How to avoid Web Dynpro Java implementation nightmares

by Chris Whealy

Chris Whealy
RIG Expert, (Web Dynpro)
SAP NetWeaver Regional Implementation Group (RIG)
EMEA, Walldorf, Germany

Chris Whealy joined SAP in 1995 as a Basis consultant and ABAP programmer. Shortly thereafter, he turned his attention to Web-based interfaces into SAP and began working with the earliest versions of the Internet Transaction Server (ITS). Since then, Chris has focused his attention on Web-based front ends for SAP functionality. In 2003, Chris began working with Web Dynpro and has worked closely with the development team, both learning the product and writing proof-of-concept applications.

(Full bio appears on page 58.)

Web Dynpro is SAP's standard toolset for developing user interfaces (UIs) for business applications. It is a powerful development tool that allows you to construct efficient, low-maintenance business applications. SAP has designed Web Dynpro to be the standard UI development tool for all future applications, whether written in Java or ABAP.

Since 2005, I have performed many code reviews of Web Dynpro implementations at customer sites. The purpose of these reviews was both to assess the quality of the coding and to determine the cause of implementation difficulties. As I performed these reviews, I observed that quite independently, both customers and implementation partners were making the same mistakes over and over again, resulting in poor-quality Web Dynpro implementations and applications that were time consuming and costly (or in some cases, impossible) to maintain.

As I analyzed why these mistakes should be so widespread, I noticed a common thread running though all of the projects: the developers often had little or no specific Web Dynpro training. Consequently, they assumed Web Dynpro to be "just like all the other Web development toolsets" and tried to use it as if it were some other product with which they were already familiar. Unfortunately, using Web Dynpro with this mindset always produces poor-quality results — with some pretty ugly outcomes in some cases. In one instance, the implementation was so poor it had to be thrown away and rewritten!

With a correct understanding of Web Dynpro, you can easily avoid the difficulties I have observed and ensure a smooth implementation in your own organization. The purpose of this article is to correct the misunderstandings at the root of Web Dynpro implementation problems and to explain the principles of good Web Dynpro design so that you will be able to write high-quality applications that will, in turn, help reduce your maintenance and support costs.

The article is divided into three main sections. In the first section, I describe what can go wrong during a Web Dynpro implementation, why these mistakes occur, and how to avoid them. In the second section, I move on to describe the high-level design principles that you should follow during the design phase of a Web Dynpro implementation, which is where the overall architecture of the application is planned. Finally, in the third section, I describe the lower-level design principles that you should follow when building individual Web Dynpro components. While this article is aimed primarily at project managers, anyone involved in Web Dynpro implementations will benefit from reading it.

This article focuses on the Java version of Web Dynpro,[1] and has been written with the SAP NetWeaver Composition Environment (CE) version of Web Dynpro in mind. However, almost all the principles described here are applicable to the SAP NetWeaver '04 and SAP NetWeaver 7.0 (formerly 2004s) versions. Where the differences between CE and earlier releases affect the design principles, I will make a specific comment. Otherwise, you can assume that the design principles in this article are version independent.

Readers would probably also be interested to read a previous *SAP Professional Journal* article I have written entitled "Web Dynpro — what it is, what it does, why it exists, and how to get the best results from it: An introduction to the fundamental principles of Web Dynpro." This article focuses on the SAP NetWeaver '04 and 7.0 versions of Web Dynpro and appeared in the January/February 2007 issue.

Let's first tackle the main reasons why Web Dynpro implementations can end up in trouble.

Misunderstanding the purpose of Web Dynpro

To explain why the purpose of Web Dynpro is so often misunderstood, I must first explain two different things. The first is SAP's design criteria for Web Dynpro, and the second are the expectations people from a Web development background tend to bring with them when using Web Dynpro for the first time. Once these two things are explained, you'll see that the misunderstandings are caused by a mismatch between SAP's design criteria and people's expectations.

SAP's design criteria for Web Dynpro

To be somewhat pedantic, Web Dynpro is not really a "Web" development toolset — although it is frequently used as if this is its only capability. In reality, Web Dynpro is a toolset for building business applications that have a medium- to long-term life span and are aimed at a generic audience. This means that there is no strict requirement that the client device be a browser running on a desktop computer. It could be a mobile device such as a Pocket PC, a Nokia Communicator, or a barcode/RFID scanner. This immediately places Web Dynpro outside the realms of "traditional" Web development toolsets.

Back in late 2001, when the need for a suitable application development toolset emerged, SAP laid down a definitive set of criteria that this new product, whatever it was, would have to fulfill. SAP evaluated various commercial and open source products for suitability, but none met all of the design criteria. Therefore it was decided that an internally developed product was required. The criteria that had to be fulfilled are the following:

1. Create a UI programming paradigm that would become the de facto standard for all future SAP software.
2. Eliminate the repetitive coding tasks currently experienced by Web developers. Most importantly, the fewer lines of handwritten code there are in the UI, the better.
3. Create a fundamental unit of software reuse that exists at a business level rather than at some lower, technical level.
4. Use a declarative approach to application design.

[1] For more on the ABAP version of Web Dynpro, see the article "Get started developing Web-native custom SAP applications with Web Dynpro for ABAP" (*SAP Professional Journal*, July/August 2007).

5. Make full use of abstract modeling. Web Dynpro applications should not need to care about:
    a. The communication technology required to access a back-end system
    b. The client technology being used to render the screens
6. Make full use of generic services. Functionality that is frequently required should be made available from a standard library of services.

The first three points are the most important.

The overarching goal of Web Dynpro is summarized in the first point — SAP wanted to get away from multiple UI development tools and rationalize development down to a single UI toolset.

The second point aims to tackle the time-consuming and repetitive task of writing UI coding. Anyone who has been involved in a Web development project will be fully aware that the largest proportion of the implementation timescale is spent fiddling around with HTML, JavaScript, and Cascading Style Sheets (CSS).

Even now, Web developers still feel the pain of having to write and rewrite the same type of code every time they create a new business application. Even Google states on its Web Toolkit page[2] that

> *Writing dynamic web applications today is a tedious and error-prone process; you spend 90% of your time working around subtle incompatibilities between web browsers and platforms, and JavaScript's lack of modularity makes sharing, testing, and reusing AJAX components difficult and fragile.*

Therefore, any toolset that can remove the tedious and error-prone nature of Web development will provide developers with a major boost in speed and efficiency. The Web Dynpro tools within the SAP NetWeaver Developer Studio (NWDS) do exactly this by generating the UI layer coding for you.

In addition to wanting to remove the need for developers to write any browser-specific coding, SAP also wanted Web Dynpro applications to be not just client independent, but client *device* independent. This criterion means that when a Web Dynpro screen is being developed, it is not important to know exactly which type of device will act as the client.[3] This means that all Web Dynpro screens have to be defined in a client-independent manner. However, the consequences of this feature are poorly appreciated, and it is this lack of awareness that has led to many of the Web Dynpro implementation problems I have observed.

The third point comes from the fact that the software architects at SAP have the benefit of some 30 years experience in the ERP software market. This experience gave them the foresight to see that if a software developer's attention is not realigned to focus on the business process, then they will be forever stuck down at the nuts-and-bolts level. So SAP stated that the fundamental unit of software reuse within Web Dynpro development should be made equal to a discrete step of the business process.

Taken together, all these requirements combine to produce a Web development toolset that is unlike any other on the market. When people with no Web Dynpro experience use this toolset for the first time, they naturally bring with them all their previous experiences and expectations — and herein lies the problem.

People's expectations of a "Web development toolset"

Anyone who has used products that fall into the broad category of "Web development toolset" will have a set of expectations based on their use of those products. These people then, often unconsciously, expect Web Dynpro to operate and behave in the same way as the other products with which they are already familiar.

The problem is that Web Dynpro cannot be used as if it is Struts, Spring MVC, Tapestry, or Ruby on Rails.

[2] From http://code.google.com/webtoolkit/ (accessed November 19, 2007).

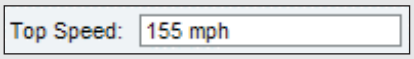
[3] The only restrictions here are that mobile devices have less available real estate on the screen, and certain mobile devices do not support a small subset of UI elements.

One specific issue concerning the Web Dynpro UI

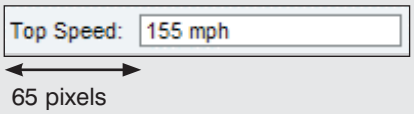
I have encountered more than one person who has formed a negative opinion of Web Dynpro because it lacks the ability to position UI elements explicitly on the screen. This feature is referred to as a "pixel-perfect" alignment of UI elements and is available in most other Web development toolsets. I mention this specific topic here because it is representative of many people's opinions.

Let's say you have a business application that will be run in both German and English. As you would expect, your screens will have labels and input fields on them. Web Dynpro has been designed to support multiple languages automatically, and does not require the developer to write language-specific coding.*

The screenshot to the right is an example of an English label and input field that could occur on the screen.

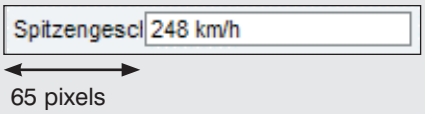


Now let's say that you were able to position these UI elements precisely on the screen. So, for instance, you have specified that the input field will be located 65 pixels from the left-hand edge of the screen, as shown in the screenshot to the right.

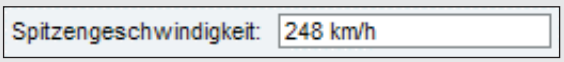


This is all fine — or is it?

Now let's run the application in German and see what would happen to the display if everything except the language were left unchanged. The screenshot to the right shows the result — as you can see, the label has been truncated because the input field is at a fixed location.



This is obviously not a good situation. What we are expecting to see is the screenshot to the right, where the input field is correctly positioned according to the natural length of the label.



If Web Dynpro were to allow the developer to position UI elements exactly on the screen, then such information would immediately become language specific because of the simple fact that words change length (and possibly direction) when translated from one language to another.

Translation is an awkward enough process as it is without adding UI element placement into the mix. Therefore, SAP took the deliberate design decision to have all UI elements positioned automatically by the Web Dynpro Framework.

* It is, however, possible to write language-specific code in Web Dynpro should you need to.

Although there is no concrete definition of a Web development toolset, there is a set of criteria that such a product is generally expected to provide. Broadly speaking, a Web development toolset will provide most, if not all of the following features:

Expectations	Met by Web Dynpro?
A strong focus on the flexibility of the UI layer	No — the stability and reliability of the business application is considered a higher priority than specific features of the UI
A library of UI objects	Yes
General adherence to the MVC design pattern	MVC has been implemented in an SAP-specific manner
Some type of "component" concept	Yes
Management of the application's state	Yes
Business object persistence	Yes
A library of generic services	Yes
A "page at a time" approach to client communication	No

Figure 1 Does Web Dynpro meet the general expectations for a Web development toolset?

Features of Web Dynpro	Expected by an untrained developer?
Serve as the de facto UI programming toolset for all future SAP software products	No, and probably not considered important
Eliminate the repetitive coding tasks currently experienced by Web developers	Yes, but not to the extent implemented by Web Dynpro
Create a fundamental unit of software reuse that exists at a business level rather than at some lower, technical level	Probably not
Use a declarative approach to application design	Yes, but would depend largely on previous experience
Use abstract modeling for communication with a back-end system	Yes
Use abstract modeling to create client device-independent screen layouts	No
Make full use of generic services	Yes

Figure 2 Would an untrained developer expect the features provided by Web Dynpro?

- A strong focus on the flexibility of the UI layer
- A library of UI objects
- General adherence to the Model-View-Controller (MVC) design pattern
- Some type of "component" concept
- Management of the application's state
- Business object persistence
- A library of generic services
- A "page at a time" approach to client communication

The most strongly expected feature is the first — flexibility of the UI layer. This feature is so pervasive amongst other Web development tools, that it is widely regarded as a requirement rather than a feature. See the sidebar above to understand why SAP has not implemented this feature.

