Leverage component-based architecture in Web Dynpro Java business applications

Part 3 — Componentization patterns in practice

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 120.)

This is the last article in our series on the componentization of Web Dynpro Java business applications. In our 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. In the second article, published in the July/August 2008 issue, we focused on the conceptional aspects of implementing a component-based architecture in Web Dynpro Java using the SAP NetWeaver Development Infrastructure (NWDI). We explored the two independent component models that exist in the Web Dynpro Java application development context: the *Web Dynpro component model* and the *Web Dynpro development component*.

In this article, we'll present a more in-depth technical description of Web Dynpro component implementation and packaging techniques. We first explore Web Dynpro componentization patterns, including the component separation pattern, component usage declaration pattern, component interface controller invocation pattern, cross-component eventing pattern, and component interface context mapping pattern. Next we explore the external component interface context mapping pattern to access a parent component's data context inside a child component and then describe the component interface view embedding pattern to design componentized user interfaces. Then we discuss the component controller delegation pattern, the component usage referencing patterns, the faceless model component patterns, and then component interface definition strategy patterns, as well as their implementations. After exploring Web Dynpro componentization patterns, we address those patterns related to the separation of diverse development entities into Web Dynpro development components. We explain how to separate different Web Dynpro development-related entities — such as models, dictionaries, and different types of Web Dynpro components — into Web Dynpro development components and how to optimize these development components. Finally we provide 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.

Web Dynpro componentization patterns

The Web Dynpro componentization patterns described in this article show you how to concretely design and implement component-based architecture in Web Dynpro Java business applications. These building blocks capture proven and mature solutions for recurrent problems in the Web Dynpro component — as well as development component context — and can therefore decisively improve your Web Dynpro Java development projects.

The described "componentization patterns" are more "reference patterns" or "cookbook recipes" than true "design patterns,"[1] as described by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides in their book *Design Patterns: Elements of Reusable Object-Oriented Softeware* (Addison-Wesley, 1994). Nevertheless, the given Web Dynpro componentization patterns are also "three-part rules, which express a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a software configuration which allows these forces to resolve themselves."[2]

In this article *context* refers to a component-based Web Dynpro Java application, *system of forces* refers to Web Dynpro component-specific design and implementation problems to be solved, and *software configuration* refers to the declaration steps inside the Web Dynpro Java design time environment and the implementation steps inside the Web Dynpro Java controller classes.

The documentation format that we use to describe Web Dynpro Java componentization patterns is comprised of the following sections:

- **Problem:** A description of a concrete component-related problem to be resolved.
- **Motivation:** A scenario consisting of a problem and a context in which this pattern can be used.
- **Example:** An example of the pattern.
- **Solution:** A description of an implementation of the pattern. In some cases, we divided the solution into an **Implementation** part, addressing the controller class implementation aspects of the solution, and a **Declaration** part, addressing the Web Dynpro tool specific modeling aspects of the solution.

Let's take a closer look at some development and declaration patterns that you'll repeatedly encounter when building componentized Web Dynpro Java applications. These patterns will also provide some dos and don'ts that optimize the development process and the architecture of your own Web Dynpro Java applications. They are a concise way to represent the knowledge applied in a state-of-the-art componentized Web Dynpro application. Most patterns are illustrated with examples from the Web Dynpro case study application presented in the first article of this series.

Component separation pattern

This componentization pattern is the basis for all other patterns. Separating Web Dynpro applications into multiple components is the first prerequisite for all other component-related patterns.

Problem: For large applications, how can you optimize the architecture, development process, and maintenance of a large Web Dynpro Java application to provide a reduced total cost of ownership, rapid and team-based development, clear separation of concerns (SoC), and the reuse of controller code and user interface parts?

Motivation: If you have a monolithic Web Dynpro component that implements several concerns, it may

[1] For more details, see "Design Patterns" and "Design pattern (computer science)" on Wikipedia at http://en.wikipedia.org (2008).

[2] Gabriel, Dick, "A Pattern Definition." http://hillside.net/patterns/definition.html (2008-06-23).

Web Dynpro component reuse

Allowing code reuse usually requires additional effort when developing the code to be reused. Planning for too much reuse, however, can significantly slow down the development speed of applications and is not necessarily justified if the actual cases of reuse are highly unlikely or not even known at development time. Usually, the additional cost of developing reusable code is only justified if there are at least several consumers of the reuse component provided, as is the case for PoSelectComp and TripSelComp in the Web Dynpro case study application. Do not try to make everything a reusable component.

Web Dynpro components should not be used to implement a single helper method. Declaring and implementing a Web Dynpro component, combined with defining related component usages, might cause too much overhead during development. To keep memory usage low at runtime, you would have to use the referencing mode for the new component, and this would require additional coding or declaration (see the "Component usage referencing pattern" section on page 110).

Instead, a pure Java utility class should be sufficient. You can expose a utility class via public parts of development components as well and reference it from all of your Web Dynpro development components. If you want to apply other Web Dynpro dependant functionality inside the helper method that requires a Web Dynpro component (like shared context nodes, etc.), then a component makes sense and you could incorporate the new method there as well.

comprise a rich functionality regarding its user interface, back-end access, and application logic, which can result in a number of major drawbacks. A large component might be highly limited or provide no reuse (for more details on component reuse, see the sidebar above). Different developers may not be able to simultaneously implement it because of its complexity. It may have a longer build, or it may have archive and deployment turnaround times that are based on missing an incremental build process.

Example: The Web Dynpro case study application is clearly separated into three root components: PoGRComp (purchase order good receipts), PoApproveComp, and PoChangeComp. The two reusable child components, PoSelectComp and TripSelComp (search component), were reused by all three root components.

Solution: Separate your large Web Dynpro application into several functionally decoupled and preferably reusable Web Dynpro components. The components should provide specific and singular functionality. Each functionality is provided once and then used in many applications.

You want to be sure that, with its visual (component interface views) and programmatic (component interface controller) interfaces, a Web Dynpro component exposes its user interface, context data, and controller logic (methods, events) to other Web Dynpro components. You need to specify the component interface so the component user can access the component via this interface.

Component usage declaration pattern

The previously described component separation pattern implies the existence of separate "decoupled" Web Dynpro components. To "couple" these components with one another, you first have to apply another fundamental componentization pattern called a *component usage declaration*

pattern. As all possible interdependencies[3] between two Web Dynpro components are based on a previously declared component usage dependency,[4] this pattern is the second prerequisite for all of the patterns described in the upcoming sections. Without the existence of two separate Web Dynpro components and without declaration of a corresponding component usage relation, all of the remaining componentization patterns cannot be applied.

Problem: What can you do if a Web Dynpro parent component needs to use another Web Dynpro component for the purpose of embedding or navigating to that component's interface view via the parent's own UI? What if the parent component needs to consume or transfer context data across component borders? Or what if the parent component wants to invoke methods implemented by its used Web Dynpro child component, or subscribe to events fired by the child component?

Motivation: You want to be able to reuse existing functionality implemented in another Web Dynpro component or defined inside another abstract Web Dynpro component interface definition (as defined in the second article). You want to compose a rich user interface by nesting visual Web Dynpro components (i.e., component interface views). You want to share (send or retrieve) context data across component borders, or perhaps connect to the business logic (represented by a Web Dynpro model) in a faceless Web Dynpro component.

Example: In the Web Dynpro case study application, the PoSelectComp component for selecting purchase orders is reused by all three root components (i.e., PoGRComp, PoApproveCom, PoChangeComp). The purchase order component PoSelectComp itself reuses the TripSelComp search component for searching buildings, cost centers, and projects.

[3] Examples of "interdependency" between two Web Dynpro components or, more precisely, between a parent component and its child component, include component interface view embedding, cross-component navigation, cross-component eventing, cross-component context mapping, or component interface controller method invocation.

[4] See the "Defining Web Dynpro component usage dependencies" section in the second article of this series.

Solution: Add a Web Dynpro component usage to the parent component that points to the used Web Dynpro child component. Whenever functions provided by another Web Dynpro component (or functionality defined by a Web Dynpro component interface definition without implementation) have to be used in an embedding Web Dynpro component, a corresponding Web Dynpro component usage entity must be declared first inside the Web Dynpro Tools development environment.

Declaration:
In Web Dynpro Explorer, select the parent component (PoSelectComp), as shown in **Figure 1**. In the Used Web Dynpro Components node, select the context menu item Add Used Component. In the New Component Usage dialog, enter a component usage name, for example, POSelection (①). Select the Web Dynpro child component to be used via Browse button (②), and keep the default Lifecycle property createOnDemand unchanged in most cases (③). Click on Finish.

Note!

A Web Dynpro child component can only be used by its embedding parent component when either the child component is contained in the same Web Dynpro development component, or stored in another Web Dynpro development component when it is made "visible" by defining a corresponding public part usage relation between both development components.

Now that you've added the newly named Web Dynpro component usage entity to the parent component (listed in Web Dynpro Explorer under the Used Web Dynpro Components node (④) so you can define other child component-specific relations (e.g., context mapping, event subscription, component interface view embedding, and navigation link definition).

