A guided tour of Java software development lifecycle management with SAP NetWeaver Development Infrastructure (NWDI)

Part 1 — Fundamental concepts

by Wolf Hengevoss

Wolf Hengevoss
Product Management,
SAP NetWeaver Development Infrastructure, SAP AG

Wolf Hengevoss studied natural sciences at the University of Kaiserlautern. He joined SAP in 1999 as a member of product management. He has worked in the Basis group, focusing on topics such as the Computer-Aided Test Tool and Business Address Services. Since the early stages of SAP Exchange Infrastructure (now known as Process Integration), he has been working on the Java environment. Currently, his focus is on the rollout of SAP NetWeaver Development Infrastructure.
You may reach him at wolf.hengevoss@sap.com.

Java developers have long enjoyed the freedom of developing, testing, and managing their source files and archives locally on their PCs, relying solely on centralized source code control systems for change management. In an enterprise environment, however, with developers working in different locations (and potentially on the same code at the same time), this model quickly becomes a nightmare to manage. For example, consider the case in which you need to apply an SAP support package to an SAP CRM application that you have modified: You need a way to easily identify which files have been modified by SAP and which have been modified by you to avoid potentially breaking the application as you apply the changes in the support package.[1]

Figure 1 on the next page summarizes the typical Java development process. If you've ever participated on a Java development team, this process should look familiar. Perhaps you can also identify with the common pain-points highlighted on the left side of the figure:

- **Difficulty locating source versions:** This problem occurs particularly when you need to "go back in time" to maintain older versions. Also difficult is coordinating work on multiple versions of a single object in parallel (e.g., to simultaneously make bug fixes and add new features) to avoid associated build conflicts.
- **Difficulty finding and tracking foreign libraries:** Most J2EE vendors do not include automated tools for locating the sources or foreign libraries developers need for a project. Even when using third-party source control systems, developers may take quite a bit of time to find the right source versions — e.g., most approaches to versioning make you navigate through all the branches of your solution to locate particular files associated with a specific release level. And this will only

[1] The modification concept, well known to the ABAP world, is now introduced for Java with NWDI.

No portion of this publication may be reproduced without written consent.