So where's the mismatch?

The problem comes from the fact that Web Dynpro is assumed to be a Web development toolset; therefore it is expected to fulfill all the expectations just listed. However, it does not. This is not because SAP decided they weren't important, but rather because it had to meet its own set of design criteria.

Figure 1 and **Figure 2** show the contrasts: firstly, whether Web Dynpro matches people's expectations

for a Web development toolset, and secondly, whether the features provided by Web Dynpro are expected.

As you can see from **Figure 1**, SAP has deliberately shifted the focus away from UI flexibility and onto application stability and reliability. Then from **Figure 2**, you can see that an untrained developer using Web Dynpro for the first time would not expect to find the unit of software reuse at the business process level rather than at the technical level.

The other unexpected feature is the use of abstract modeling to create client device-independent screen layouts instead of directly using HTML, JavaScript, or CSS. Web Dynpro does not allow you to use these markup and scripting languages — if you did, then you would immediately have created client-dependent screens, and in doing so, violated a fundamental design requirement of Web Dynpro. However, some developers are so used to having this level of control they believe that without it, Web Dynpro could not possibly work successfully.

These factors — software reuse at the business level and using abstract modeling to create client device-independent screen layouts — are the key differentiators between Web Dynpro and other Web development toolset. As a result of this shift in design priorities, SAP has created a Web development toolset that has a strong focus on:

1. **Code reuse at the business process level:** The unspoken principle here is coding is bad … if you have to write the same piece of code twice.
2. **Code maintainability:** The easier an application's architecture is to understand, the easier it is to maintain. This has a significant impact on lowering an application's total cost of ownership (TCO).
3. **Separation of concerns:** The different software units within a Web Dynpro application are divided into those that generate data and those that consume data. It is very important that you do not blur the functional boundaries between these different units; otherwise you will create an application that is very difficult to maintain.

The bottom line is that as a result of the natural tendency for people to think that the new product (Web Dynpro) ought to behave in a way they are used to, they often try to press-fit Web Dynpro into their expectations — and when it doesn't fit, they adopt one (or both) of the following stances:

- They feel justified in condemning the product.
- They force Web Dynpro to behave according to their expectations.[4]

If the second stance is adopted, the result is an application that may well be functional, but is excessively complex, difficult to maintain, and will probably experience performance and scalability problems.

OK, can anything else go wrong?

Unfortunately, yes: a couple of things in fact.

These problems actually have nothing to do with the Web Dynpro product, but rather are related to the way in which SAP implementation projects are often managed. I will mention them in this article only in summary form because they are tangential to the main subject of Web Dynpro for Java.

I recommend that you download a more detailed description of these problems from the *SAP Professional Journal* Web site at http://www.sappro.com/downloads.cfm. This will give you a greater insight into the problems that can be created when a poor understanding of Web Dynpro and poor project management coincide. Of particular importance is the diagram in the "Vicious Circle" section. **Warning:** It gets ugly!

In short, the purpose of Web Dynpro was misunderstood and this led to a variety of problems:

- **Developers trying to work from ambiguous functional specifications:** If a functional specification does not describe the required processes with clarity, then there will be room for "interpretation" of the specification's meaning. The developers then write, test, and deliver what they think the specification means, only to find that the program fails

[4] A square peg will fit into a round hole if you hit it hard enough….

user acceptance testing. I have worked on projects where the specification meant one thing to the business users, but a different thing to the developers. Needless to say, a lot of time, effort, and money was wasted during extended periods going around the test-reject-fix cycle.

- **Allowing the planned functional scope of a project to flex when "spur-of-the-moment" ideas are thought up:** There are always situations during an implementation where you find that the original plans were not comprehensive enough, and so some in-flight adjustments must be made to the scope of the delivered product. However, a balance is required here between dogmatically sticking to the plans, and letting spontaneity run riot. When the scope is allowed to expand without a corresponding increase in project resources or an extension of the time frame, then the quality of the delivered application will necessarily drop.
- **Ending up in the "Vicious Circle":** If all the problems described here coincide, then you will probably end up in what I describe as the "Vicious Circle" scenario. This is not a nice place to be at all, but when I conducted a quick straw-poll during one of my SAP Tech Ed 2007 sessions, I asked the attendees if they could identify with this situation — at least 1/3 of the people in the room put their hands up.

Avoiding the pitfalls

I've spent quite a long time describing what can go wrong during Web Dynpro implementations, and if I were to stop here, you could easily go away thinking that any project you attempt would turn into *Nightmare on Web Dynpro Street*. However, the best way to avoid falling into these traps is to go into a Web Dynpro implementation with your eyes open, fully understanding the purpose of Web Dynpro, understanding the mistakes made during other implementations, and then ensuring they are not repeated.

So now let's look at the positive side, and we'll see that as long as you are correctly prepared, a Web Dynpro implementation can run smoothly and result in software that is easy both to understand and maintain.

I have divided the following design principles into the broad categories of "high level" and "low level," the difference being that high-level design principles apply to the construction of the overall application where your building block is the Web Dynpro component, while the low-level principles apply to the internal design of a single component.

In the following sections, I have spent more time focusing on the high-level design topics than the low-level ones. This is partly because the low-level design information is already covered in the standard SAP training courses for Web Dynpro Java (JA310 and JA312), and also because the consequences of a poor overall application architecture are far harder to correct than correcting a single, poorly written Web Dynpro component.

High-level Web Dynpro design principles

Let's now put all those mistakes behind us and look at how to do things correctly.

First, we will look at the most fundamental or core principles.

- SAP has modified the MVC design pattern.
- Java developers think in classes; Web Dynpro developers think in components, where one Web Dynpro component = one business task.

Next, we will look at the design principles for constructing a business application from multiple Web Dynpro components.

- Create Web Dynpro components to perform specific business tasks.
- For traditional Web development, the main design priority is specific features of the UI. For Web Dynpro development, the main design priority is the stability and reliability of the business process.
- Use development components of type "Web Dynpro" to group together related Web Dynpro components.

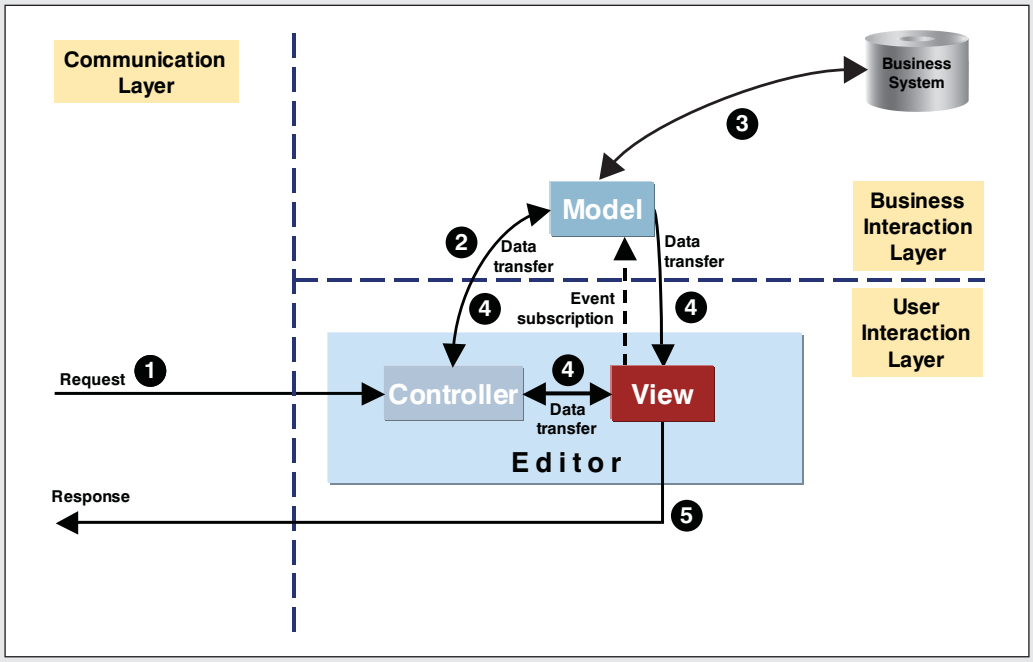


Figure 3 A typical MVC implementation

- Use standalone component interface definitions (SCIDs) to increase design flexibility.

These fundamental principles create consequences that have a profound knock-on effect throughout the whole of Web Dynpro design and architecture. Therefore, if you understand these principles, you will understand why SAP recommends certain coding styles and architectural structures.

How SAP has modified the MVC design pattern

The MVC design pattern has been in use since 1978 and, like any other design pattern, there is no single "correct" way of implementing it. However, a typical implementation is shown in **Figure 3**.

In this scenario, the following sequence of events is typical:

1. A controller receives a request directly from the client — typically a browser.
2. After analyzing the data in the request, the controller then decides what action to take. This could involve passing information to a model to retrieve data from a back-end system.
3. The model retrieves the data from the back-end system. The original MVC specification regarded views as "presentation filters" through which to display the model's data. The consequence of this feature is that models often raise events to which the views directly subscribe.
4. Once the model has retrieved the data, it is distributed to the various controllers and views in the application. In this scenario, views can receive data directly from models.
5. The view then creates the output suitable for the client device and returns the response.

There are several key differences here between the way a typical MVC application is constructed, and the design criteria laid out by SAP for Web Dynpro.

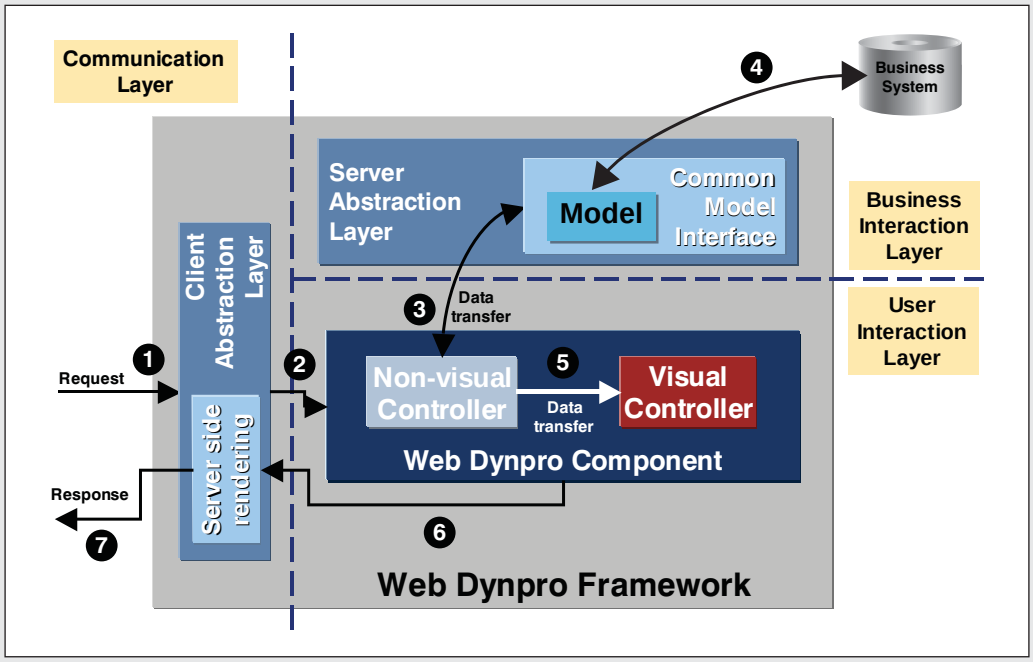


Figure 4 How SAP has implemented MVC in Web Dynpro

Firstly, the above implementation does not naturally allow for client independence. This immediately forces the coding in both the controller and view to be client dependent. As I mentioned earlier, this is where as much as 90% (according to Google) of the developer's effort can be swallowed up on a Web project.