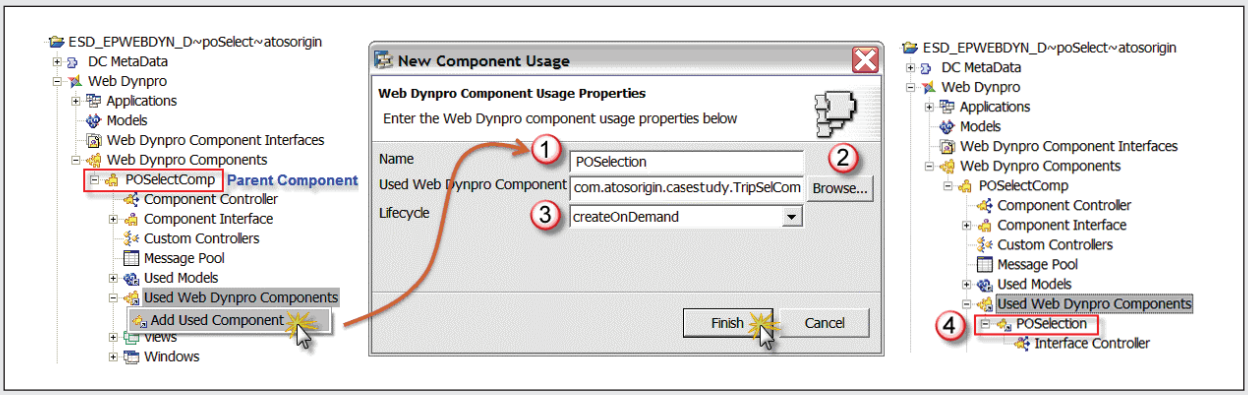


Figure 1 Adding a Web Dynpro component usage to the parent component

Component interface controller invocation pattern

This componentization pattern requires that the component separation pattern and the component usage declaration pattern be applied first.

Problem: How can a parent component interact with its embedded child component? How can a Web Dynpro parent component delegate the controller logic to a child component (i.e., to one of its comprised component or custom controllers)? How can a parent component invoke methods exposed by the component interface controller of a child component?

Motivation: Many used Web Dynpro components expose methods in their interface controller to be called inside the embedding parent component. You want to implement controller logic once inside a single Web Dynpro component, and you want to call that controller logic from another component. You want a child component to react on the changes inside its parent component. A visual Web Dynpro parent component must execute a back-end call implemented in a faceless child Web Dynpro component that is bound to a Web Dynpro model. A child component needs to be initialized or "configured" by its embedding component to run correctly.

Solution: Inside a controller of the parent component, invoke a method exposed by the IExternal API of the child component's interface controller.

Declaration:
Declare a method in the interface controller of the used child component. Based on this declaration, the generated IExternal API with the name IExternal<*child component name*> gets automatically enriched with the defined method to be "externally" invoked in a parent component. In the invoking controller of the parent component, declare a controller usage relation to the component interface controller of the used child component.[5] After you define a component interface controller usage relation, the IPrivate API (accessed using the shortcut variable `wdThis`) of the invoking controller gets automatically enriched with the method `wdGet<child component usage name>Interface()`, which returns the typed IExternal API of the child component's interface controller. (For more information on the component interface controller as it relates to this context, see the sidebar on page 100.)

[5] This is done in the Properties tab of the invoking controller (custom, component, view, interface view) of the parent component. A corresponding component usage relation to the child component must be declared beforehand.

Defining the right usage dependency

After you've defined a component usage within the embedding component, you can then define up to three types of usage dependencies at controller or UI level at design time:

- **Component usage:** A controller of the embedding component uses the component usage to control the life cycle of the associated component instance. It is also possible to provide a component usage with the reference to another component usage of the same type. The used component usage can be accessed by a controller via the IWDComponentUsage API.
- **Component interface controller:** A controller of the embedding component uses the component interface controller that is associated to the component usage. Thus, the context exposed in the component interface (for defining context mapping chains) and its methods and events are visible to the controller. The component interface controller provides its IExternal API as an interface to a used component.
- **Component interface view** (optional)**:** A window or ViewContainer UI element can embed the component interface view of the component instance associated to the component usage. By using the inbound and outbound plugs defined by the component interface view, navigation links can be defined across components. Because non-visual components do not have component interface views, this usage dependency is only available to UI components.

Implementation:
Implement the defined method in the child component's interface controller by delegating the interface controller method call to that same method inside the component controller. (This delegation principle is explained in the "Component controller delegation pattern" section on page 108.) In the parent component's controller class, invoke the interface controller method of the used child component with the code line:

```
wdThis.wdGet<child component usage
  name>Interface().<method name>();
```

Example: In the case study application, the user approves a purchase order in the PoApproveComp root component. To make the embedded child component PoSelectComp react on this approval step triggered by the user, its parent component PoApproveComp invokes the method `refreshData()` exposed by the child component's interface controller with the code line:

```
wdThis
  .wdGetPOSelectCompInterfaceController()
  .refreshData();
```

To invoke this method in the component controller of component PoApproveComp, you must first define a controller usage relation to the interface controller of the used component POSelectComp. The `wdGetPOSelectComInterfaceController()` method gets automatically added to the IPrivate API of the invoking root component controller after you've defined a corresponding interface controller usage dependency. This generated method returns the IExternalPOSelectCompInterface API of the child component's interface controller in which the declared interface controller method `refreshData()` can be invoked. See **Figure 2**.

Cross-component eventing pattern

This componentization pattern requires that the component separation and the component usage declaration patterns be applied first.

Problem: How can a child component interact with its embedding parent component? How can a child component make its parent component react on its

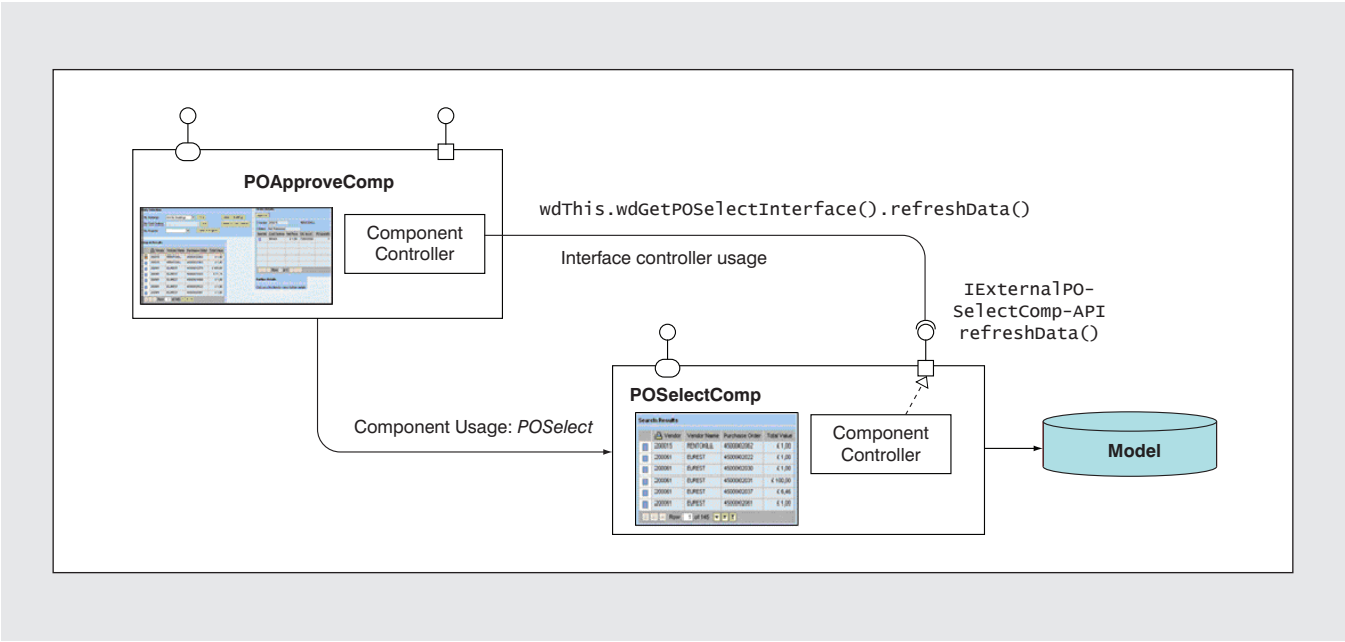


Figure 2 Applying the component interface controller invocation pattern

own controller logic or on user actions? How can a parent component handle events fired by its child component?

Motivation: A child component needs to interact with its parent component on the controller logic level so that the parent component can react on it. Since one of the most important requirements for components is that they don't make any assumptions about their "surrounding," there cannot be any direct interaction between a child component and its embedding parent component via a method invocation. You need to create an "outbound" interaction in which the child component is released from "knowing" its parent component.

Solution: Create an outbound interaction of type event source/event sink so the used child component can throw events that you defined in its component interface controller (event source) and the events can be captured by an event handler in a controller of the embedding parent component (event sink).

