This technique is particularly useful when embedding another Web Dynpro user interface component via an abstract external component interface definition (loose component coupling). The embedding user interface component can then enter another user interface component usage, maintained by the parent component, in referencing mode to display its component interface view. In this way, the embedding user interface component is released from creating and destroying the user interface component (implementing the standalone component interface definition) at runtime itself; this is done outside by the parent component.

Conclusion

In the second part of this article series, we focused on the conceptional aspects of implementing a component-based architecture in Web Dynpro Java using the NWDI.

We first illustrated the Web Dynpro component concept with the component anatomy, component interfaces, and component usages for assembling multiple Web Dynpro components. Based on these concepts, we then described basic technical, implementation-specific and declaration-specific details of the Web Dynpro case study application, which included declaring component usages, invoking component interface controllers, firing and catching events across component borders, sharing data between components with interface context mapping, and embedding visual components using component interface views. We also presented details of the Web Dynpro component diagrams used in the article.

We then covered the development component model defined by the NWDI, concentrating on Web Dynpro development components to package, build, and deploy Web Dynpro components and other Web Dynpro development objects. Alongside the elementary concepts of sharing Web Dynpro entities across several Web Dynpro development components by defining public parts and the related usage dependencies, we described how to use an external JAR file in different Web Dynpro development components and how to deploy it to SAP NetWeaver AS Java.

Finally, we explained the new streamlined Web Dynpro component model in SAP NetWeaver CE 7.1 that provides enhanced functions not available in SAP NetWeaver 2004 and 7.0.

In the next article, we'll present a more in-depth technical description of Web Dynpro component implementation and packaging techniques. We'll explore Web Dynpro componentization patterns in more detail, such as interface context mapping, cross-component eventing, controller delegating, component usage referencing, and the component interface definition strategy. We'll also cover some tips and tricks to help you work more efficiently with Web Dynpro development components inside the NWDI such as finding the right granularity of Web Dynpro development components to optimize productivity, maintainability, and flexibility.