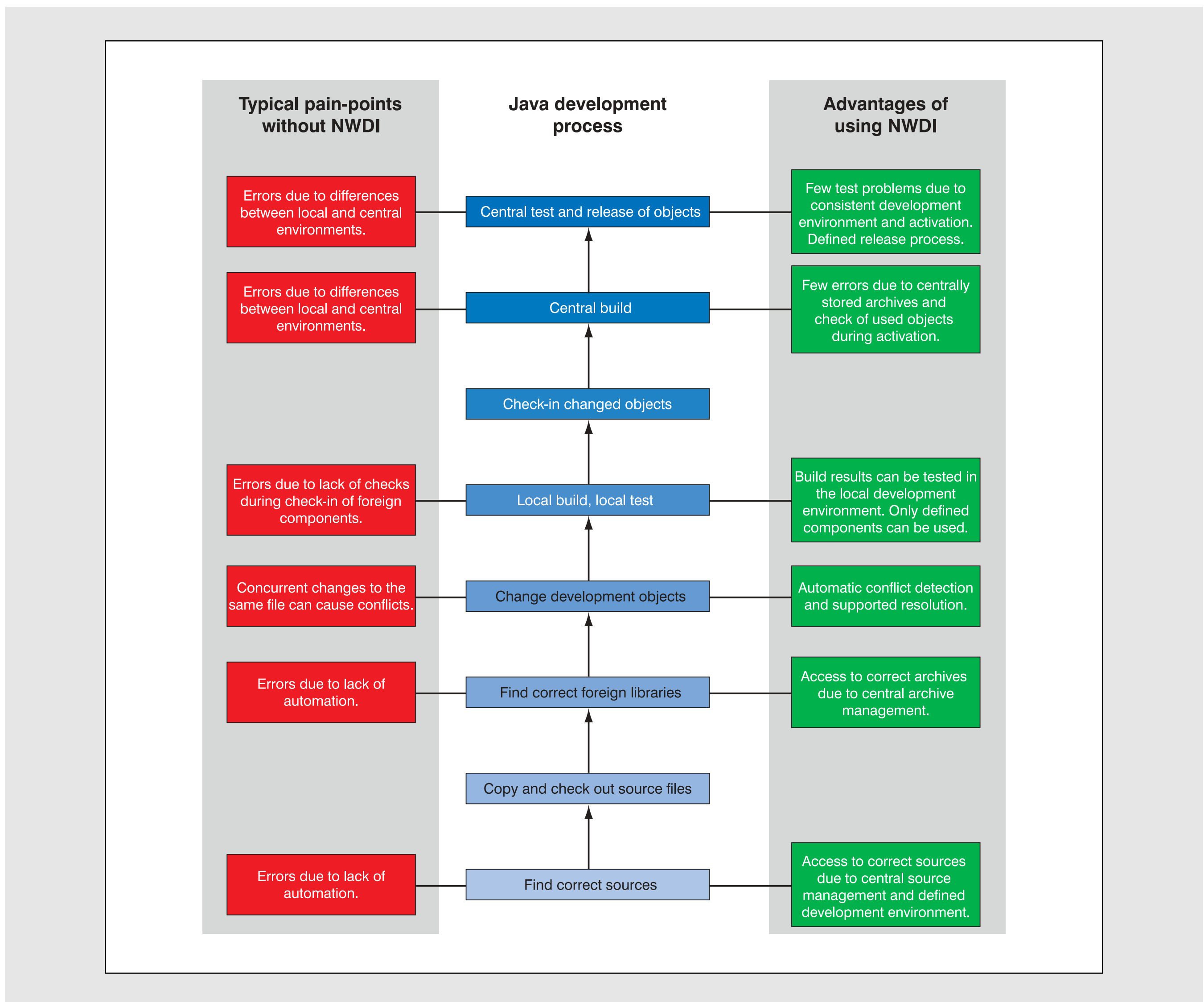

Figure 1 Typical enterprise Java development process, without and with NWDI

work smoothly if you *and* all your team members manage to get the correct versions!

- **Discrepancies between local and central environments:** Often archives being referenced or reused are changed on the central server while local development is occurring. Developers often build and test their components against outdated copies of referenced components, resulting in downstream build problems on the team server. Even if developers refresh local copies before doing a local build, changes in used foreign archives can still occur between the time the local build occurs and the central nightly build.

- **Clashes when multiple developers work on the same object at the same time:** Changes to different versions of the same object (e.g., maintenance and upgrade) made by developers can often collide during development or after modification and update, especially during the build process.

Each of these pain-points is a symptom of a common problem: the lack of an enterprise-scale

 ©2007 SAP Professional Journal. All rights reserved.

Java development platform, designed for potentially hundreds of developers working in parallel on thousands of interrelated development components.

SAP NetWeaver Development Infrastructure (NWDI for short)[2] offers a solution and brings many of the proven, world-class change management capabilities of ABAP to the Java world.[3] NWDI addresses all of the major pain-points identified previously, as shown on the right side of **Figure 1**:

- **Sophisticated archive management:** Perhaps the greatest benefit of NWDI for Java teams is a near elimination of (costly) failed builds (usually due to changes to objects that reference each other). NWDI maintains a single, active baseline of components that are almost guaranteed to build successfully. It protects the integrity of this baseline by only activating objects that build successfully against the existing active state, which is stored in a central "buildspace." For any single build of a component, the system sets up a temporary area in the local file system where it can safely "try" to build new or changed objects without impacting the baseline. The central buildspace therefore provides consistent access to the right archive versions, for the whole team, at any given time.
- **Enhanced source version control and search capabilities:** During software development, NWDI stores "pointers" to the software's source files (to the software's codelines, to be more precise) in "workspaces," which NWDI displays as folders in a hierarchical file structure. Each workspace represents a version of a piece of software, so you can easily access the software's versions to locate an old release you need to maintain, for example. Also, because workspaces contain pointers rather than the actual sources, you can reuse sources across workspaces (if certain codelines are unchanged between versions, for example) without copying them. NWDI also automatically filters unnecessary files from view, making it easy to find the files you need.
- **Developer-initiated, incremental build capability:** NWDI empowers developers to initiate incremental build requests (versus waiting for central nightly builds), ensuring real-time detection of build issues. Developers can request builds (the addition of changed versions to the active baseline) at will by attempting to activate their changes. If the system detects any conflicts with the active set of objects, the system will notify the developer immediately and leave the active baseline (used by the team) unharmed.
- **Parallel, multi-developer application development:** Developers working on various pieces of a development effort can build and consolidate their work without colliding, including leveraging (or extending) any number of preexisting software components. NWDI manages the dependencies between software components, and ensures that their code is always being tested against the appropriate population of dependent components to work properly in production.
- **Simultaneous development on a single component:** Developers can work on different versions of the same code at the same time. For example, one developer can make bug fixes to the "live" version of a software component, while another developer works on adding "next release" features to the component. NWDI ensures that these versions remain completely isolated.

This two-part article series teaches you the fundamental concepts you need to know to begin building, deploying, and managing changes to your Java applications with NWDI. In this first installment, I introduce critical terms and concepts you can't do without when working with NWDI, and then I step you through the revised Java development process you'll follow. In the next installment, which will appear in an upcoming issue of *SAP Professional Journal*, I'll show you how to directly apply this

[2] NWDI was first introduced with SAP NetWeaver 2004 under the name Java Development Infrastructure (JDI).

[3] SAP wants to fill this "Java capabilities gap" because it has historically prevented SAP, its partners, and its customers from fully leveraging Java's potential in enterprise computing. The Java runtime was added to SAP Web Application Server (AS) 6.20 — renamed to SAP NetWeaver AS with the 2004s release — and since that time has seen widespread adoption. For example, it is the platform underlying SAP NetWeaver Portal, as well as some other key components of SAP solutions, such as the Internet Collaboration Hub (part of SAP SCM) and the Internet Pricing Engine (part of SAP CRM).