Declaration:
Declare an event in the interface controller of the used child component. In the parent component, define a component usage for this child component. Optionally, you can transfer values or objects to the event handler by defining event parameters. In a controller of the parent component, declare a controller usage relation to the component interface controller of the used child component. (For more information on the component interface controller, see the sidebar on page 100.) In the same controller, declare an event handler[6] (e.g., onReceiveBuildingSearch) and subscribe it to the event (e.g., BuildingSearch) exposed by the child component's (e.g., TripSelComp) interface controller (see **Figure 3** on the next page).

Implementation:
After you define an interface controller event in the child component, the IPublic API[7] of the component interface controller gets enriched with the new method `wdFireEvent<event name>( [event parameter {"," event parameter}])`. To fire the interface controller

[6] This is done in the Methods tab of the event-handling controller (custom, component, view, interface view) of the parent component.

[7] The IPublic API of a Web Dynpro controller can only be invoked by other controllers of the same Web Dynpro component.

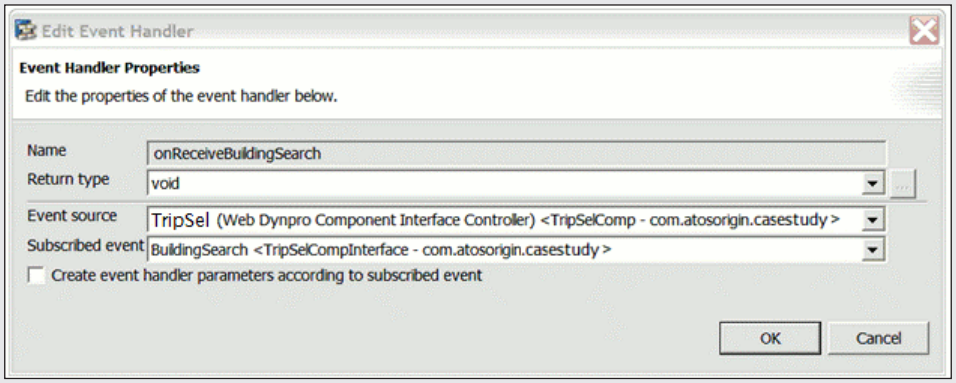


Figure 3 Declaring an interface controller event subscription inside a parent component

event from within the component controller in the same component,[8] first declare an interface controller usage relation in the Properties tab of the component controller. Then, in the component controller class, you can fire the interface controller event with the following code line:

```
wdThis
  .wdGet<component
  name>InterfaceController()
  .wdFireEvent<event name>(
   [event parameter {","
   event parameter}]);
```

Based on the declared event subscription inside the parent component, the Web Dynpro Java Runtime environment invokes the subscribed event handler in a controller of the parent component.

Example: In the Web Dynpro case study application, the user selects one of three criteria buildings, cost centers, or products to search purchase orders that relate to the selected criteria. The lowest-level reusable component, TripSelComp, only implements the user interface for the search input form (choose a building, cost center, or product), whereas its embedding parent component PoSelectComp implements the controller logic to retrieve all matching purchase orders and display them in a table UI.

The child component TripSelComp fires the interface controller event BuildingSearch after the user's building search criteria selection. The embedding purchase order selection component PoSelectComp must then handle this event to retrieve and display all matching purchase order search results.

We now assume that the required declarations are correctly made. As shown in **Figure 4**, the call sequence from firing the interface controller event BuildingSearch in child component TripSelComp to handling it in parent component PoSelectComp should work like the following:

- In the child component TripSelComp: In the selection view controller, the action event-handler invokes a public component controller method `fireBuildingSearch()`. This method is additionally declared in the component controller to avoid a direct invocation of the component interface controller from within the view controller (see the section "Component controller delegation pattern").

[8] For the sake of a simpler, optional component migration in SAP NetWeaver CE 7.1, you should only call the component interface controller from within the component controller, not from within a custom or view controller directly. See the section "Component controller delegation pattern" later in this article.

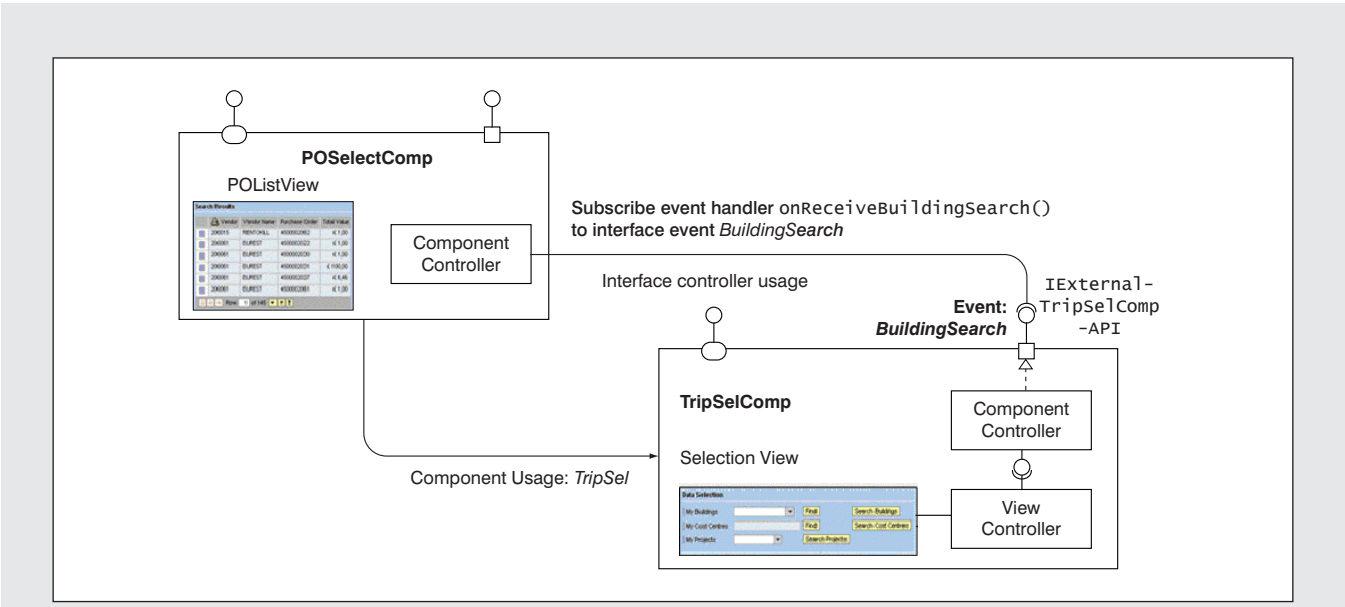


Figure 4 Cross-component eventing between two Web Dynpro components

- In the child component TripSelComp: The component controller method `fireBuildingSearch()` fires the defined interface controller event BuildingSearch with one code line:

```
wdThis
  .wdGetTripSelCompInterfaceController()
  .wdFireEventBuildingSearch
  (criteriaString);
```

- In the parent component POSelectComp: The parent component controller event handler `onReceiveBuildingSearch()` is subscribed to the interface controller event BuildingSearch of the used component TripSelComp. The event

Note!

Download the Web Dynpro naming conventions document from the SAP Professional Journal Web site at http://www.sappro.com/downloads.cfm. Adhering to consistent naming conventions will help to ensure the successful support, modification, and maintainability of your component-based Web Dynpro applications.

Note!

Web Dynpro does not yet support the "external" subscription of interface controller event handlers. This means that a parent component cannot "externally" subscribe an event handler method of its child component's interface controller to its own "external" event source. This would be an alternative to interface controller method invocation because the child component would then be able to react on an event fired by the parent component. Instead of directly invoking an interface controller method of one or several child components of the same type, the parent component would just fire an event.

parameter searchCriteria is passed to the event handler so that the controller logic can search for all matching purchase orders.

Component interface context mapping pattern

This componentization pattern requires the component separation and the component usage declaration pattern to be applied first.

Problem: How can a parent component access context data stored in an embedded child component?

Motivation: Web Dynpro uses contexts as hierarchically structured storage places inside a controller. Contexts are located across several controllers and components. You want to avoid passing multiple objects, object lists, values, or nested data structures across component borders based on method invocation or server side eventing, because it implies additional and often complex controller coding to transfer and receive them. You also want to avoid copying context data from one context to another via method call (based on value semantics) to optimize performance by reducing memory consumption on the server side. Store context data once and share it across component borders via context mapping (based on reference semantics) because it yields an automated data transport performed by the Web Dynpro Java Runtime environment.

Solution: Expose context data to other components in the component interface controller context. Other components can then reference this context data after having defined a context-mapping relationship to the interface controller context of the used component.

Declarations:

- Child component: In the child component's interface context, define the context elements that are mapped to the data context inside the child component (in component or custom controller context). Do not define a data context (un-mapped context storing the original context data) in the interface controller context itself (see the "Component controller delegation pattern" section for more details).
- Parent component: Declare a component usage relation to the child component inside the parent component. Open the Data Modeler dialog via the Open Data Modeler context menu item inside the Web Dynpro Explorer. Inside the Data Modeler, draw a data link from a parent component's controller to the interface controller of the used child component. Then you can define a context-mapping relation to the interface controller context of the used child component via drag and drop. The mapped context can then be used to map other contexts to it or to bind UI elements to it, as shown in **Figure 5**.