Secondly, notice the units of reuse. Models, views, and controllers are all reusable individually and independently of each other. This puts the level of reuse granularity at too low a level. A single view or controller is very unlikely to represent a single step of the business process; therefore, in a typical MVC application, code reuse occurs at a lower technical level, rather than at the business level. This immediately complicates reuse efficiency because multiple units of code now need to be referenced in order to complete a single step of the business process. SAP's design criterion explicitly states that Web Dynpro applications should be based on units of code reuse that represent distinct steps of the business process, not lower, technical units of code.

Thirdly, there is no concept of abstract modeling. The controller that receives the incoming request must understand how the specific client devices will communicate with it. Also, the views must be able to generate output that is specific for the range of clients acting as front-end devices. The consequence of this situation is that if ever you wanted to support a new type of client device, you would have to modify the coding in all your business applications. SAP decided that Web Dynpro should not care about the specific requirements of either the client device or the back-end server. Therefore all Web Dynpro applications have been designed so that their UIs are specified in a client-neutral manner and all model objects look the same irrespective of the protocol required to communicate with the back-end server.

Remember SAP's design criteria listed on page 30. You can see that the traditional implementation of MVC does not meet these criteria; therefore SAP had to implement the MVC design pattern in a modified manner. As you can see in **Figure 4**, a Web Dynpro

application looks somewhat different from that seen in **Figure 3**.

The Web Dynpro implementation of MVC includes all of SAP's design criteria, without losing the fundamental concept (more on this in the next section).

1. When an incoming request arrives at the Java server running Web Dynpro, the Client Abstraction Layer (CAL) first converts the information into a client-neutral format known simply as a "data container."

2. The client-neutral data container is then passed to the relevant instance of the Web Dynpro component responsible for the running application. Notice here that the unit of software being invoked is a component, not an individual controller or view.

3. When data is required from a back-end system, the consequences of SAP's requirement for abstract modeling can be seen in action. Instead of the Web Dynpro component communicating directly with the model, it communicates with the Server Abstraction Layer, which in turn hides the model object behind the Common Model Interface (CMI). The purpose of the CMI is to make all model objects look the same irrespective of the protocol[5] required to communicate with the back-end server.

4. The model now communicates with the back-end system and the data is returned to the Web Dynpro component.

5. In a Web Dynpro component, instead of having distinct entities called controllers and views, SAP has modified the MVC concept so that there are two basic categories of controller. The difference between these categories is simply whether the controller has a visual interface or not. A Web Dynpro view is simply regarded as a controller that has a visual interface. This architectural change is the direct consequence of SAP's requirement for Web Dynpro to be client device-independent.

   Notice the arrow under step 5. It is only single headed. This is because a non-visual controller always acts as a supplier of data to the visual controller. This is another fundamental principle in Web Dynpro, but more on that later.

6. Once the business processing has completed for the current roundtrip, the Web Dynpro component sends its output back to the CAL. The Web Dynpro component does not know (neither does it care) about the precise implementation details of the client. Consequently, it does not need to worry about generating any HTML or JavaScript. This task is performed by the server-side rendering within the CAL.

7. The client-specific output is now returned to the client, and the current roundtrip comes to an end.

As you can see from the diagram in **Figure 4**, all of these processing steps take place within the Web Dynpro Framework. This is the runtime environment within which all Web Dynpro components are executed, and it exists because of the design requirements for both client independence and abstract modeling. The Web Dynpro Framework automates all aspects of:

- Client communication — through the CAL
- UI rendering — through server-side rendering within the CAL
- Back-end communication — through the Server Abstraction Layer

Taken together, these factors will eliminate at least 50% of your implementation development effort. However, these benefits come with the realization that Web Dynpro cannot be used as if it were Ruby on Rails or Struts. Since Web Dynpro's primary focus is the implementation of business processes in an efficient, reliable, and reusable manner, the flexibility of the UI to which people have become accustomed[6] is treated as a secondary priority.

[5] The back-end access protocols currently supported by Web Dynpro are Remote Function Call (RFC), Simple Object Access Protocol (SOAP), and Remote Method Invocation (RMI).

[6] And therefore regard as "indispensable."

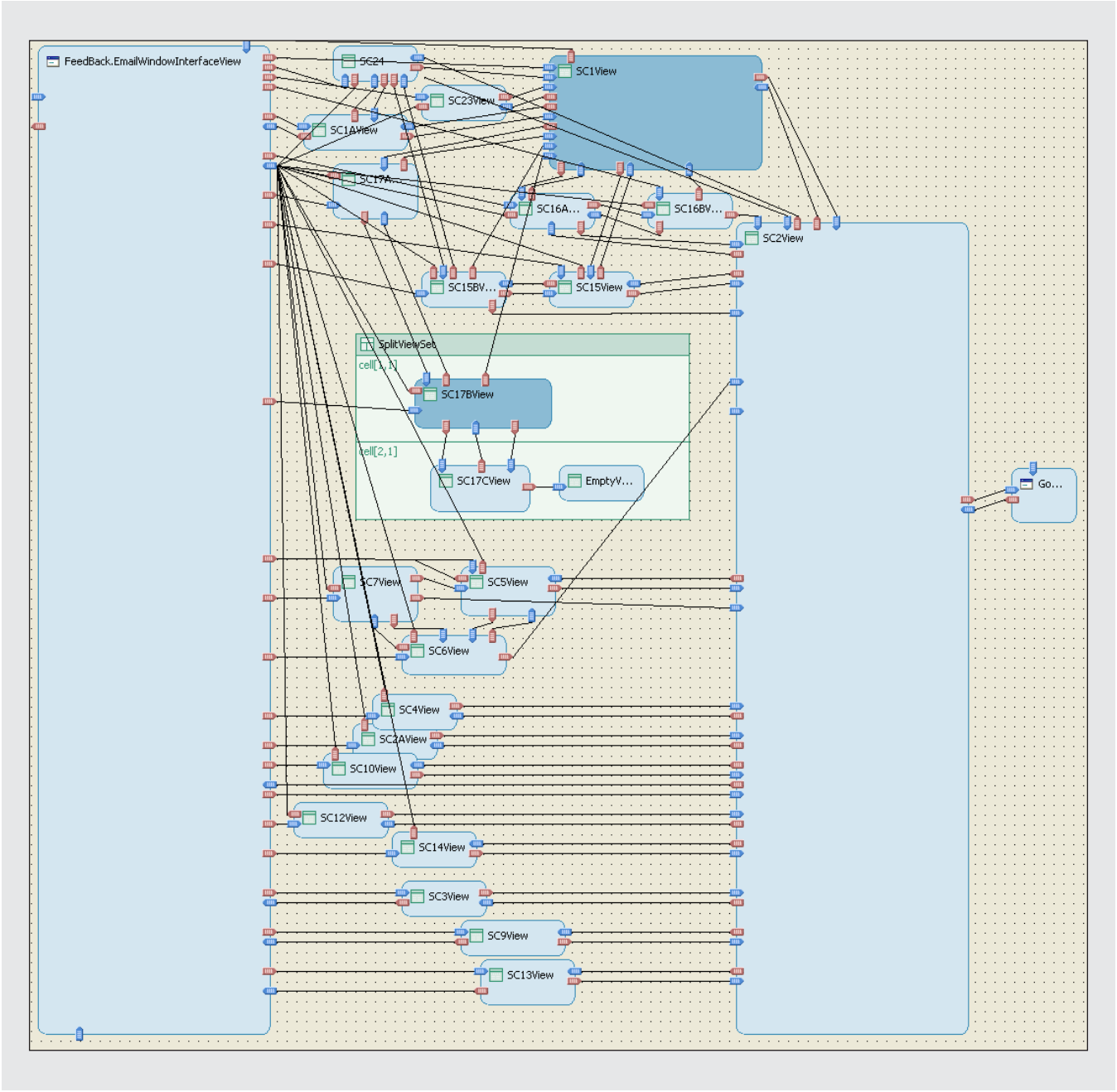


Figure 5 The navigation window of a monolithic component

Think in components, not classes: one component = one business task

The next mental shift developers need to make is to think of the Web Dynpro component as both their fundamental unit of development and reuse — not an individual Java class. This is essential in order to be able to create low TCO Web Dynpro applications.

As before, the importance of this point can be illustrated by showing what happens if it is not followed. In **Figure 5** and **Figure 6** (on page 40), you

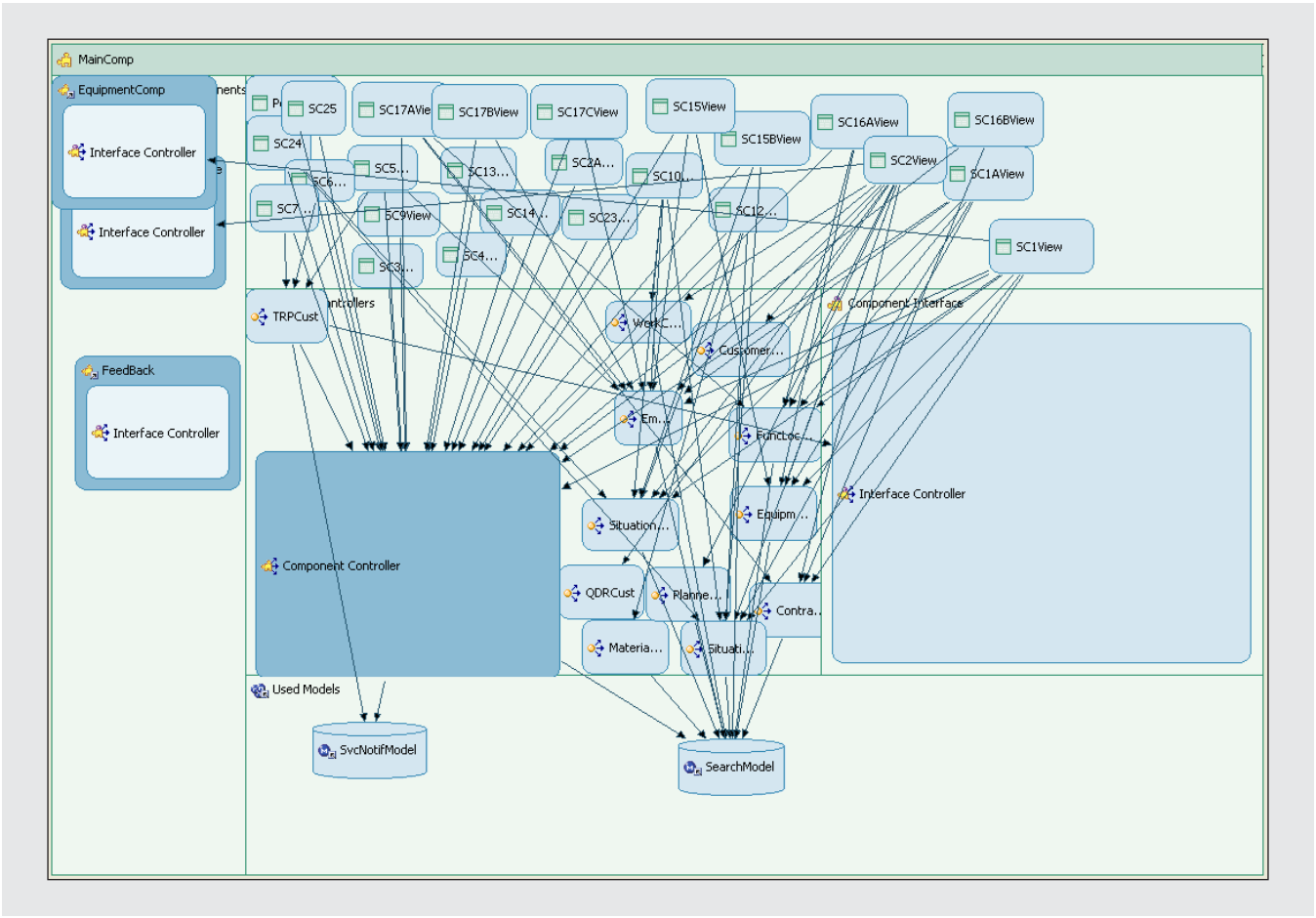


Figure 6 The component editor window of a monolithic component

can see what happens if you try to put all your functionality into a single component.