No portion of this publication may be reproduced without written consent.

knowledge by walking step-by-step through a practical development example. Along the way, you'll see how the development landscape is set up for the project, how dependencies between components of the application are managed, and how transports are managed in the Java environment. At the end, you'll emerge with a valuable understanding of NWDI that will benefit you as you begin to develop or support both SAP and custom Java-based development projects for your SAP NetWeaver platform. There's a lot to cover, so let's get started!

> ***Note!***
>
> NWDI runs on SAP Web Application Server (AS) Java 6.40 and higher systems. It was originally delivered with SAP NetWeaver 2004 and is now delivered as usage type DI with SAP NetWeaver 7.0 (2004s).

Key elements of the NWDI architecture

Although NWDI doesn't drastically change the methodology that Java development teams use to code software, it adds strong back-end support for setting up the development landscape and for the build, production, and maintenance of software. There are a fair number of component names, terms, and concepts you need to know to begin creating, migrating, and administering applications in this new environment. This, and the following two sections, will quickly bring you up to speed, focusing first on the key components within the NWDI landscape.

Figure 2 provides an overview of the key components of NWDI, which include the following:

1. **NetWeaver Developer Studio (NWDS)** is SAP's PC-based Java development environment within which developers build custom Java applications according to the SAP component model and also test them. NWDS is tightly integrated with NWDI, meaning that it includes menu commands for NWDI activities such as checking out source code and activating changes.[4]

> ***Note!***
>
> NWDS uses the same build tool as the CBS, letting developers build and test code locally before checking in and activating their changes on the server.

2. The **Design Time Repository (DTR)** stores the complete history of all source code for all Java objects in specific DTR workspaces. The DTR offers a Web-based administration tool, but is typically accessed behind the scenes by NWDS when the developer wants to view, check out, or check in objects. The versioning mechanism of the DTR allows concurrent development by different developers with automatic conflict detection, even in distributed scenarios, because versioning information is included with source files. This capability is the basis for a real modification concept for Java-based applications.
3. The **Component Build Service (CBS)** builds deployable Java archives and stores them in a CBS buildspace when a developer activates new or modified objects. These new versions can then be automatically deployed from the CBS to update the development or consolidation system. The CBS ensures that only one version of each development component is active at a time, and guarantees system stability by making sure that all active objects will compile without errors at the time of their activation. Developers interact with the CBS through specific functions within NWDS (e.g., check in and activate), but like the DTR, the CBS also offers a Web-based administration interface (e.g., to trace errors due to incompatible changes of single development components).

[4] For a detailed introduction to NWDS, see the article "Get Started Developing, Debugging, and Deploying Custom J2EE Applications Quickly and Easily with SAP NetWeaver Developer Studio" (*SAP Professional Journal*, May/June 2004).

 www.SAPpro.com ©2007 SAP Professional Journal. All rights reserved.