Implementation:
Based on the declared context-mapping relation to the child component's interface controller context, you can programmatically access its context as if it were defined inside the parent component. Note that context-mapping-based data access is based on a reference, not on a value semantic. This means that you access the data context stored inside the child component via reference, not via copied value.

Example: In the Web Dynpro case study application, the component PoSelectComp uses the child component TripSelComp as a search form for entering purchase order search criteria. The component PoSelectComp needs access to additional data (e.g., company codes or user roles) besides the searchCriteria string retrieved via interface controller eventing stored inside the child component TripSelComp.

To easily reference this context data inside the POSelect component controller, you can map elements in the component controller context to the interface context elements (nodes and attributes) of the used TripSelComp interface controller context as shown in **Figure 6**. You can then access the interface context of the used child component

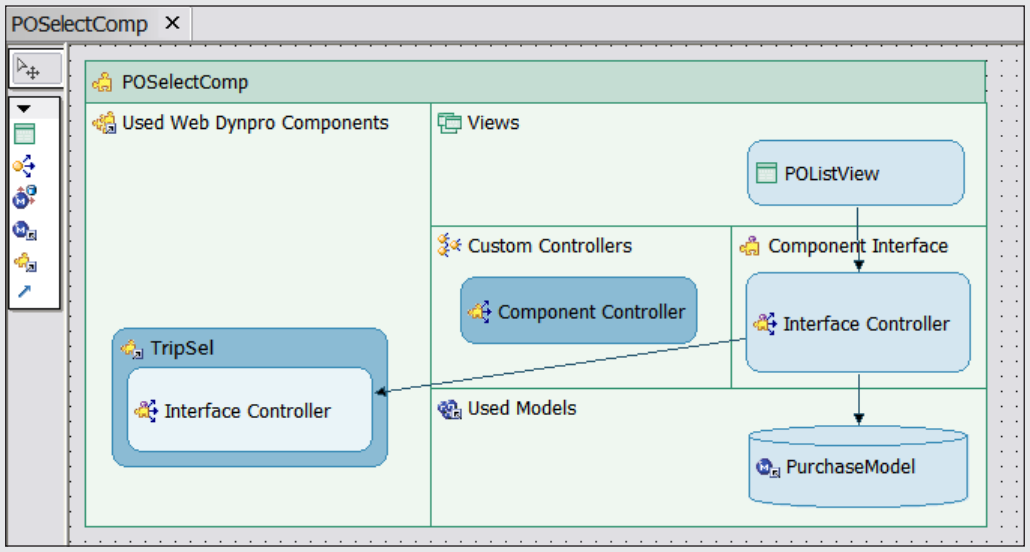


Figure 5 Data Modeler displays interface context mapping as data link

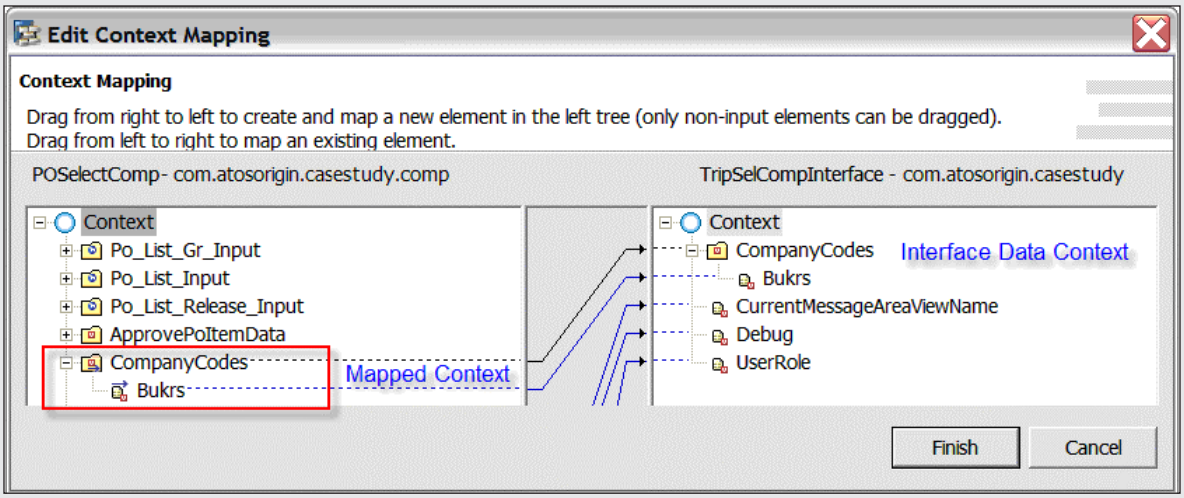


Figure 6 Edit Context Mapping dialog for defining the relationship between POSelectComp and TripSelCompInterface

TripSelComp by invoking typed context APIs[9] in the POSelect component controller:

```
ICompanyCodesElement elem =
  wdContext.nodeCompanyCodes().
  getCompanyCodesElementAt(0);
```

External component interface context mapping pattern

This componentization pattern requires that the component separation and the component usage declaration pattern be applied first. We did not apply this pattern in the Web Dynpro case study application.

Problem: How can a child component access context data stored in its embedding parent component?

Motivation: A child component needs to access external context data stored inside its parent component. As a child component must not and cannot define a cyclic component usage relation to its parent component, it cannot map its own interface context to the parent component's external data context.

Solution: Apply the external interface context mapping technique. Mark interface context elements of a child component as *input elements* first. To get the same context structure, copy the required structure of the parent's data context, and then paste it to the child's interface context. Its parent component then externally maps the interface controller context's input elements (e.g., marked nodes and attributes) to its own external data context (e.g., inside the component controller). In this way, the context mapping chain is completely defined.

Example: You can find a practical example of external interface context in sample application "Server-Side Eventing"[10] in the Web Dynpro Java tutorial on SDN.

Component interface view embedding pattern

This componentization pattern requires the component separation and the component usage declaration pattern to be applied first.

Problem: How do you visually embed the user interface of a child component inside its parent component?

Motivation: You want to be able to reuse user interface parts implemented once by a visual Web Dynpro component inside other components. You need to design nested user interfaces based on component separation. Based on the black-box principle described in the second article, you cannot directly embed a view of a child component into a view of its parent component because inner component parts, like views, are not directly visible outside.

Example: The Web Dynpro case study application realizes a multicomponent-based application user interface by nesting the user interface of search component TripSelComp into the purchase order selection component POSelectComp, which is in turn nested or embedded into all three root component user interfaces. This nested embedding of component interface views, implemented in two separate Web Dynpro components, is shown in **Figure 7**.

Solution: Embed the component interface view exposed and implemented by a used Web Dynpro component into a window, view area, or ViewContainer UI element of the parent component.

Every window defined inside a Web Dynpro child component gets exposed to a parent component with

[9] Typed context APIs are context interfaces that are generated for the declared context nodes and attributes. For every declared context node <*node name*> a corresponding pair of typed context APIs, I<*node name*>Element and I<*node name*>Node, gets generated. Return types and parameter types of typed context API methods are not the generic Web Dynpro context APIs (IWDNode and IWDNodeElement), but the typed context APIs themselves.

[10] You can download this Web Dynpro sample application from SDN on the Web Dynpro Java Sample Applications and Tutorials for SAP NetWeaver 2004 Web page. Login credentials are needed for access to SDN.