Not a pretty sight!

There are several distinct problems with monolithic Web Dynpro applications (these problems are in fact not specific to Web Dynpro, but to software applications in general). As the component grows in size:

- The coding it contains becomes less reusable.
- The coding becomes harder to maintain as layer upon layer of fixes or enhancements are added.
- The developers who work on the fixes and enhancements gain specialist application knowledge that becomes increasingly important to the company. If these developers leave (taking their knowledge with them), then a new developer will require a significant block of time to study the code before they can safely modify it.
- Eventually, the TCO of such a piece of software becomes disproportionately large when compared with the benefits it provides. When you reach this point, it's time to throw the application away and start again; however, it's better not to end up in this situation in the first place (for an insight into how this can happen, see the download available at http://www.sappro.com/downloads.cfm).

There's no simple answer to the question, "How large should a Web Dynpro component be?" However, you should find a balance that is based on optimal reusability. This is illustrated in **Figure 7**.

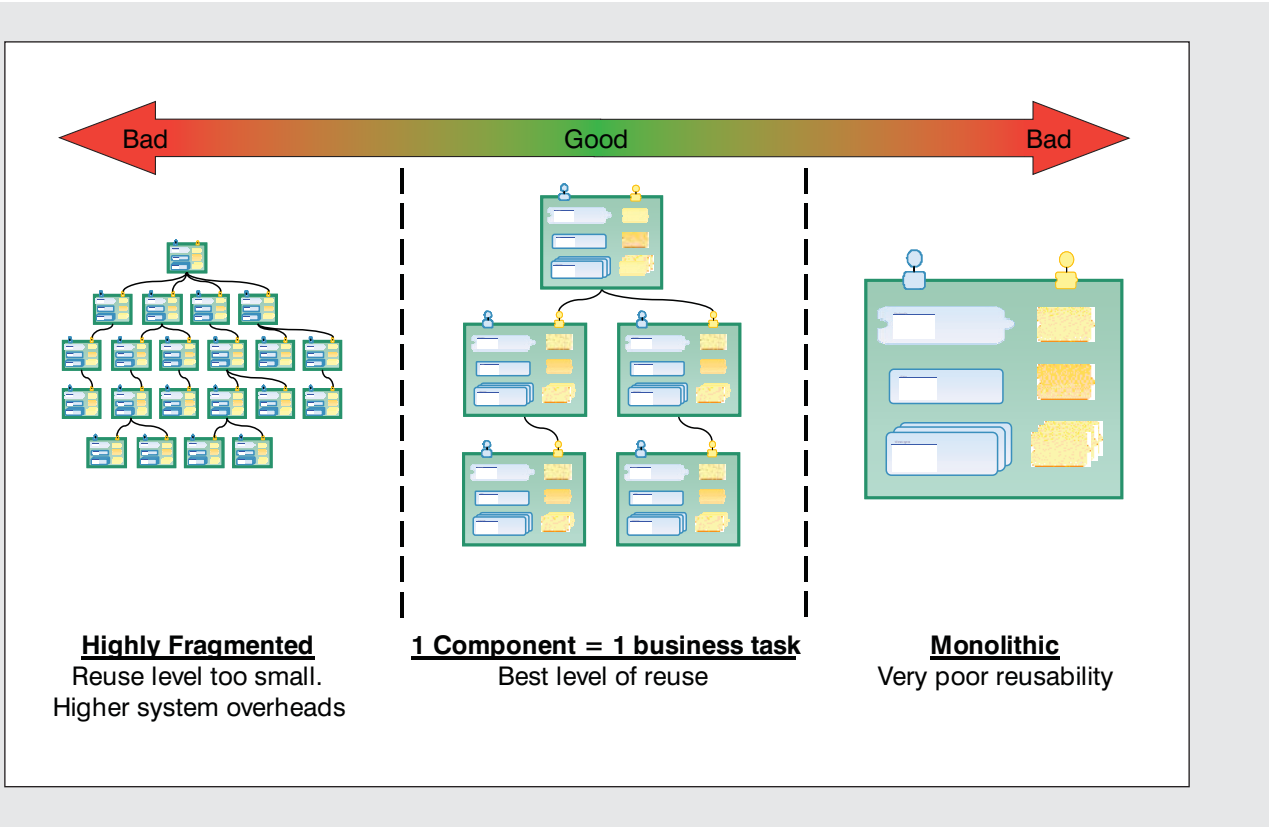


Figure 7 Component granularity is a balance between two extremes

If you make your Web Dynpro components too large, their reusability suffers because they contain too much functionality. On the other hand, if you make the components too small, then the reuse scenarios become too complex, and the SAP Java Server will have a greatly increased workload due to the increased number of component instances needed for each application instance at runtime. This leads directly to the design principle that *one component = one business task*.

The component has been designed to act as the building block for all Web Dynpro applications. Within this design concept, you should always design a component to have the maximum level of reuse: Having said that, a component's position in a hierarchy will also determine its level of individual reuse.

As shown in **Figure 8** (on the next page), the components at the bottom of the hierarchy will typically contain highly reusable units of functionality. This would include such tasks as access to a back-end system or managing error or help messages. These low-levels components have been specifically designed for maximum reuse and are generally referred to as "utility" components.

As we move up the component hierarchy, the reusability of individual components decreases because they are designed to bring the functionality of the lower level or "child" components together in such a way as to perform a specific business task. The component at the top of the hierarchy is known as the "root" component and has the specific task of bringing the functionality of all the child components together into a working application.[7] Even though the root component is not reusable individually, it is

[7] See my article in the January/February 2007 issue of *SAP Professional Journal* for more on how components function within a hierarchy.

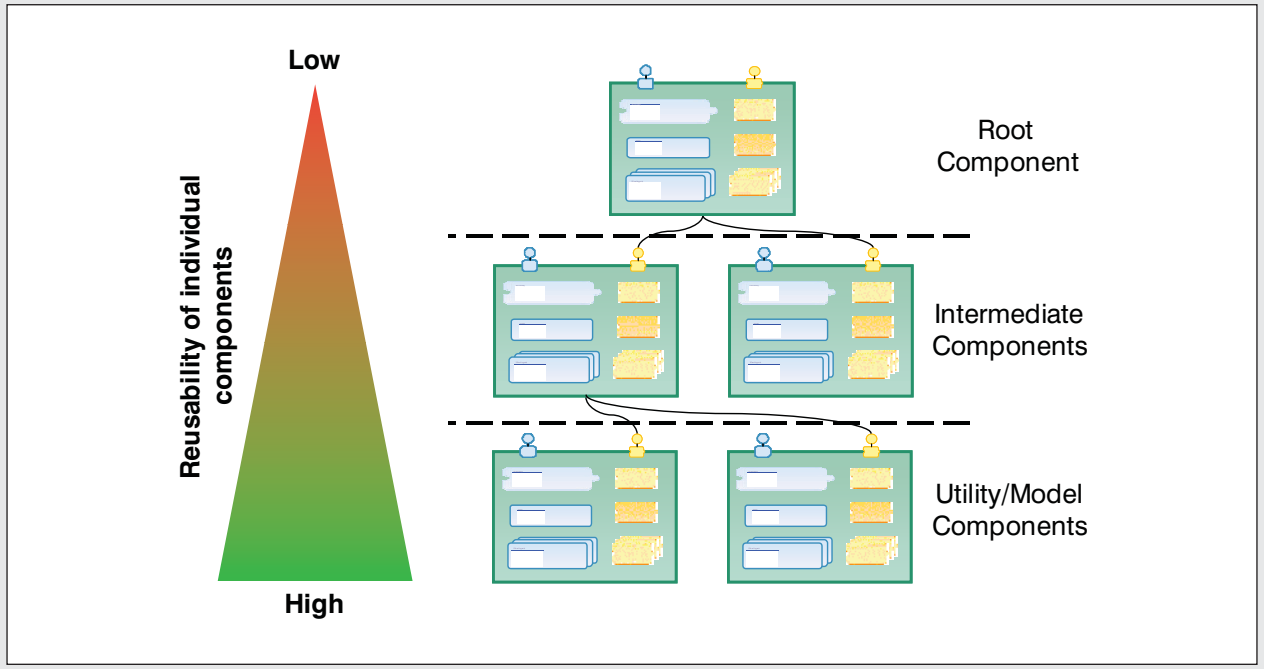


Figure 8 The reusability of a component varies according to its position within the hierarchy

completely reusable if you treat your unit of reuse as the entire hierarchy.

Create Web Dynpro components to perform specific tasks

All Web Dynpro components are constructed from the same basic units arranged in the same way, as shown in **Figure 9**. As you can see, the component has a vertical dashed line down the middle that separates the visual entities on the left from the non-visual entities on the right. This means that there are entities within the component that are responsible for placing UI elements on the screen and populating them with information, and entities that take no direct part in the management of these UI elements. The non-visual entities generate the business data and manage the flow of information through the application (i.e., they are data generators); the visual entities are responsible for handling user interaction and are consumers of data — either from the user or from a non-visual entity.

The horizontal dashed line across the top of the diagram divides the component into those parts that can be seen outside the scope of the component (i.e., that are externally visible) and those that are hidden from view. It is important to understand that a Web Dynpro component presents two interfaces (i.e., two externally visible entities). One is programmatic and the other is visual. **Figure 10** provides more details on the controllers found in a Web Dynpro component.

You might wonder why the vertical dashed line cuts through the interface view, window, and view controllers; it might seem from my description above that the entire view controller is a visual entity. This is not quite true.

In the case of a view controller, the "visual entity" is the view layout. This is the only part of a view controller that is responsible for the presentation of data on the screen. The view layout, however, is integral to and cannot exist outside of a view controller. The coding in the view controller acts upon the UI elements, but does so in a way that is decoupled from