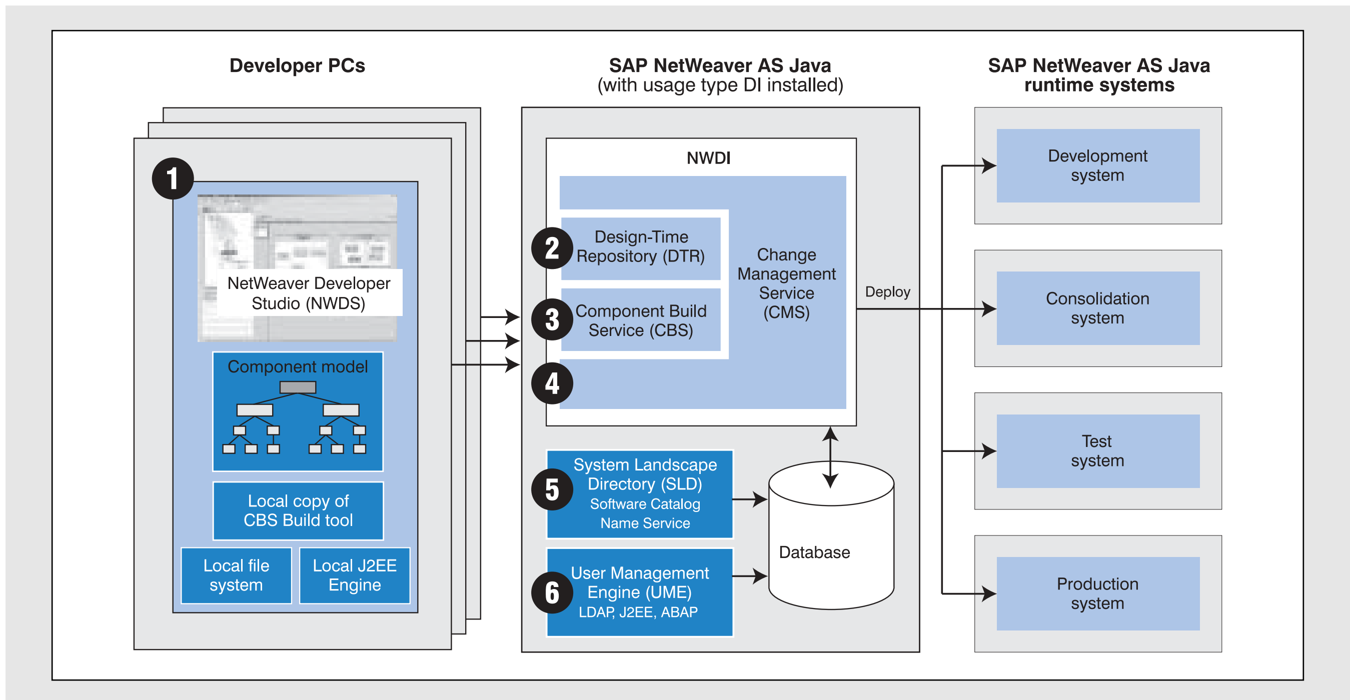

Figure 2 Key components within NWDI (DTR, CBS, and CMS)

4. The **Change Management Service (CMS)** is responsible for managing the setup of the development landscape with all required sources and archives and for the transport of these objects during the entire development cycle. The CMS centrally administers almost all the necessary actions in the DTR and CBS and uses the archives from the CBS buildspace to update connected runtime systems. The CMS is analogous to the ABAP Change and Transport System (CTS), only it enables you to set up development environments for many development teams *in the same* NWDI. The CMS enables administrators to easily manage development teams working simultaneously on multiple versions of an object (e.g., next release and production support at the same time). Administrators use the CMS's Web-based user interface to prepare for development, and NWDS accesses the CMS behind the scenes when a developer wants to view, retrieve, or check in versions of particular development components.

5. The **System Landscape Directory (SLD)** serves two important roles in the NWDI-based development process. First, it is a master catalog for all your systems, software products, and software components (more on these in the next section), as well as of the build and installation time dependencies between the software components (these dependencies are essential to the setup of the development environment). Second, the SLD contains a "name service" that ensures unique names for new development objects and compiled Java archives. It verifies that names entered by developers are not already taken. For all these reasons, the SLD is vital to the CMS, DTR, and CBS. Because the SLD is a central part of SAP NetWeaver (not technically part of NWDI), you may already have an SLD in your SAP landscape.[5]

[5] The SLD is a central data repository that is used by SAP applications such as SAP Solution Manager, Supplier Relationship Management (SRM), Auto-ID Infrastructure (AII), and Process Integration (PI), which was formerly known as Exchange Infrastructure (XI), to access information on systems and solutions deployed in the system landscape. For a detailed introduction to the SLD, see the article "A system administrator's practical guide to SAP System Landscape Directory (SLD)" (*SAP Professional Journal*, November/December 2006).

No portion of this publication may be reproduced without written consent.