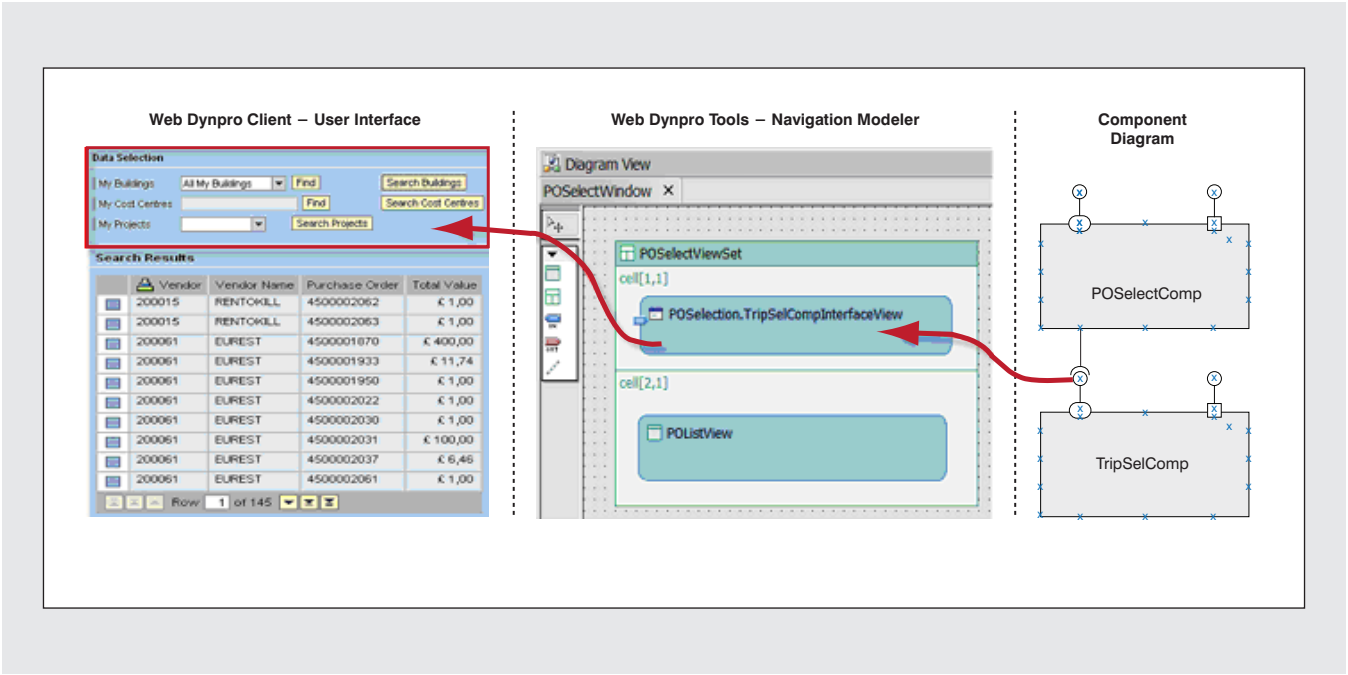


Figure 7 Embedding the component interface view of component TripSelComp

a unique component interface view.[11] The parent component can embed this component interface view like any other view, so the parent component acts as a "component interface view embedder" or simply as an embedder. In this way the child component's interface view is part of the navigation schema of the embedder. The embedder can navigate to a component interface view via an inbound plug (and trigger the associated event handler in the component interface view controller). The embedded component can cause navigation to the embedder by triggering an outbound plug of its interface view (cross-component navigation[12]).

Declaration:

- Child component: Every window (i.e., embedding views or other component interface views) inside a Web Dynpro component is automatically exposed to other components as a related component interface view.[13] To declare navigations links to or from an embedded interface view inside the parent component, define inbound or outbound plugs on the component interface view level.
- Parent component: Declare a component usage relation to the visual child component inside the parent component. Open the Navigation Modeler dialog via the Open Navigation Modeler option on the context menu inside the Web Dynpro Explorer. Inside the navigation modeler, embed a component interface view of the used child component (**Figure 7**). (For more information on the component interface view, see the sidebar on page 100.) You can then set the initial visibility of this component interface view

[11] In SAP NetWeaver 2004 and 7.0, there is a one-to-one relation between a component interface view and a corresponding window inside the component.

[12] See Web Dynpro Java tutorial and sample application "How to Navigate Inside Web Dynpro Component Interface Views" on SDN to get more details on this topic.

[13] In SAP NetWeaver CE 7.1, the Web Dynpro component model no longer exposes windows as interface views by default. Instead a window must explicitly implement one or more component interface views defined in a local or in a standalone component interface definition to be visible for other components outside.

with the Boolean property of default, or you can define navigation links to and from inbound and outbound plugs of the embedded component interface view.

Implementation:
To make a Web Dynpro component interface view visible on the user interface programmatically at runtime (i.e., based on controller code), make sure that the following two alternatives exist[14]:

- Component interface view navigation
- Component instance creation and destruction[15]

To navigate to a component interface view, first define a navigation link to an inbound plug of the used component interface view (navigation target). Fire the outbound plug in the view controller of the navigation source (`wdThis.wdFirePlug<outbound plug name>()`) to display the component interface view on the user interface. In case the navigation target component instance does not yet exist, the Web Dynpro Java Runtime creates it automatically if the related component usage is declared with the Lifecycle property set to createOnDemand.

When a visual component is used with the component usage Lifecycle property is set to manual, the Web Dynpro Runtime will not automatically create or display it when its component interface view is reached via navigation link. Also, it will not destroy this component when the view has disappeared from the UI via navigation. In this case, you have to explicitly create the visual component (invoking the IWDComponentUsage API) to make an interface view appear, or you have to destroy the component instance so the interface view disappears. This second alternative is recommended for scenarios in which navigation away from a displayed component interface view (no longer displayed afterwards) should be combined with the explicit destruction of the associated component instance. Otherwise, the Web Dynpro Java Runtime won't explicitly delete the embedded component instance.

Component controller delegation pattern

We recommend (it is not mandatory) that you apply this componentization pattern in all SAP NetWeaver 2004 and SAP NetWeaver 7.0 Web Dynpro Java applications so that you can migrate your applications easily to the enhanced Web Dynpro component model in SAP NetWeaver CE 7.1.

Problem: How do you implement the component interface controller class?

Motivation: The component interface controller class no longer exists in the new Web Dynpro component model introduced in SAP NetWeaver CE 7.1. Instead, component interface controller will be implemented by the component controller class. You want to avoid moving code from the component interface controller class to the component controller in SAP NetWeaver CE 7.1. You should also avoid refactoring controller usage dependencies to the component interface controller, which will no longer be provided in the enhanced component model.

Solution: Implement and declare the component interface controller as a pure delegator of interface functions (such as methods, events, or context-based data transfer) between an external user of the component interface controller (a controller inside the parent component) and its implementation inside the child component using the following guidelines:

- For every interface controller event, declare a public method inside the component controller to fire this event. In this method, you should fire the interface controller event by accessing the interface controller's IPublic API.
- Copy all of the methods declared in the component interface controller to the component controller and implement them there. Delegate to these methods inside the component interface

[14] A component interface view is displayed by default with its embedding component when the component interface view usage (defined in a window using the Web Dynpro Tool Navigation Modeler) has defined the property default equals true. In this case, you need not implement controller code to display the component interface view on the UI.

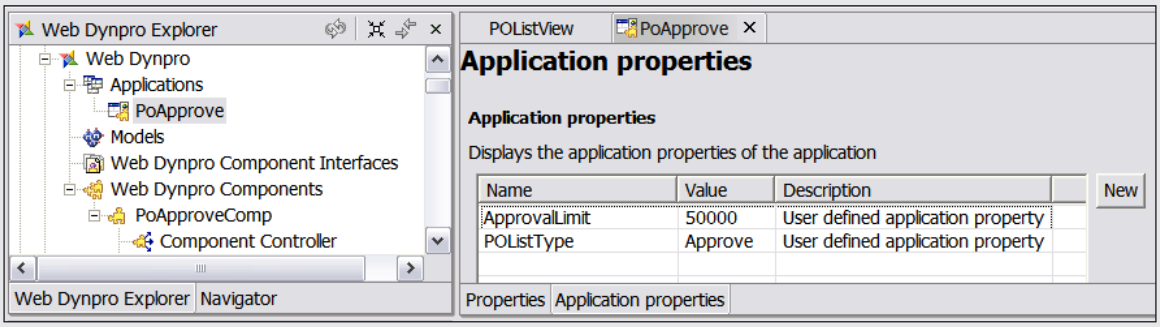
[15] See the "Differentiating a Web Dynpro component usage from its associated component instance" and "Understanding the Lifecycle property of a component usage" sections in the second article of this series.

How reusable components can adapt their flavor

You might have already wondered how the two Web Dynpro components, TripSelComp (search component) and POSelectComp (purchase order selection component), adapt their flavor — that is, specialization or adaptation of a component to the context in which it is running — to the root component (PoGRComp, PoApproveComp, or PoChangeComp) in which they are actually running.

In the Web Dynpro case study application, we met this requirement by defining application properties on the Web Dynpro application entity level and accessing them within the reused child components. We made sure that all three root components started with three separately defined Web Dynpro applications.

For a Web Dynpro application entity, you can set pre-defined application properties (such as authentication or LogoffULR) and self-defined application properties (such as a property to identify the application type). For example, the PoApprove application starting root component PoApproveComp defines the two custom application properties — ApprovalLimit and POListType as shown in the screenshot below.



The search component TripSelComp can then access this application property by invoking the generic IWDApplication API to adapt its flavor (e.g., display or hide fields in view layout):

```
IWDApplicationPropertyInfo appPropertyInfo =
  wdComponentAPI.getApplication().getApplicationInfo()
    .findInApplicationProeprties("POListType");
if ("Approve".equals(appPropertyInfo.getValue()) {...} else {...}
```

As an alternative, a root component can pass such configuration data to its embedded child component by means of external context mapping — that is, the parent component writes configuration data into its own context and the child component accesses the mapped context in its `wdDoInit()` controller hook method. You can also initialize child components by invoking a component interface controller method from the embedding parent component.

controller after defining a component controller usage relation.

- Do not declare data contexts inside the interface controller context. Instead, map the interface context to the component controller's data context.
- Do not declare any direct controller usage relation to the component interface controller in a view controller or in a custom controller (bypassing the component controller) to invoke interface controller methods, fire interface controller events, or access the interface controller context via context-mapping. Instead, access the component interface controller from other controllers in the same component using the component controller as delegator.