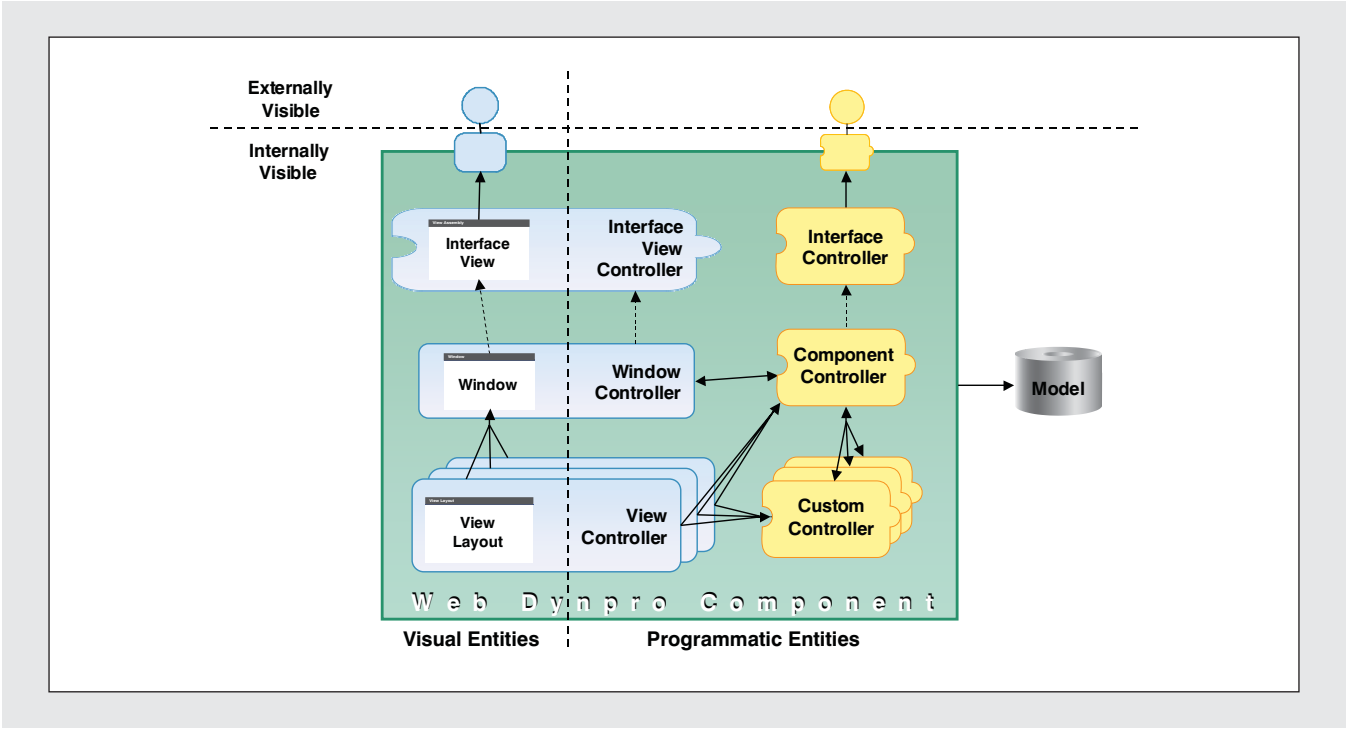


Figure 9 The architecture of a Web Dynpro component in SAP NetWeaver CE

Entity within a component	Type	Description
Component controller	Non-visual	Acts as the central point of control for all processing within the component. In effect, this controller *is* the component.
Interface controller	Non-visual	In SAP NetWeaver CE, this controller is a true interface implemented by the component controller. In SAP NetWeaver ’04 and 7.0, this controller is a distinct class containing its own coding.
Custom controller	Non-visual	Created only when explicitly required by the developer. Used to encapsulate functionality such as that required for an Object Value Selector.
View controller	Visual	Responsible for the display of information on the client and for user interaction.
Window controller	Visual	Handles the processing related to an aggregation of view controllers. This controller does not exist in the SAP NetWeaver ’04 and 7.0 versions of Web Dynpro.
Interface view controller	Visual	Acts as the component’s visual interface at runtime and is implemented by the window controller.

Figure 10 Description of each controller’s role within a component

the actual UI element objects. This is achieved via “data binding.” See the section “Manually setting or getting UI element property values instead of using data binding” on page 54 for more details on this.

As you design your application, you will find that certain units of functionality are frequently required and become natural candidates for encapsulation as Web Dynpro components. Such units of functionality would be things like simplifying the access to a back-end system if a particularly complex interface must be used, or standardizing the handling of error messages, or defining an abstract screen layout that is only populated at runtime. These types of tasks fall into one or more of the following general categories:

- **Model components:** Access to back-end systems, particularly if the interface is complex or large
- **Utility components:** General purpose tasks such as error message management
- **Faceless components:** Any component that makes no contribution to the application's visual appearance (both model and utility components could also be faceless components)
- **Visual components:** Define an abstract screen layout that is dynamically populated at runtime

Model components

The interface to a model object is often large and complex. For instance, take the BAPI used to create purchase order documents. It has a large and complex interface, and the business scenario being implemented often does not require access to all the parameters in the interface. Therefore, it makes no sense to build complex interface coding into every component that uses this model object. It is far more efficient to encapsulate the interface logic into a Web Dynpro component and reuse that. This is where the creation of a model component is recommended (remember, a model component is simply a regular Web Dynpro component that has been written to simplify the interface to a model object).

Into this model component, you place any specialized coding needed to perform pre- or post-processing on the business data as it passes into and out of the model object. You then add various methods to the component's interface to provide easy access to the model object's otherwise complex interface.

Once you have developed a model component, the business application component requiring the model's functionality declares a usage of it, thus gaining access to the simplified interface.

Utility components

A utility component is a regular Web Dynpro component that has been written to perform a widely used and frequently required task. Examples of such components are those that handle error messages or that perform user authentication. These components will occupy the lowest levels of a component hierarchy and will be widely used by multiple applications (see **Figure 8**). Often, these utility tasks will require no interaction with the user; therefore, utility components will often have no need for a visual interface. Such components are known as "faceless" components.

Faceless components

A faceless component is any component that makes no contribution to the business application's visual interface. In most cases, utility components will not be required to present information directly to the screen; therefore it is generally true that a utility component will also be a faceless component.

However, you should understand that even if a component has a visual interface, it can still be considered "faceless" if it makes no contribution to the business application's visual interface. For instance, a model component is required when access to a back-end system is performed through a large or complex interface. In this case, the model component presents the rest of the application with a simplified interface to the back-end system.

As far as the business application is concerned, the model component behaves simply as a special kind of utility component that operates in the background. However, the model component will probably need to have a visual interface for the purposes of unit testing and administration. So in this case, even though the model component has a visual interface, it would still be described as "face-

less" because that visual interface is not required for the business application's user interface.

Visual components

It should be no surprise that a visual component is the opposite of a faceless component! A faceless component contains functionality, but no visual interface; a visual component has a well-defined visual interface, but contains little or no functionality. The purpose of such a component is to create an abstract screen layout definition, without being concerned for the exact content. Such components are used to define *floorplan manager components*.

Focus on the reliability and stability of the business process, not UI features

You should now have a good understanding of why Web Dynpro does not provide the same degree of flexibility in the UI found in other toolsets (though future releases will include some enhanced capabilities; see the sidebar below). This is simply because

Future enhancements to Web Dynpro UI capabilities

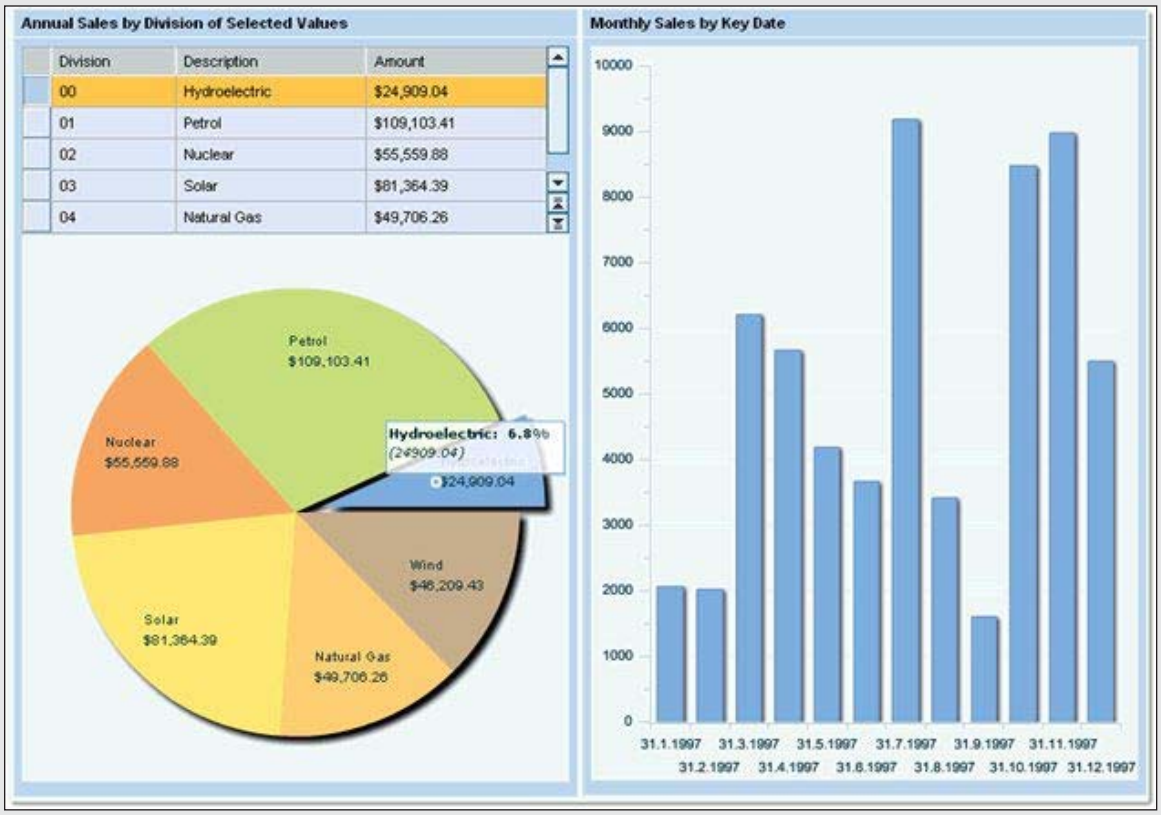
Now that Web Dynpro is a stable product, SAP has turned its attention to its more cosmetic aspects. Therefore, in future releases you will see two major additions to Web Dynpro's UI capability:

- **Java Server Face (JSF) bridge:** For those situations in which you really need to implement some highly customized UI elements, you will soon be able to embed JSF applications within Web Dynpro views. This will then allow you to create whatever custom UI functionality you require, but treat it as a generic UI element within the Web Dynpro view controller (**Figure 9** on page 43 shows the position of a view controller within the Web Dynpro component). The screenshot below shows a UI mashup created using Web Dynpro and JSF.

Continues on next page

Continued from previous page

- **Adobe Flash islands:** In addition to the JSF bridge, a GenericFlash UI element will be delivered that will allow you to use Adobe Flash controls to provide things like transition effects and drag-and-drop capability. The screenshot below shows an Adobe Flash island embedded into a Web Dynpro screen.



if it were provided, then Web Dynpro would have failed to meet one of its fundamental design requirements: namely, client device independence. At the end of the day, it's far more important that your software is functional and reliable. Making it look snazzy is fine, but if it keeps falling over, or can't scale to meet usage requirements, where's the benefit?

Use development components to group related Web Dynpro components

We now have to deal with a clash of terminology. The word "component" is used in different situations with different meanings. Within the scope of Web Dynpro, the word component means the basic unit of software development and reuse. However, within the scope of the SAP NetWeaver Development Infrastructure (NWDI), the phrase "development component" is used.[8]

A development component (DC) is a metadata wrapper that contains information on:

[8] For a detailed introduction to the NWDI, see *SAP Professional Journal* articles "A guided tour of Java software development lifecycle management with SAP NetWeaver Development Infrastructure (NWDI): Part 1 — Fundamental concepts" (July/August 2007) and "A guided tour of Java software development lifecycle management with SAP NetWeaver Development Infrastructure (NWDI): Part 2 — The development process" (September/October 2007).

- How the contained software entities should be compiled
- The transport path through the system landscape
- Functionality exposed by means of "public parts"
- Usage dependencies on the public parts of other DCs

DCs come in a variety of types; for instance, Dictionary, J2EE Enterprise Application, Portal Content, and Web Dynpro. The problem comes from the fact that people confuse Web Dynpro components with DCs of type Web Dynpro. The Web Dynpro component is a unit of software, whereas a DC of type Web Dynpro is a container for compilation and transport. A DC of type Web Dynpro will typically contain multiple Web Dynpro components.

SAP strongly recommends that Web Dynpro development should not be performed without a correctly installed and configured NWDI. Without NWDI, Web Dynpro development becomes an awkward and error-prone undertaking — particularly when several developers are working together on the same application.

The next question that people ask here is, "What should I put into my DCs of type Web Dynpro?"