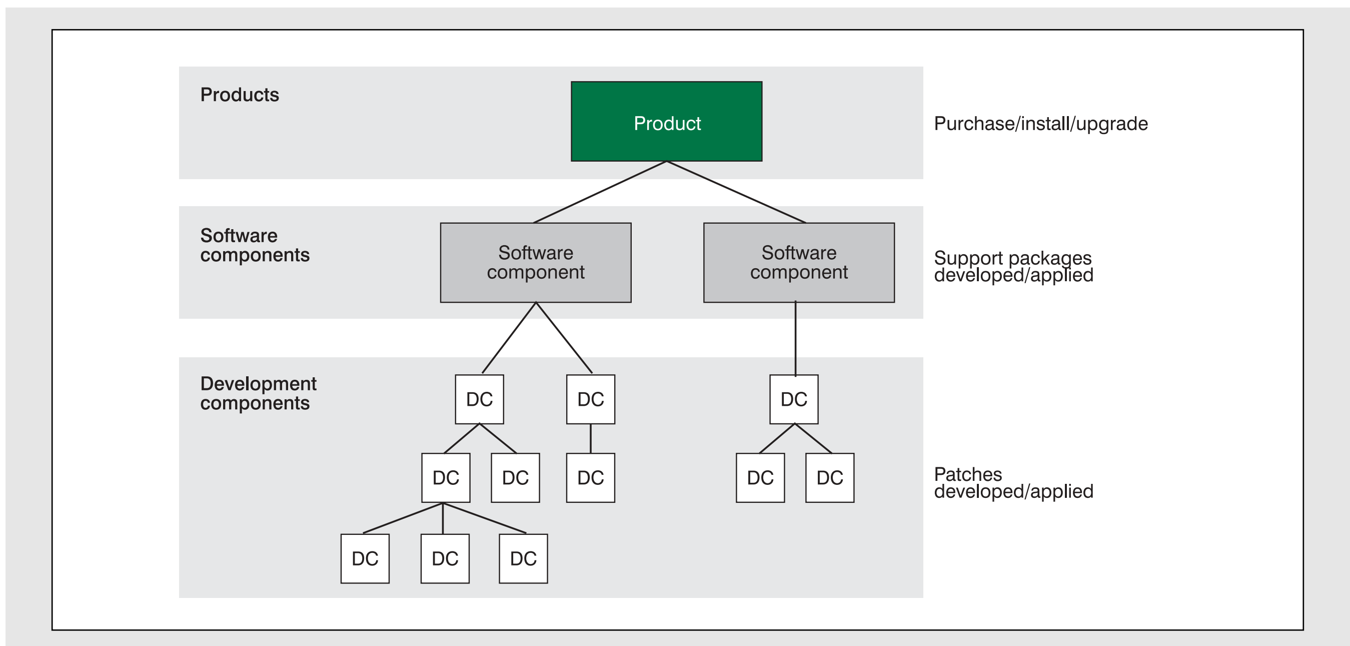

Figure 3 The SAP component model — products, software components, and development components

6. The **User Management Engine (UME)** is the Java analog of the ABAP runtime's user management and authorization functionality — it provides tools with which to assign user IDs and passwords for users of your Java applications, assign permissions to particular applications/features/functions, and program an API to check a user's authorizations from within your Java programs. UME data can reside in an LDAP directory, a local J2EE database, or an ABAP system.

There are two other very important things to notice from **Figure 2**: First, NWDI only needs to be physically installed *once* in your landscape. All of your Java development and consolidation runtime systems (which, for most companies, reside on the same physical server) and your test or production runtime systems are simply targets, running on their own servers (and even their own releases). The Java archives generated by the CBS are deployed to these targets using the CMS Transport Studio (a tool you'll see in action in the next installment of this two-part article series).[6] Although NWDI doesn't host these runtime systems, one NWDI can manage a number of them, even if they are on different releases. Second, NWDI should be installed on its own system, not on your development system. This is very important because you want to isolate your NWDI system from the instability, overall transient nature, and maintenance/upgrade schedule of your development system. This is why NWDI is shown as a separate system in the figure.

[6] This holds true even if you develop and maintain software for different target releases — such as SAP NetWeaver 2004 and 2004s — because the NWDI server is independent of the software version it hosts.

Essential NWDI programming constructs — the SAP component model

Now that you understand the tools and elements you'll be working with, let's turn our attention to the three units into which all Java applications — both custom-built and SAP-provided — are divided in NWDI (see **Figure 3**):

- Products
- Software components

 ©2007 SAP Professional Journal. All rights reserved.

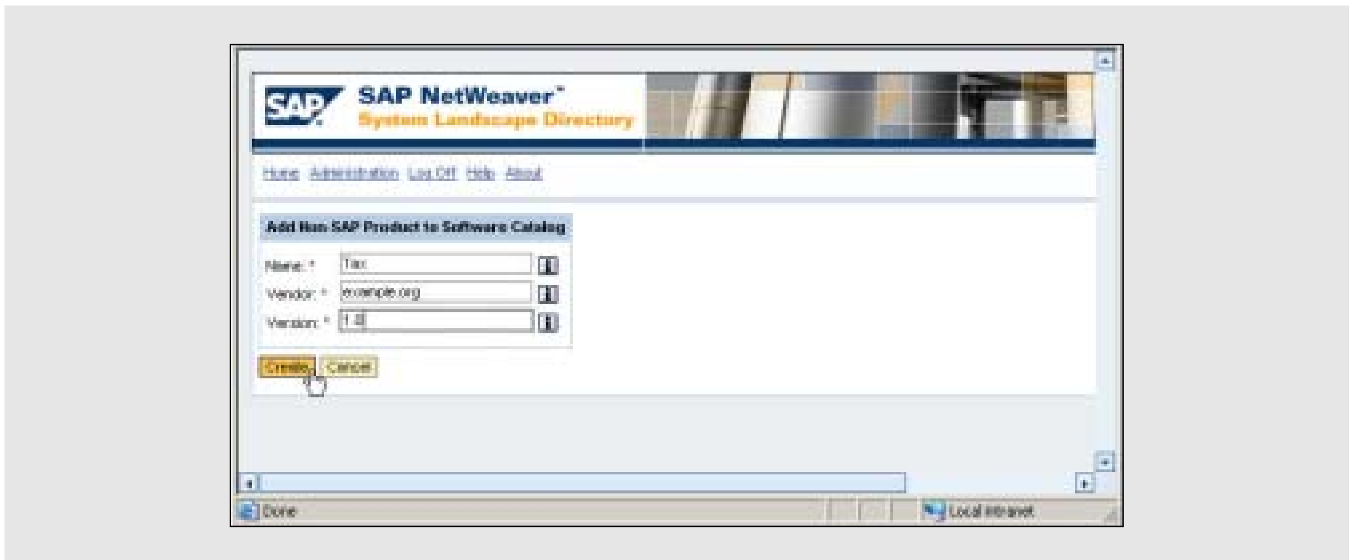

Figure 4 Defining a Tax product in the SLD

- Development components

These three components types form the SAP component model upon which all NWDI software development is based. It supports a strong reuse concept and puts into practice an easy way to define a complete product that can easily be reproduced. Let's take a closer look at each of these key building blocks.