Component usage referencing pattern

This componentization pattern requires that the component separation and the component usage declaration pattern be applied first. We did not apply this pattern in the Web Dynpro case study application. Nevertheless, this pattern provides a solution for sharing a single instance of the component in a componentized Web Dynpro Java business application.

> ***Note!***
>
> In SAP NetWeaver 2004 and 7.0, you can apply the component usage referencing pattern only to *faceless* Web Dynpro components, which do not implement user interfaces. You cannot apply this pattern to visual Web Dynpro components. In SAP NetWeaver CE 7.1, you will be able to reference a component usage that points to a visual component instance. With this technique, you can display a child component in a component user interface without being responsible for its life cycle management. This is particularly useful in combination with the component interface definition strategy pattern for loosely coupling parent with child components.

Problem: How can different components use the same instance of another component?

Motivation: You have multiple Web Dynpro parent components that use the same child component via their own independent component usage relations. These component usages have a default Lifecycle property of createOnDemand, which implies that all parent components separately create their own independent child component instances. You want to be able to release a component from managing the life cycle of its used child component individually, so that other components (e.g., "sibling" components) can share the same child component instance. Therefore, you want to centrally manage the life cycle of a single component instance in a higher-level component (e.g., the root or application component) and share this component with other child components.

Example: A typical use for this pattern is a central, faceless Web Dynpro model component (connecting to business logic) in mostly visual Web Dynpro components. The other components must all use the same model component instance at runtime. See an example of this pattern in the "Faceless model component pattern" section on page XX.

Solution: Enter the model component usage managed by the root component in the referencing mode within both child components. The child components then jointly reference the same model component instance to which the referenced model component usage points.

Declarations:

- Child component: In child components that need to share the same component instance, declare a component usage relation to the jointly used faceless component. Explicitly set the Lifecycle property for all of these component usages to manual. Also, declare an interface controller method

```
referenceComponentUsage
    (IWDComponentUsage sharedUsage)
```

and implement the method in the component

controller. For this, first define a usage relation to the faceless component usage (not to its component interface controller) inside the component controller. (See the sidebar on page 100 for more information on component usage.)

- Parent component*:* In the parent component, declare a component usage relation to the jointly used faceless component. Explicitly set the Lifecycle property of this component usage to manual (not mandatory, but recommended). Declare controller usage relations to all interface controllers of the other used child components.

Implementations:

- Parent component: Create an instance of the faceless child component by invoking the IWDComponentUsage API. Pass the reference from the faceless child component usage to all other child components by invoking the defined interface controller method `referenceComponentUsage (IWDComponentUsage sharedUsage)`.
- Child component: Implement this method in all child components' interface controllers by delegating the method to the component controller. In the component controller, enter the passed component usage — which that points to the single faceless component instance — in referencing mode with the following line of code:

```
wdThis.wdGet<faceless component usage
  name>ComponentUsage()
  .enterReferencingMode(sharedUsage);
```

Afterwards, the child component refers to the component usage maintained by its parent component and therefore points to the single instance of the jointly used faceless component.

Note!

When the parent component destroys the associated component instance by calling a corresponding method in the child components' interface controller, the parent component should inform all child components that reference the shared faceless component. The child components must then leave the component usage referencing mode that they entered to avoid access to a non-existing instance of the faceless component afterwards:

```
wdThis
  .wdGet<faceless component usage name>
  ComponentUsage().leaveReferencingMode()
```

Faceless model component pattern

This componentization pattern[16] requires that the component separation and the component usage declaration patterns be applied first and combined with the component usage referencing pattern. We did not apply a faceless model componentization pattern in the Web Dynpro case study application.

Problem: How can you connect to the business logic inside a central faceless Web Dynpro model component?

Motivation: A visual Web Dynpro UI component should be released from directly accessing a Web Dynpro model itself (SoC). Instead, it should access model data via interface-context-mapping. You should define the context-to-model binding and implement access to the business logic (e.g., executing model objects, invalidating response context nodes) in a faceless Web Dynpro model component.

Example: You have an application in which different user interface components need access to the interface context of the same model component instance. In addition, the user interface components are sibling components and are therefore not related to one

[16] For more details on this topic, see Web Dynpro Java tutorial and sample application "Designing Component-Based Web Dynpro Applications" on SDN.

another. The life cycle of the single model component instance is centrally managed by the root component, which uses both sibling components as child components. To retrieve model data, both sibling components, using the same model component, map their contexts to the interface-context of the model component at design time. They both enter the central model component usage (managed by the root component) in referencing mode at runtime, and then invoke business methods (e.g., `executeGetList_PoDetails()`) exposed in the model component's interface controller to update the referenced interface-context. By entering the same model component usage in referencing mode in both sibling components, they can share the same model data via context-mapping.

Solution: Implement a faceless Web Dynpro model component and expose model data to other components in interface-controller-context for interface-context-mapping. Enter the model component usage (defined in root component) in referencing mode within visual Web Dynpro UI components.

Declarations:

- Faceless model component: Declare a model usage and bind the component or custom context to the model classes. For executable model classes, expose corresponding execute methods in the component interface controller to be invoked by other components consuming model data.
- Child components that refer to the model component usage: Apply declarations described in the "Component usage referencing pattern" section on page 110. Define a usage relationship to the Web Dynpro Model to which the faceless model component is bound.[17] Map context(s) inside the child component to the interface-context of the used model component. Declare a usage relation to the model component's interface-controller to invoke methods in controller code. (For more information on component

[17] A model usage must be defined to declare a valid interface-context-mapping-relation to the model component's interface context. Otherwise, the model classes (part of the interface-context-mapping definition) are not visible.

Note!

We did not apply the faceless Web Dynpro model component pattern in the Web Dynpro case study application, in which we defined and implemented the context-to-model binding and the model class execution directly in the different UI components. Consequently, the case study application combines model access and user interface display in separate UI components. Model data that was retrieved in a UI child component gets exposed to higher level UI parent components via the UI component's interface context.

This approach is feasible in a nested, "vertical" Web Dynpro component hierarchy where no sibling UI components on the same component hierarchy-level are given and where the model data retrieved in a child component must only be accessed by higher level parent components. As soon as multiple sibling UI components are used by the same parent component, sharing model data between these sibling components becomes more difficult.

The best solution for sharing the same model data across independent Web Dynpro UI components (e.g., for two child components of the same parent component) is the faceless Web Dynpro model component pattern combined with the component usage referencing pattern.

You can find more detailed technical information on how to apply the faceless Web Dynpro model component pattern with the component usage referencing pattern in Web Dynpro Java sample applications and tutorials for SAP NetWeaver 2004 "Designing Component-Based Web Dynpro Applications" on SDN.

interface controller usage, see the sidebar on page 100.)

- In parent component managing the model component usage: Apply declarations described in the "Component usage referencing pattern" section.

Implementation:

- Faceless model component: Implement controller logic to bind the context to the model at runtime. Implement `execute()` methods exposed by the component-interface-controller inside the component controller (see the "Component controller delegation pattern" section).
- Child components that refer to the model component usage: Follow the implementation description in the "Component usage referencing pattern" section. Define a usage relation to the Web Dynpro Model to which the faceless model component is bound. Map context(s) inside the child component to the interface-context of the used model component. Declare usage relation to the model component's interface-controller to invoke methods in controller code.
- Parent component that manages the model component usage: Follow the implementation description in the "Component usage referencing pattern" section.

Component interface definition strategy pattern

We did not apply this pattern in the Web Dynpro case study application. Nevertheless, it provides a solution for loosely coupling a Web Dynpro parent component with its child component without knowing the child component implementation(s) at design time.[18]

Problem: How can a Web Dynpro component use another component without knowing the other component's implementation at design time?

Motivation: In some scenarios, you need to find an easy replacement for one Web Dynpro child component implementation with another at runtime. "Easy replacement" means that the used component implementation can be replaced via configuration but without code modification, or without being rebuilt and redeployed. For example, a customer wants to have the possibility to replace your shipped default component implementation with another custom component implementation developed by the customer. A custom component implementation is needed when the requested component adaptations cannot be handled by configuring or personalizing the default implementation. For example, hiding certain form fields can be configured, but extensively changing the component's user interface layout cannot be configured.

Solution: At design time, loosely couple a parent component with its child component using a component usage that points to a Web Dynpro component interface definition, which has no implementation inside. At runtime, programmatically create a component instance implementing this component interface definition.

Declarations:

- Parent component: Declare a component usage relation pointing to a Web Dynpro component interface definition (implies that the Lifecycle property is set to manual) instead of a specific Web Dynpro component implementation. Use this interface definition in the parent component to define all dependencies to the child component (context-mapping, interface view embedding, navigation links, event subscription, method invocation). Define a usage relation to the component usage (pointing to a component interface definition) in component or custom controller.
- Child component: Declare an implementation relation to the component interface definition inside the child component implementation (in Web Dynpro Explorer, follow the menu

[18] For more details on this topic, see the "Web Dynpro Component Interface Definitions in Practice" on SDN.

path <*component node*> → Component Interface → Implemented Interfaces to specify the declaration).