Since a DC is a collection of software entities that are all compiled in the same way and all need to be transported around the system landscape together, you should group your models or Web Dynpro components together according to their overall functional relationship. For example, all the components that handle interaction with a particular type of business object in a back-end SAP system could be grouped together into a single DC.

Save development time by grouping related model objects into a DC

One time-saving tip is to place related Web Dynpro model objects into a single DC of type Web Dynpro. This DC will not contain anything other than model objects for the following reasons:

- Model objects tend to be large
- Model objects are time consuming to compile
- Model object do not change very often

Now consider what would happen to your rate of development if your Web Dynpro model objects and Web Dynpro component live together in the same DC. Every time you want to test a change to the business application, you must recompile and redeploy the coding. Is it necessary to recompile all the model objects every time you change the business application? Obviously not!

The whole point of this recommendation is to speed up development time by separating those parts of the code that change frequently from those parts that change rarely. If a unit of code changes only rarely, then don't package it together with coding that changes frequently.

To gain access to the model objects inside the model DC, each model object should be exposed to the outside world by adding it to a "public part." Then you only need to perform a "DC build" and your model DC is ready for use by other DCs.

Once the model DC has been created, it will remain relatively static — that is, you compile, build, and deploy it once, and thereafter, it will hardly ever change. Then the DC containing the Web Dynpro model components provides the simplified interface to the models. Together, these two DCs form a single reusable unit of functionality.

Figure 11 illustrates this principle (on page 48).

Use standalone component interface definitions (SCIDs) to increase design flexibility

In a typical business application of any complexity, you will have multiple Web Dynpro components functioning together in a hierarchy. The component at the top of the hierarchy, known as the "root" component, will need to have detailed knowledge of functionality provided by the lower level compo-

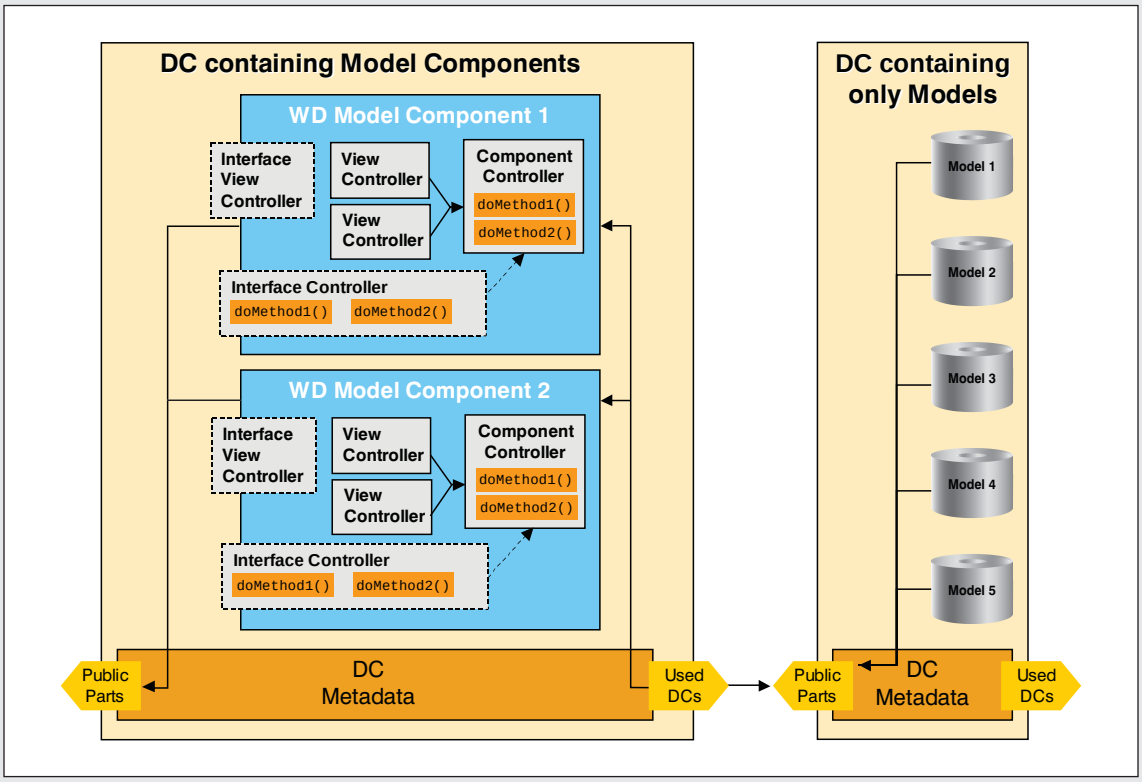


Figure 11 Two DCs of type Web Dynpro function together as a reusable pair

nents (generally known as "child" components). Conversely, a well-designed child component need not know anything about the root component acting as its parent.

However, if you tie a parent component too closely to its child components, you lose a significant degree of design flexibility. This is known as tight coupling and is illustrated on the right side of **Figure 12**. The parent (or root) component is tightly coupled to child component B. This is not necessarily a problem, but if you wanted to swap out a component for some other suitable component, as the relationship currently stands, you would have to make changes to the definition of the root component, even though no coding there needs to change.

Web Dynpro components can therefore implement something known as a standalone component interface definition (SCID), which is illustrated on the left side of **Figure 12**. The purpose of the SCID is to define a generic interface that can then be implemented by multiple components. As far as the parent component is concerned, it implements the SCID, and need not concern itself with exactly which child component supplies the delivered functionality. As long as the child component instantiated at runtime implements the same SCID, then any child component could be used at that point in the hierarchy.

This facility allows you to develop flexible, polymorphic applications that can dynamically swap different component instances in and out of the child component's position without affecting anything in the parent component.

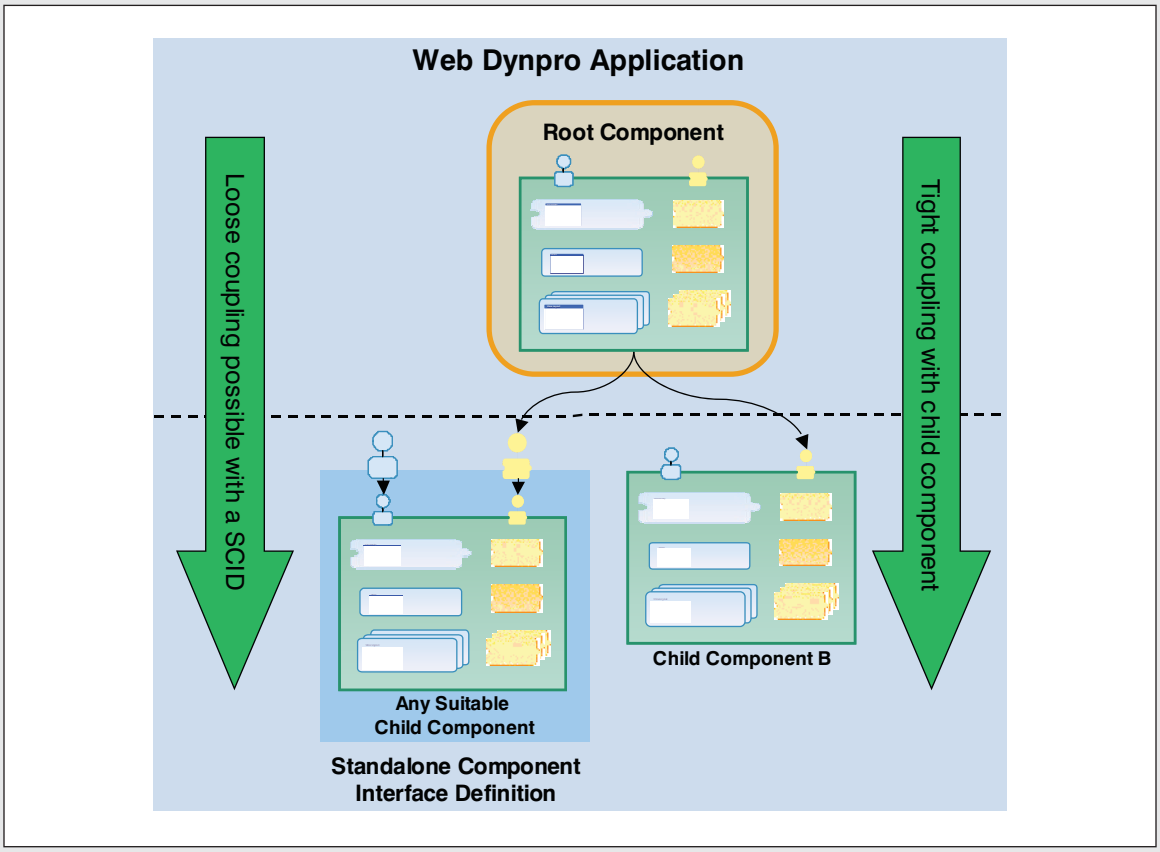


Figure 12 Tight coupling vs. loose coupling with a child component

Lower-level Web Dynpro design principles

Now that you have had a brief overview of the high-level principles, we should turn our attention to the lower-level design principles that apply to the internal design of a single Web Dynpro component. In the next sections, we will take a closer look at the following two key areas that can make or break your Web Dynpro components:

- The separation of data generators from data consumers
- The role of view controllers — and how to avoid abusing them

The separation of data generators from data consumers

The MVC design pattern is frequently cited as the best pattern for separating data presentation from data processing — and this is certainly true. However, the separation of data presentation from data processing is just one use case of a much more fundamental principle: namely, the separation of those parts of the program that *generate* data from those parts that *consume* data.

In order to get the best from Web Dynpro, you need to have a firm grasp of how this fundamental concept has been implemented within Web Dynpro. Once you understand this, you will then have a

common thread that ties together all the design principles that follow.

When I explained earlier how SAP modified the MVC design pattern, I said there are two basic categories of controller within a Web Dynpro component. The difference between them is simply whether the controller has a visual interface or not. This change to the standard MVC implementation was made necessary by the requirement that all Web Dynpro screens be developed in a client device-independent manner. However, you can also see the data generator-data consumer principle at work here.

Models

In **Figure 9** you can see that a model stands outside the scope of the Web Dynpro component. This is so that model functionality can be reused independently of the component functionality. In this architecture, a model is *always* considered to be a data generator. Even though the model requires input data to function, its role in life is to act as the interface to some back-end system.

Say, for instance, you want to create a purchase order document. A large quantity of information must be supplied to the model in order for this process to complete successfully. However, even though a large quantity of data is consumed and a small quantity of data is generated, the sole purpose of that input data is to complete a step of the business process. We know this step is complete when we receive the small quantity of data containing (amongst other things) the purchase order document number.

The data required by a model is used to drive each step of the business process, and we know whether the step completed successfully or not by looking at the data the model generates. When you look at the situation from the business process point of view, you will understand why models are always considered to be generators of data in spite of the fact that they frequently consume more data than they generate. In other words, an entity within Web Dynpro is judged to be a consumer or generator of data, not on the basis of the quantity of information that passes into or out of it, but on the basis of the role it plays within the business process.

Visual controllers

On the other hand, the visual controllers in a Web Dynpro component are *always* considered to be data consumers: they consume data either from a non-visual controller or from the user via the keyboard and mouse. The purpose of the visual controller is two-fold:

- To present information on the screen that has been generated by some other non-visual controller within the component
- To consume information from the user in response to the presented information

It is most important that the role of a visual controller is not abused — because this is one of the most frequently violated design principles I have seen in Web Dynpro implementations. (I will address this topic in the next section.)

Non-visual controllers

The role of a non-visual controller is to act as a middleman between the model and the visual controller. This means that with respect to the model, the non-visual controller acts as a consumer of data generated by the model, but with respect to the visual controller, it acts as a generator.

Separation of concerns

The distinction between these three fundamental units of coding[9] has a profound impact on the way a Web Dynpro component should be written. The best way to illustrate the correct design here is to look at the abuses of this principle.