Products

A software product (or just "product") represents a business solution and serves as the root of all development. A product is made up of any number of software components. From a deployment and maintenance perspective, products are the level at which software is purchased, installed, or upgraded.

The first development step in the SLD is defining a new product — more specifically, defining a new *product version*, such as version 1.0 of a new Tax product, as shown in **Figure 4**. An NWDI administrator, in consultation with the software architect, should define the product (I summarize the different roles involved in the development process a little later in the article when I walk you through the new development process).

Software components (SCs)

A software component bundles together technical objects (development components) that relate to each other (e.g., a set of development components that relate to tax computation). Software components are always assigned to one or more products, and like products, have versions. A software component can also use another software component if a dependency is defined between them, and it can be reused among different products. From a software maintenance perspective, software components are the level at which support packages are developed and applied.

Software components first must be defined in the SLD — again, by an NWDI administrator in consultation with the software architect — and associated with a product. **Figure 5** on the next page shows the definition of an example software component called TECHNOLOGY, which is a software component of the Tax 1.0 product.

After a software component is defined in the SLD, it can be used in the CMS for configuration and development. **Figure 6** on the next page shows what the software components of the Tax product look like in the CMS Landscape Configurator tool (we'll look at this tool in detail in the second installment of this article series). The development environment (the

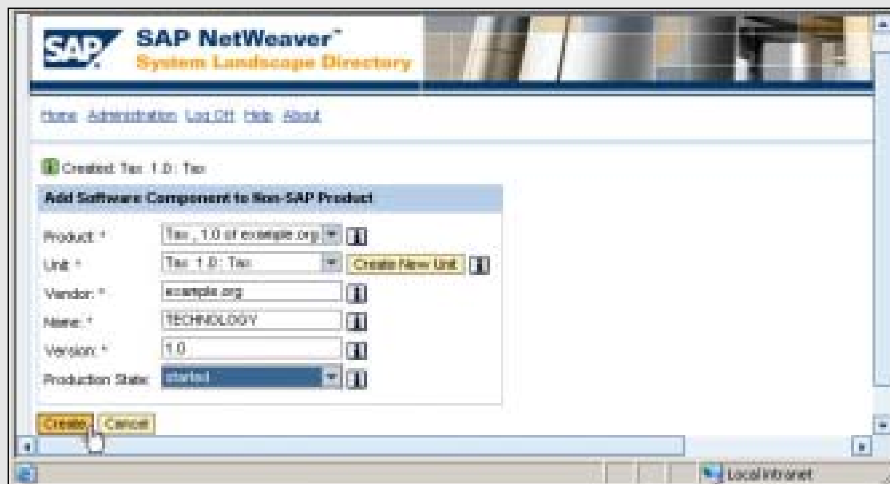

Figure 5 The TECHNOLOGY software component of the Tax product created in the SLD

Software Components (SC)

Software Components for Development

Add SC ... | Delete Table Entry | Select All | Exclude All | Include All | View/Edit XML Content ... | Load SC Configuration ...

Software Component Name	Vendor	Release	Package Type	SC State	Exclude from Deployment
TECHNOLOGY	example.org	1.0	Source and Archive		☐
APPLICATION	example.org	1.0	Source and Archive		☐

Row 1 of 2

Show: 0 additional rows

Required Software Components

Add SC ... | Delete Table Entry | Select All | Exclude All | Include All | View/Edit XML Content ...

Software Component Name	Vendor	Release	SC State	Exclude from Deployment
SAP_JTECHS	sap.com	7.00		☐
SAP_BUILDT	sap.com	7.00		☐
SAP-JEE	sap.com	7.00		☐

Row 1 of 3

Show: 0 additional rows

Figure 6 Software components to be used in the development of the Tax product

  ©2007 SAP Professional Journal. All rights reserved.

DTR workspaces and CBS buildspaces) will be configured according to the settings shown here — that is, the software components displayed here will be available to developers working on the Tax product application. In the figure, you can see that two software components (TECHNOLOGY and APPLICATION, shown in the upper portion) will be newly developed, and that the functionality contained in three existing software components (SAP_JTECHS, SAP_BUILDT, and SAP-JEE, shown in the lower portion) will be used in the development of the two new ones.

Development components (DCs)

After a product and its software components have been defined in the SLD — and a development configuration, which defines the components available for development, has been imported into NWDS (more on development configurations in an upcoming section) — developers can start creating development components. A development component is a container for actual Java objects — Java classes, Java Server Pages (JSPs), Enterprise Java Beans (EJBs), etc. — and is the unit with which Java developers work. Development components are always assigned to exactly one software component and can be leveraged by other development components. From a software maintenance perspective, development components are the level at which patches are developed.