Implementations:

- Child component: Implement the component interface definition in one or more Web Dynpro components.
- Parent component*:* In component or custom controller, read the fully qualified name of the component instance to be created at runtime from a database or from a configuration file. Create a concrete child component instance that implements the used component interface definition by passing the fully qualified name of the implementing child component with method `IWDComponentUsage.createComponent (componentName, deployableObjectName)`.

Invoke interface methods and access the interface context via the IExternal API of the used component interface definition's interface controller.

Web Dynpro development componentization patterns

In this section, we move from the Web Dynpro component-related patterns to the Web Dynpro development component-related ones. We provide best practices and guidelines for optimizing the development component granularity of your Web Dynpro Java business application and for effectively separating all comprised development entities, (such as different types of Web Dynpro components, component interface definitions, models or dictionaries) into multiple development components. Finally we provide some practical tips and tricks to help you work more efficiently with Web Dynpro development components inside the NWDI.

The described Web Dynpro development componentization patterns and development recommendations can yield an efficient and team-oriented development process based on an optimized granularity of build units,[19] as well as better maintainability and understandability.

Web Dynpro development component separation patterns

There are Web Dynpro development componentization patterns that you can use to separate Web Dynpro Java business applications into Web Dynpro development components. There are patterns for separating certain types of Web Dynpro entities, such as models, component interface definitions, local dictionaries, and reusable Web Dynpro components. There are also patterns used to separate other Web Dynpro components, including the root or application components, UI components, and faceless model component. Applying these patterns yields an optimized development component granularity required for team-based application development, rapid build cycles, better maintainability, and easy understandability.

Problem: How can you find the best development component granularity?

Motivation: Several developers have to jointly implement a large-scale Web Dynpro Java application. You want to minimize the turnaround time required for completing build, deployment, and run cycles during the development process. One of the first tasks is to split this Web Dynpro application into separately versionable deploy units.

Solution: First, separate special Web Dynpro entities into different Web Dynpro development components. Then combine functionally related development objects into one reusable group.

Keep in mind that development components are the build units of the Central Build Service (CBS), so the deployment granularity (number and size of development components) plays a decisive role when optimizing the performance of a single build-deploy-run cycle. You want to optimize the granularity by applying some simple separation rules

[19] Every Web Dynpro development component is one single build and deployment unit.

on how to move different Web Dynpro entities to the NWDI component model.

Separating Web Dynpro entities

You can separate Web Dynpro models, component interface definitions, local dictionaries, and reusable Web Dynpro components into Web Dynpro components.

Separating models

Separate models into another development component and expose them as public parts. Web Dynpro models should be separated into individual development components (one model per development component). These models can be referenced in other Web Dynpro components based on a defined development component usage.

Typically, models contain classes that do not change during development, especially if the interfaces have been defined thoroughly. Since it is optional to build dependant components during design and build time, the development component that holds the models can be excluded from the build, thereby reducing build time as well as deploy time.

Since different development components are not allowed to share the same namespace, placing each model in a different development component enforces the following rule: Put each model in its own namespace or Java package.

In **Figure 8** (on the next page) you see the component diagram for the Purchase Order Approve application. The adaptive RFC model entity in development component PoModel is separated from the other Web Dynpro entities building the Purchase Order Approve application (①).

Separating component interface definitions

Using Web Dynpro component interface definitions separates the API from the implementation at the Web Dynpro component level. By combining component interface definitions within a central Web Dynpro development component — independent from the related implementing Web Dynpro components within other development components — you achieve a separation of these component interface definitions from their component implementations on the NWDI component model and therefore on software life cycle and deployment level. You want to spend a sufficient amount of time planning the component interface definition so changes and re-declarations can be avoided. We did not apply Web Dynpro component interface definitions in the Web Dynpro case study application.

Separating local dictionaries

You should centrally declare all required local dictionary types[20] within a separate Web Dynpro development component. For example, you will need a local dictionary simple type in a Web Dynpro application to populate a DropDownByKey UI element with a value set, which is not defined in a logical dictionary of an imported adaptive RFC model.

You can reuse local dictionary types that are stored in a separate Web Dynpro development component within all other Web Dynpro development components. You only have to make changes within the definition of dictionary types once and all objects using these types are automatically adapted (the advantage is even greater if the type can be used across several applications). By avoiding a redundant definition of the same dictionary types within several Web Dynpro development components, the initialization process speeds up at runtime.

Separating reusable Web Dynpro components

To optimize the reuse potential of a generic (and therefore potentially large) Web Dynpro component, you should separate its functionalities into an individual Web Dynpro development component. Using this approach, your reusable Web Dynpro components

[20] Local dictionary types are not defined in an ABAP Dictionary on the back-end side so they are not reflected in a logical dictionary, which comprises the simple types and structure types within an imported adaptive RFC model.

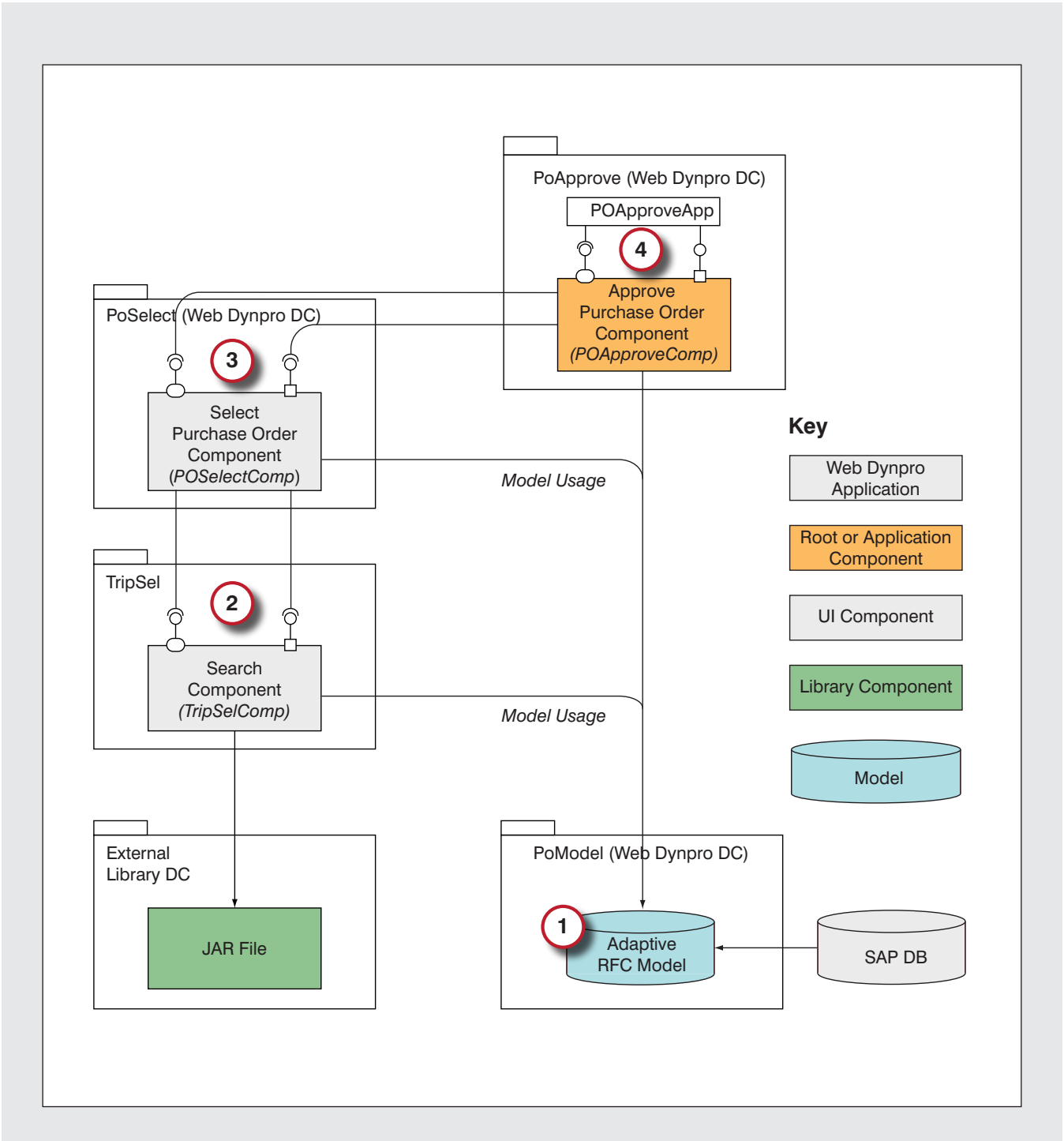


Figure 8 Web Dynpro development component architecture diagram for the Purchase Order Approve application

will reside in a separate build unit that is independent from its consumers.

In the Purchase Order Approve application, we applied this development component separation

principle to the two reusable components: TripSelComp for selecting sites and POSelectComp[21] for selecting purchase orders. These components are stored in two different Web Dynpro development components: TripSel (**Figure 8**, ②) and PoSelect (**Figure 8**, ③).