For instance, if a visual controller should only ever act as a consumer of data, should it be written in such a way that it takes responses from the user,

[9] That is: models, visual controllers, and non-visual controllers.

directly interacts with the model, and then places the results on the screen?

Hopefully, you answered "no" to that question! The reason is that a visual controller is *not* responsible for generating the data it displays. When a Web Dynpro controller makes use of a model, then that usage is considered internal to the controller. Hence, if a visual controller were to interact directly with a model, then as far as the other controllers in the component are concerned it is behaving as if it were the generator of that data.

To enforce adherence to this design principle, SAP has designed visual controllers such that they are unable to share the data they contain with other controllers in the component. In other words, a visual controller may not act as a data source from which another controller can consume data. Consequently, the direct interaction between a visual controller and a model object is considered very poor design. To understand this better, consider the reuse case for data within a visual controller. If you were to code a visual controller such that it could interact directly with the model, what could you do with that data once it arrived within the visual controller?

Answer: very little.

Since a visual controller should never act as a data source (or data generator), it would be difficult (at least without breaking even more rules) to supply the information it contains to the rest of the Web Dynpro component.

The outworking of this principle means that a visual controller should never contain the coding to perform the actual business logic, because any data received from a model cannot be legitimately shared with other controllers in the component.

Unfortunately, many developers fail to understand this principle! I reviewed one customer project where the implementation partner's developers had no specific Web Dynpro training and were implementing their first Web Dynpro application. Without knowing any better, they had written it using a Java Server Pages (JSP) architecture style. They placed a copy of the database access coding (including the schema) into each of the 18 view controllers — and then they complained that the application was very difficult to maintain!

It should be pretty easy to spot the mistake here. As a consequence of having no specific Web Dynpro training, the developers assumed they could use a Web Dynpro view controller as if it were a regular JSP page. This was not their fault, but it did become their fight!

The resulting application was functional, but eventually became more costly to maintain than it was to write: consequently, it had to be scrapped and rewritten!

Do not abuse the role of view controllers!

As I stated earlier, the MVC design pattern does much more than separate the processing layer from the presentation layer: It separates those parts of the program that generate data, from those parts of the program that consume data. This is a fundamental principle that has a big impact on the internal structure of a Web Dynpro component. The most frequent example of a violation of this principle is found in the way people code view controllers.

Untrained developers tend to put their coding into whichever controller they can to obtain the required result. Consequently, they never maintain any functional distinction between visual and non-visual controllers — the coding is placed wherever it works!

A view controller should be written such that it is only ever a *consumer* of data — never a *generator*. This means that view controllers should *never* contain coding to interact with a back-end system.

This is really dangerous because the problems created by this type of abuse often do not become visible until it is too late to do anything about it. Just because a program is functional does not mean it has been well written!

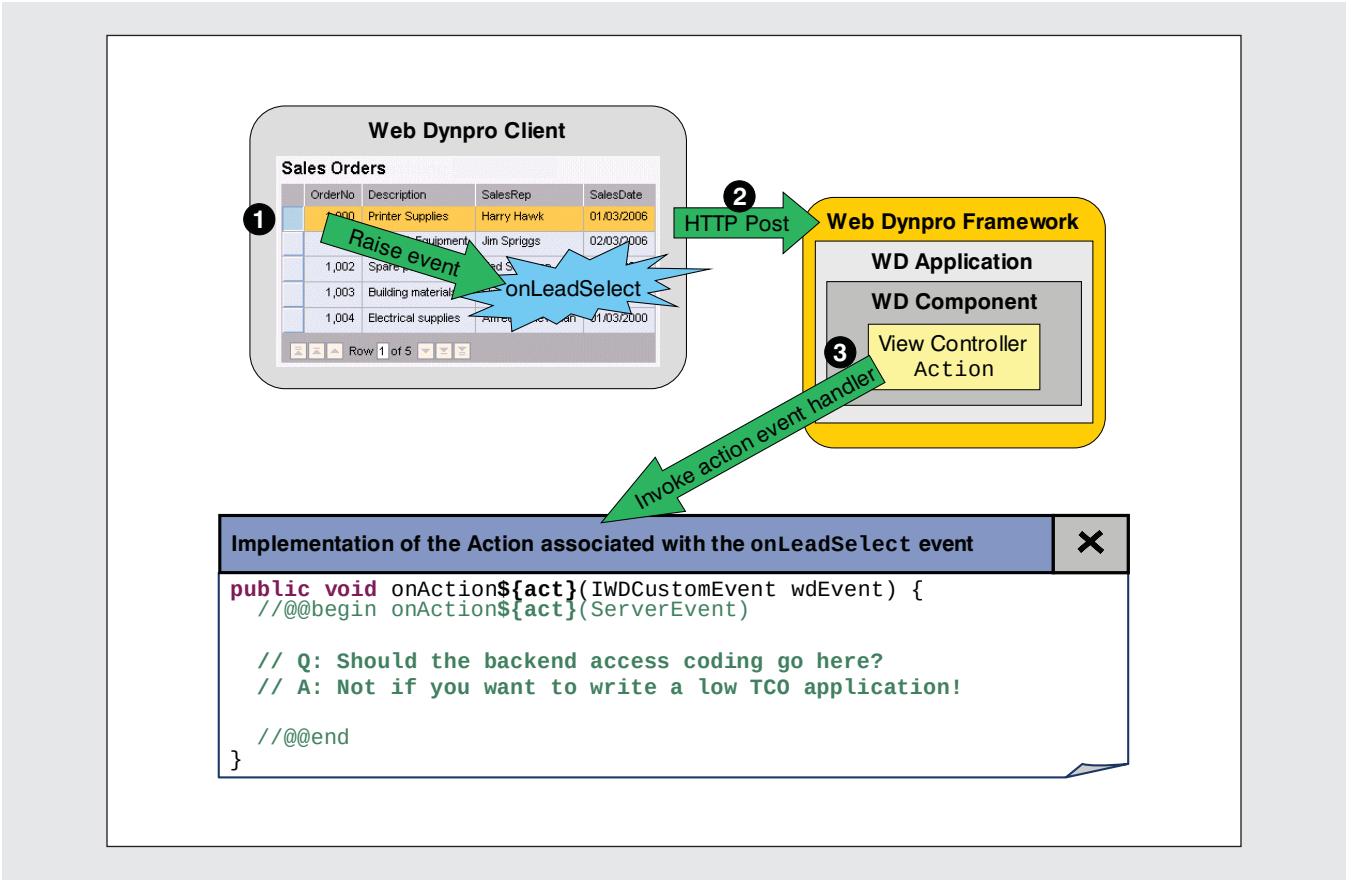


Figure 13 How a client-side event is processed by the Web Dynpro Framework

We will now take a quick look at the three main areas in which the role of a view controller is abused:

- Placing business logic within them
- Using the `wdDoModifyView()` method for anything other than UI element manipulation
- Manually setting or getting UI element property values instead of using data binding

Placing business logic into a view controller

First of all, I must explain exactly what is meant by "business logic." By "business logic," I am referring to coding that, having received information from the user, uses it to initiate the next step of the business process. This includes the pre-processing prior to invoking the business process and the post-processing required before the received data is presented to the user. In Web Dynpro development, initiation of the next step of a business process almost always adds up to invoking the functionality found in a model object.

The bottom line here is that Web Dynpro view controllers should never need to interact directly with a model object. From a technical perspective, it is perfectly possible to write such coding, and that coding will be functional. However, this is considered to be a very poor coding style because it creates a muddled architecture that rapidly becomes difficult and time-consuming to maintain.

Figure 13 shows a typical business scenario in which view controllers can be abused:

1. The user is looking at a table of sales orders on

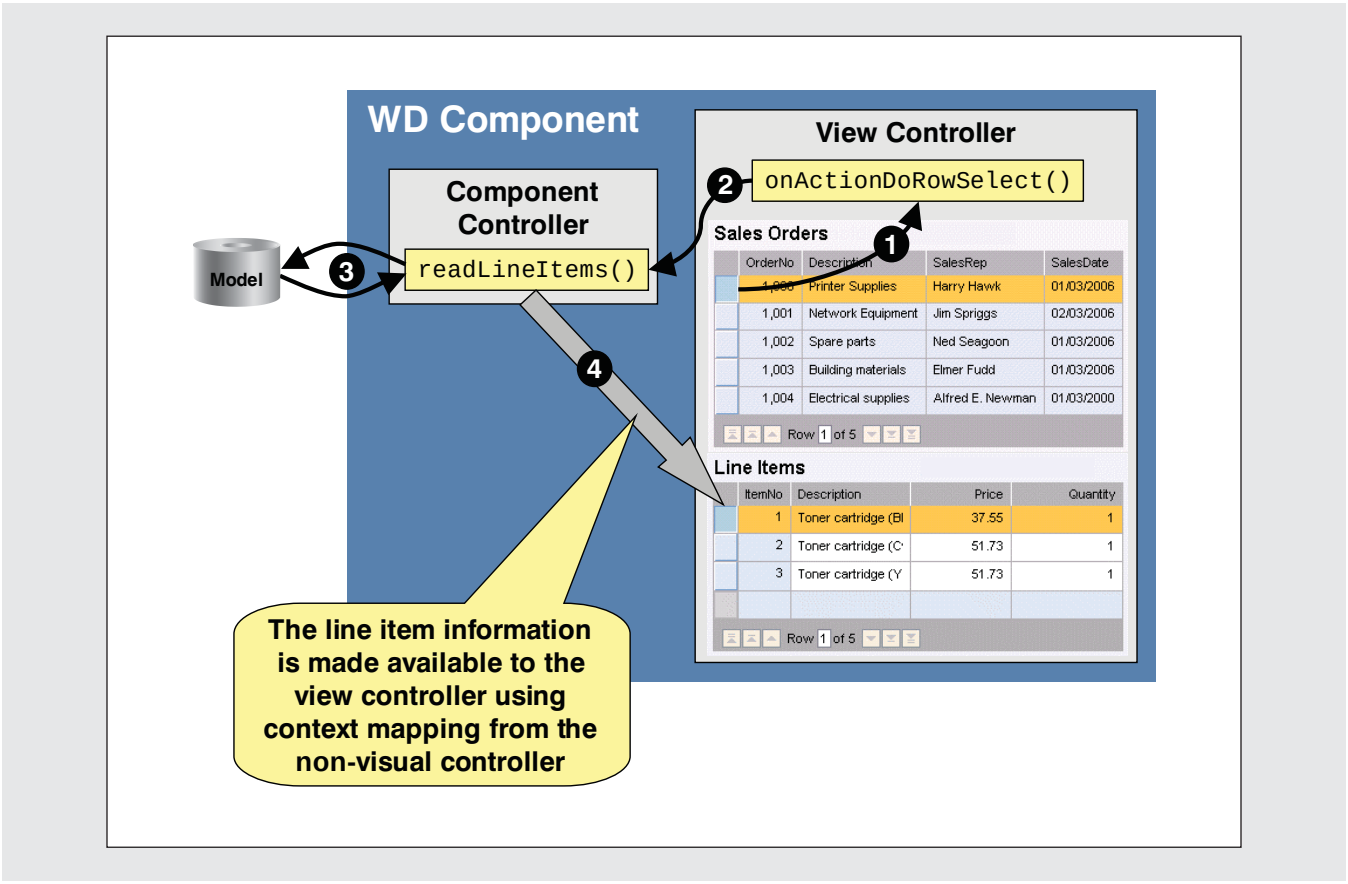


Figure 14 The correct architecture for back-end interaction in response to a client-side event

the screen. To see the line items that belong to a particular order, the user selects that row of the table. This causes an event to be raised in the client. In this particular case, that event is called `onLeadSelect`.

2. As a result of this client-side event being raised, the Web Dynpro coding embedded within the Web page now starts a roundtrip to the server and passes across all the relevant information.