> ***Note!***
>
> Development components define the structure of your application, and their development must be considered carefully — ideally, developers should design them in collaboration with a software architect (I discuss the different roles involved in NWDI development later in the article).

Developers create development components within NWDS by creating a project. **Figure 7** shows the development component project wizard, where you specify the owner (or "vendor") of the project; the

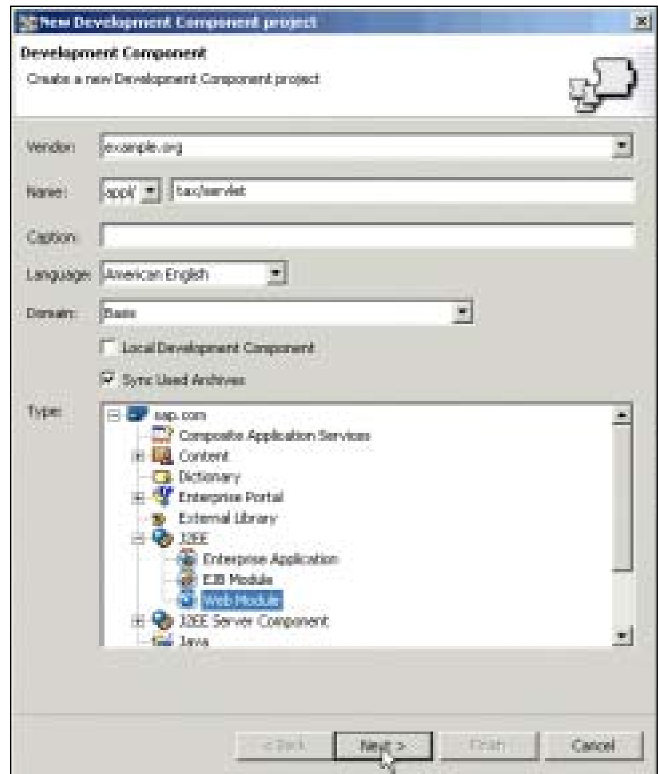

Figure 7 Creating a new development component project

name of the project, which defines the hierarchical structure of the workspace folder that will contain the development objects in the DTR; and the type of the project, which is used to automatically generate the structure of the project and to automatically apply the correct build script (I go into detail on this in the second installment of this article series).

Development components are really the core of the new SAP component model. They are analogous to the "project" concept used in most other Java integrated development environments (IDEs) — they contain the development objects that make up the project and store all of the important metadata for a project, including the interfaces a development component exposes to other development components, and the dependencies a development component has with other development components (I go into more detail on this in the second installment of this article series). This metadata is the basis for the build process triggered by the developer on a component level.