Separating specialized Web Dynpro components

In the Web Dynpro case study application, we separated the models, component interface definitions, local dictionaries, and the reusable components into different Web Dynpro development components. The following guidelines can help you determine the best size and number of the Web Dynpro development components that comprise the remaining specialized Web Dynpro components, such as root or application components, UI components, and faceless Web Dynpro model components.

You should apply the Web Dynpro development component separation strategy when partitioning these Web Dynpro components on the NWDI development component level. This approach yields many advantages, such as shorter build and deployment times for small development components. Conversely, it increases the total generation and build time for the complete application because it is inevitably combined with an additional overhead for defining usage-relations of entities across development component borders.

Nevertheless, the advantages of separating most Web Dynpro entities within different Web Dynpro development components outweigh the possible disadvantages in most cases. This is especially true within a team-based development scenario because a partitioning strategy favors small development components, which can efficiently enhance the development performance.

[21] The component POSelectComp is reused by additional application components POChangeComp (purchase order change component) and POGoodReceiptsComp (purchase order goods receipt component).

Separate Web Dynpro root or application components

Embed the Web Dynpro root component into a separate Web Dynpro development component. The root component can be loosely coupled with its contained visual UI components by using component interface definitions instead of implementing Web Dynpro components. With this approach, you can eliminate an inevitable rebuild of the embedded visual UI components when rebuilding the Web Dynpro development component comprised of the root or application component. Remember that the development component build process is still nonincremental, so all comprised and even unchanged Java classes, interfaces, and other resources get regenerated with every development component build process.

In the Purchase Order Approve application, you applied this development component separation principle to the three root or application components: POApproveComp (purchase order approval component, **Figure 8**, ④), POChangeComp (purchase order change component), and POGoodReceiptsComp (purchase order goods receipt component).

Separate the Web Dynpro UI components

You should separate all visual UI components into at least one single Web Dynpro development component. When developing more complex user interfaces with many visual UI components, you should separate all of these visual components into many different Web Dynpro development components while keeping all related UI components in the same development component. For example, a UI component that uses another UI component as pop-up window should reside in the same development component.

Separate faceless Web Dynpro model components

You can apply the previous rule to the faceless Web Dynpro model components. A Web Dynpro model component centrally implements the back-end data

access, but it does not implement any user interface. Within its component interface controller, the Web Dynpro model component exposes the accessed back-end data (as interface controller context) and business functions (as interface controller methods). Other components can then refer to this data by defining a context-mapping relation, and can further invoke the interface controller methods to access the business logic.

There is one major reason for not combining Web Dynpro models and Web Dynpro model components within the same Web Dynpro development component. A model comprises a set of model classes (and in the case of an Adaptive RFC model, a logical dictionary with several simple type and structure type definitions), which is not completely required by a single model component. For example, you could have a model that contains ten executable model classes, but your model component might only use two of them. In this case, combining model and model component within the same development component would unnecessarily increase the required build time.

Web Dynpro development component recommendations

Here are some practical rules and recommendations to develop and build Web Dynpro development components more effectively and rapidly.

- **Optimize public part definitions:** You need to define objects released for use by other (Web Dynpro) development components as public parts. Make public parts as specific and small as possible, including the entire content of the development component that is still being changed into a public part. The automatic rebuild of used development components is necessary only if your change uses a public part that refers to the used development component. You can avoid unnecessary rebuilds by publishing objects that will probably be used by different development components for different public parts. Every development component can contain several public parts — even of the same type.

  Another reason for creating small and specific public parts is to limit rebuilds. The fewer places the public parts are used, the fewer the rebuilds that will occur. Whenever possible, refer directly to public parts and not to their development components.

- **Create inner development components for structuring large development components:** Generally, there is no restriction on the size of a Web Dynpro development component. Since development components represent the units for the build process, development component size considerations can speed up the development process. Functionality that (almost) always belongs together should be provided as one development component or public part so that it can be referenced with only one dependency.

  After a change, the entire development component must be rebuilt. Therefore, choose the size carefully so that it does not contain too many objects. If a development component becomes big over time, you can split it internally into smaller child development components and provide the same functionality and granularity by the old (now parent) development component for the outside world, while having smaller units of compilation and editing.

  As a rule of thumb, development components can reach a size where up to four developers are responsible for Web Dynpro entities.

- **Apply some practical rules when working with Web Dynpro development components:** You should apply the following rules when building Web Dynpro development components.

  - Be sure there is a connection to DTR, even if the complete component is checked out. The reason is that on save, a deleted object is moved from “checked out for edit” to “checked out for delete.”

- Do not check out Web Dynpro objects that are checked out by other users.
- Do not keep bottle neck resources checked out (e.g., the complete Web Dynpro component).
- Always use the same activity for objects in the same Web Dynpro development component.
- Save your changes before performing repository operations (check in, sync, etc.) in the development configurations perspective.
- Click on Reload in Web Dynpro Explorer after performing a repository operation in the Development Configurations perspective that adds/deletes/changes files (e.g., sync).
- Rebuild a development component as soon as its public part definition has changed. Be sure to rebuild a Web Dynpro development component via the context menu option "Rebuild Project" (this does not imply the rebuild of the public part archive). This archive is only created within a development component build, which is started via the context menu entry Development Component Build.
- Perform the Deploy New Archive and Run step for changed development components.

Finally, we have some guidelines for naming Web Dynpro components that you download from the *SAP Professional Journal* Web site at http://www.sappro.com/downloads.cfm. Using a consistent naming convention will help keep your development projects organized and sharing them a smooth and straightforward process.

Conclusion

In the third part of this series, we focused on the practical aspects of designing, implementing, and packaging a component-based architecture in Web Dynpro Java using the NWDI.

By introducing a set of 11 Web Dynpro componentization patterns, we captured proven and mature solutions for recurrent problems in the Web Dynpro component context. We started with the component separation and usage declaration patterns, which must both be applied first to leverage component-based architecture in Web Dynpro Java business applications. We then described a set of patterns dealing with interface-controller method invocation, cross-component eventing, and cross-component context-mapping. To ease a future component migration to the new SAP NetWeaver CE 7.1 component model, we then recommended applying the component controller delegation pattern in all of your Web Dynpro components developed in SAP NetWeaver 2004 or 7.0. Next, we explored the component interface view embedding pattern for creating nested, component-based user interfaces. With the next two patterns, component usage referencing pattern and faceless model component pattern, we described a solution for accessing model data provided by a single model component instance from different sibling components. We then completed the Web Dynpro componentization patterns section with the component interface definition strategy pattern, applied to loosely couple a Web Dynpro parent component with its child component implementation by using an abstract component interface definition at design time.

In the second part of this article, we advised a clear development component separation of different Web Dynpro components (UI components, faceless model components, component interface definitions) and non-component (models, dictionaries) entities. By applying these Web Dynpro development component separation patterns, you can optimize the Web Dynpro development component granularity required for effective team-based application development, rapid build cycles, better maintainability, and easy understandability.

Finally, we provided some practical rules and recommendations for building Web Dynpro development components more effectively and more rapidly by optimizing the public part, using child development components or accelerating the build process.

Bertram Ganz studied mathematics, physics, and computer science for the high school teaching profession at the Albert-Ludwigs-Universität Freiburg in Germany, where he completed his first state exam in 1997. In 2000, he received his teacher certification before working as technical trainer for Day Software, a provider of integrated content, portal, and digital asset management software in Basel, Switzerland. He joined SAP at the end of 2001 as a member of the Web Dynpro Java Runtime development team. Bertram's work focuses on technical knowledge transfer, training, roll-out, roll-in, documentation, and development consulting on the Web Dynpro for Java UI framework. Bertram regularly publishes blogs and articles on Web Dynpro for Java and is co-author of the books "Maximizing Web Dynpro for Java" (SAP PRESS, 2006) and "Java Programming with SAP NetWeaver" (SAP PRESS, 2008). He is currently working as the senior product specialist on the SAP NetWeaver UI Foundation Operations team. You may reach him at bertram.ganz@sap.com.

Richard Tucker joined Atos Origin as an ABAP consultant in 1998. He quickly recognized SAP Web development as an emerging market and was instrumental in building Atos Origin's SAP Web development capability. He has successfully implemented all SAP Web technologies for the last 10 years for numerous customers. Since 2005, he has concentrated on Web Dynpro implementations, with a long track record in building successful component-based applications. This track record was the basis for his well-received SAP TechEd 2007 presentation on behalf of SDN. You may reach him at richard.tucker@atosorigin.com.

Atos Origin is an international information technology services company with 50,000 employees in 40 countries. Its business is turning client vision into results through the application of consulting, systems integration, and managed operations. Atos Origin is the Worldwide Information Technology Partner for the Olympic Games and has a client base of international blue-chip companies across all sectors. Atos Origin is the 2007 Winner of the SAP Partner Excellence Award for Customer Satisfaction and Quality Performance for a third consecutive year.