3. During the development process, the `onLeadSelect` event has been associated with something called an "action." An action is a runtime object that contains the coding that will be invoked when the associated client-side event is raised.

This is all fine and dandy, but this is where so many developers end up writing poor-quality code. Technically speaking, there is no reason why the call to the model could not be placed directly in the view controller. Doing so will not cause any failure of your software from a technical or even functional perspective. However, it will create an application architecture that rapidly becomes very difficult to maintain.

SAP describes this split of functionality as a "separation of concerns" (see page 34).

When the coding within the Web Dynpro component follows the correct design principles, you will see the style of architecture shown in **Figure 14**.

1. Step 1 in **Figure 14** represents all three steps described in **Figure 13**.

2. Once control has been passed to the action event

handler method in the view controller, this method should delegate all interaction with the back-end system to a non-visual controller. Hence the call to a method in the component controller. In this example, this method happens to live in the component controller and be called `readLineItems()`.

3. The `readLineItems()` method then interacts with the back-end system via the model object. Another reason for recommending this architecture is that once the model object has returned the data to the non-visual controller, that controller can then share that information with *any* other controller in the component.

4. The data returned from the model object is passed from the component controller to the view controller through a data-sharing technique known as context mapping.

This architecture is much cleaner because there is now a single point of access from the entire Web Dynpro component to the back-end system via that particular model object. If you were to move the logic contained in the `readLineItems()` method into a view controller, then each view controller would require its own copy of the coding. Not only will this add redundant coding to the application, it will increase the overall system complexity and thus lead to an increased chance of error during maintenance. All in all, you do not want to find yourself in this situation. Fortunately, it is one that can easily be avoided.

Abusing a view controller's wdDoModifyView() method

As I stated earlier, a view controller is a Web Dynpro controller that has associated with it a visual interface. However, because of SAP's requirement to support client independence, a Web Dynpro visual interface cannot be defined using HTML, CSS, and JavaScript. This is simply because you cannot guarantee that all clients will understand your particular markup definition. Therefore, the UI elements present in a view controller's visual interface are specified in an abstract manner.

When the application is running, the only point in time during a roundtrip at which you can gain access to these abstract UI element objects is during the invocation of the view controller method `wdDoModifyView()`. Unfortunately, I have come across all manner of bizarre coding in this method!

By the time the Web Dynpro Framework has invoked this method, all the business processing for that roundtrip should have been completed.[10] All data received from the back-end system should have been processed and be in a state ready for presentation. The `wdDoModifyView()` method exists solely for the purpose of allowing dynamic modifications to the UI element hierarchy (see **Figure 15**). So, for instance, if some new data arrives from the back-end system whose structure has been defined dynamically, the `wdDoModifyView()` method can parse the structure and dynamically create UI elements suitable for its display.

Manually setting or getting UI element property values instead of using data binding

The principle of separating data generators from data consumers applies not only to models and controllers, but also at a smaller scale within an individual view controller. Since all UI elements within the view controller are specified in an abstract manner, there is a significant degree of decoupling between the coding found in the view controller and the UI element object.

Therefore, to present information on the screen in a reliable manner, Web Dynpro treats a property of a UI element as a data consumer, and an individual value held in the view controller's memory[11] as the data generator. This technique is called "data binding."

[10] For more details on how the Web Dynpro Framework handles standard methods such as `wdDoModifyView()`, see chapter 4 of my book *Inside Web Dynpro for Java* (Second Edition), available from SAP PRESS.

[11] I have not used the correct Web Dynpro terminology here since at no prior time in this article have I mentioned how Web Dynpro controllers manage local data storage. The correct terminology here is to use the term "context" instead of "memory."

Stub coding for the **wdDoModifyView()** method

```
public static void wdDoModifyView(
                    IPrivate${nv} wdThis,
                    IPrivate${nv}.IContextNode wdContext,
                    IWDView view,
                    boolean firstTime) {
  //@@begin wdDoModifyView
  if (firstTime) {
    // Dynamic UI manipulation code goes here

    // Do not put any business logic coding here!

    // Do not obtain references to UI elements in order to
    // set or get their property values!

    // Use context binding instead!
  }
  //@@end
}
```

Figure 15 Use wdDoModifyView() only for dynamic changes to the UI element hierarchy

Once you have placed a UI element on the screen, the properties of that UI element will need to obtain data from somewhere. This is where the declarative process of data binding comes in.

Let's say you have an input field that will sometimes be open to receive a value from the user, and sometimes be read only. In this case, data binding must be performed for two properties of the InputField UI element:

- The first is the `value` property. The UI element must have a defined location in which to store whatever the user types in; consequently, binding this property is mandatory. Failure to bind this property causes the input field to be completely non-functional and would lead to a runtime error.

- The second is the `readOnly` property. Binding this property is not mandatory. If you wished, you could leave it set to the hard-coded, default value of `false`. However, since we want to exert programmatic control over its value, this property should be bound to a Boolean value in memory[12].

Now that the data binding declarations have been made, the Web Dynpro Framework handles everything else for you. To find out what the user typed into the field, you do not need to interrogate the `value` property of the UI element object itself. Instead, all you need to do is read the variable[13] to which the `value` property is bound. In this manner, your application coding is decoupled from the specific implementation details of the UI.

If you want to prevent the user from typing into that particular input field, all you need do is set the variable to which the `readOnly` property is bound to `true`, and immediately the input field will be disabled.

[12] Or, to use the correct Web Dynpro terminology, "this property should be bound to a Boolean context attribute."

[13] Again for the sake of simplicity, I have not used the correct Web Dynpro terminology here. The `value` property of an InputField should always be bound to a context attribute of type `string`.

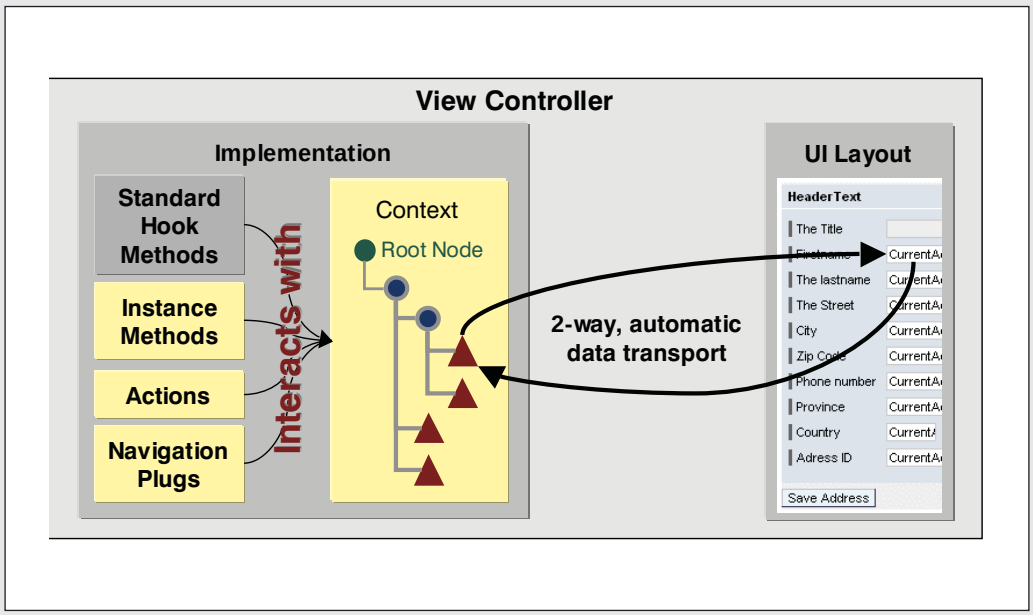


Figure 16 The view controller context decouples the application coding from the UI elements

The power of the data binding principle lies in its simplicity. There is no reason why multiple UI elements could not have their `readOnly` properties bound to the same Boolean value. In this manner, using just a single line of code, you can enable or disable a whole set of UI elements without needing to access each UI element object individually.

In **Figure 16**, you can see how the UI elements have been decoupled from the application coding by means of the view controller's data storage area, known as the "context." The application coding on the left interacts with the context, and the UI elements on the right have their various properties bound to the nodes and attributes found in the context.

Unfortunately, many developers fail to understand this principle, and so add coding directly into the `wdDoModifyView()` method to manually set or get UI element property values. This coding is not "wrong" insomuch as it is functional and achieves the desired result. But it is considered very poor style because it is quite redundant and therefore does little more than add unnecessary complexity to the application.

Conclusion

I trust that by now, you understand what can go wrong during a Web Dynpro implementation and, more importantly, *why* things go wrong and how to avoid them.

Before embarking on a Web Dynpro implementation, please take every reasonable step to ensure that you avoid making the mistakes described here. Here's a quick checklist.

Tackle the misunderstandings!

Web Dynpro is *not* "just like any other MVC-based development toolset."

Due to SAP's specific design criteria for Web Dynpro, certain flexibilities present in other UI development toolsets have been removed — such as pixel-perfect placement of UI elements.

Web Dynpro has been designed to create business applications that:

- Have a medium- to long-term life expectancy
- Are aimed at a generic or loosely defined target audience
- Offer a high degree of business process flexibility
- Place the highest priority on stability and reliability of the business process
- Place a lower priority on specific features of the UI

Education, education, education!

Make sure all the developers on the project have attended the standard SAP training courses for Web Dynpro for Java.[14] Also, make sure that there is at least one copy of the SAP PRESS books on Web Dynpro generally available to all project members. These are *Inside Web Dynpro for Java* (Second Edition)[15] and *Maximizing Web Dynpro for Java*.[16]

Web Dynpro implementation costs are directly related to the developer's level of understanding:

- Little or no training = increased TCO due to excessive and/or redundant coding complexity. In extreme cases, the maintenance costs can become so high that it is cheaper to throw the entire application away and rewrite it.
- Well trained = significantly lower TCO due to the application containing only the coding required to achieve the business purpose. This in turns lowers the most significant cost incurred during the lifespan of a piece of software — maintenance.

Do not abuse view controllers!

- A view controller should only ever act as a consumer of data — either from a non-visual controller or the user (via the keyboard and mouse).
- If you are writing code in a view controller that interacts with a back-end system, it will probably work — but you are not following good Web Dynpro design principles.
- Coding to interact with back-end systems belongs only in non-visual controllers.

Practice good project management!

- Make sure the implementation project is managed by someone who has already been through at least one Web Dynpro implementation.
- Flexibility of scope is fine as long as it does not happen in an uncontrolled manner. If the project's scope needs to be altered, then make sure that senior project management has reviewed and signed off on the variation, and that the project's resources and time factors have been adjusted accordingly to ensure the required quality levels can still be delivered.
- Plan for periodic code reviews.[17]
- Carefully scrutinize your intended implementation partner's capabilities, and don't be persuaded by a smooth sales pitch. Ask for customer references and do your own research.

When used correctly, Web Dynpro is an efficient and powerful tool for building robust and reliable business applications. If you are about to embark on your first Web Dynpro implementation, then I trust that you now have enough knowledge to enter the situation with confidence and clarity of understanding.

All the best and enjoy using Web Dynpro (correctly)!

[14] These courses are JA310 ("Introduction to Web Dynpro for Java") and JA312 ("Advanced Web Dynpro for Java").

[15] Available in English only.

[16] Available in English and German.

[17] For more on how to perform code reviews, see the article "Put Better Programs into Production in Less Time with Code Reviews: What They Are, How to Conduct Them, and Why" (*SAP Professional Journal*, July/August 2003).

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

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

In January 2003, he started working with Web Dynpro and has worked closely with the development team in Walldorf, both learning the product and writing proof-of-concept applications. The knowledge gained while working with the developers became the foundation for the book "Inside Web Dynpro for Java" published by SAP PRESS in November 2004. This book is now in its second edition (September 2007).

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

You may reach him at chris.whealy@sap.com.