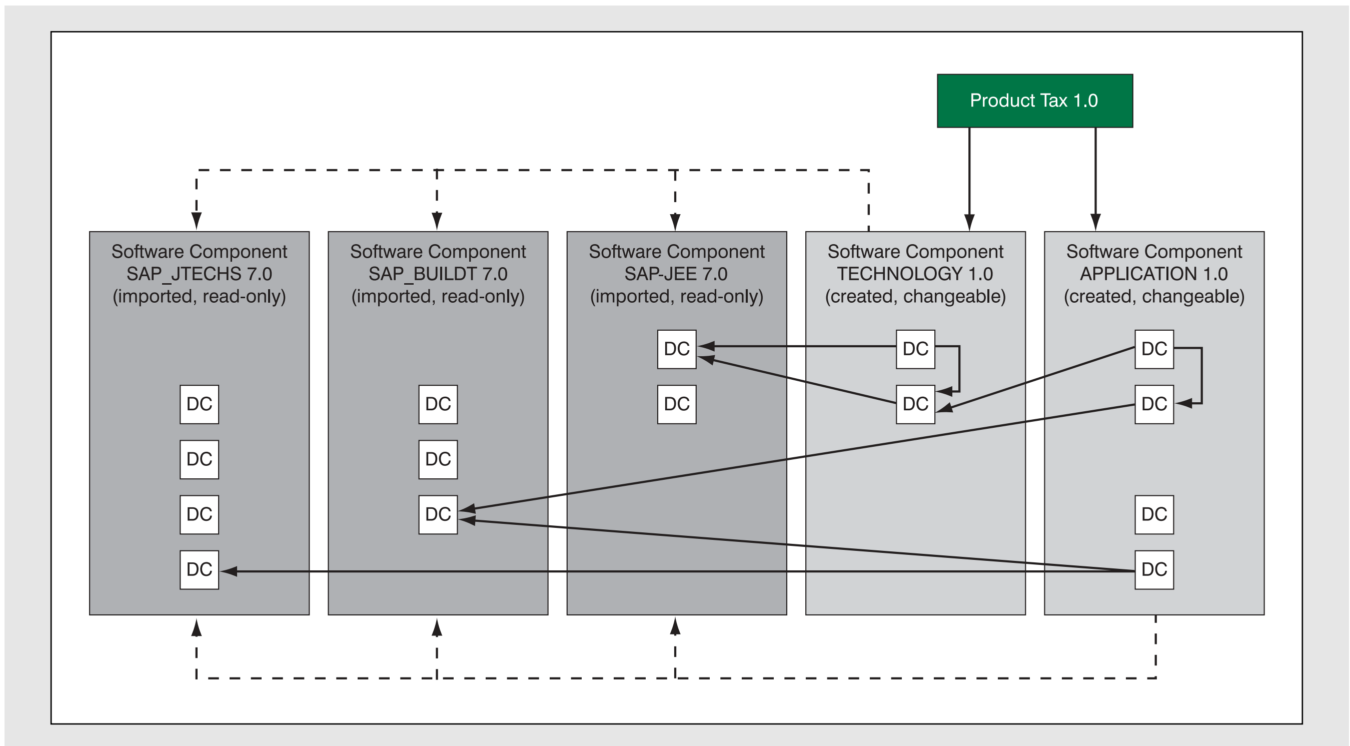

Figure 8 The architecture of the example Tax application

This three-tier division of applications (products into software components, software components into development components) facilitates team development and code reuse, enhances maintainability, and enables a structured development process — as demonstrated by the example screenshots you just saw, the NWDI tools are implemented in such a way that you naturally follow a manageable, organized structure.

Note!

Don't worry: These concepts will not change any of your Java coding. You still code with Java classes, JSPs, EJBs, or your technology of choice. Development components simply serve as a container into which these objects are placed for change management and software logistics with NWDI, and they provide metadata to the project on interfaces and dependencies.

How all the units work together — the example application

Figure 8 shows how all of these units come together in the example Tax application, which we'll walk through in detail in the next installment of this article series. Let's briefly review its components.

As you can see, the development team's task is very well defined — to develop the first version (version 1.0) of the product Tax, which is a Web-based tax calculation application. The product will consist of two software components that will be created — TECHNOLOGY, which will contain the tax calculation functionality, and APPLICATION, which will contain the user interface. Because these software components are intended for use with version 1.0 of the Tax product, they also have the version 1.0. APPLICATION will use TECHNOLOGY to provide the tax calculation functionality through the user interface, which means that the APPLICATION software component has a "dependency" on the TECHNOLOGY software component. Because these

 www.SAPpro.com ©2007 SAP Professional Journal. All rights reserved.

two components will be developed, they are changeable, meaning that developers can modify them.

Three existing software components provided with SAP NetWeaver 7.0 — SAP_JTECHS, which contains basic Java functionality, SAP_BUILDT, which contains build information, and SAP-JEE, which contains parts of the SAP J2EE Engine API — will also be imported for developers' use. All the functionality in these three components will be available for use in the development of APPLICATION and TECHNOLOGY, but the SAP-provided components themselves are read-only, meaning that developers cannot modify them during the development of the Tax product. Because version 1.0 of the Tax product is intended for use in an SAP NetWeaver 7.0 (2004s) environment, this version (7.0) is required for the three SAP-provided software components.

> ***Note!***
>
> SAP_JTECHS, SAP_BUILDT, and SAP-JEE contain libraries of basic functionality that are required for all Java-based development in SAP NetWeaver 7.0 (2004s).

As you can see in the figure, each software component contains a number of development components that provide the software component's functionality (e.g., the tax calculation functionality is provided via Java classes in TECHNOLOGY and the user interface is provided via JSPs in APPLICATION). The development components can be used by those in other software components to reduce the risk of incompatibility problems (e.g., APPLICATION uses the ejb20 development component in SAP_JTECHS 7.0, which is guaranteed to be compatible with the 7.0 version of the target runtime system) and to enhance the developers' productivity (a Java Connector development component in SAP-JEE, which allows data exchange with an ABAP back end, is used in TECHNOLOGY).

In the next installment of this article series I will go into detail on the steps involved in creating this example. Here, I just wanted to provide you with a "big picture" of how the units work together to create a full-fledged application.

Now that you understand the key elements and units you'll be working with to create applications, let's take a look at some of the new administrative concepts that have been introduced with NWDI.

Essential NWDI administrative constructs

Figure 9 on the next page summarizes the four key administrative terms you'll encounter within the NWDI tools and documentation. Beginning at the highest level and drilling down, they are:

- Tracks
- Development configurations
- Workspaces
- Buildspaces

Let's take a closer look at each in turn.

Tracks

A track defines the entire development environment: the software components of a product (both those to be developed and existing ones that are required to develop the new ones), the DTR workspaces and CBS buildspaces, and a predefined series of systems through which an application will pass on its way to deployment. A track can be thought of as a container with "slots" that represent the series of "states" that an object can pass through (development, consolidation, test, and production).[7]

For systems that permit code changes (develop-

[7] Development, consolidation, test, and production are "phases" during development. The runtime systems used in each phase are optional. For example, a vendor that does not want to run its own software productively (because it doesn't have a manufacturing system for which it provides software) does not need a production system. This is in contrast to ABAP, where separate physical systems are set up for development, test, and production purposes.

No portion of this publication may be reproduced without written consent.

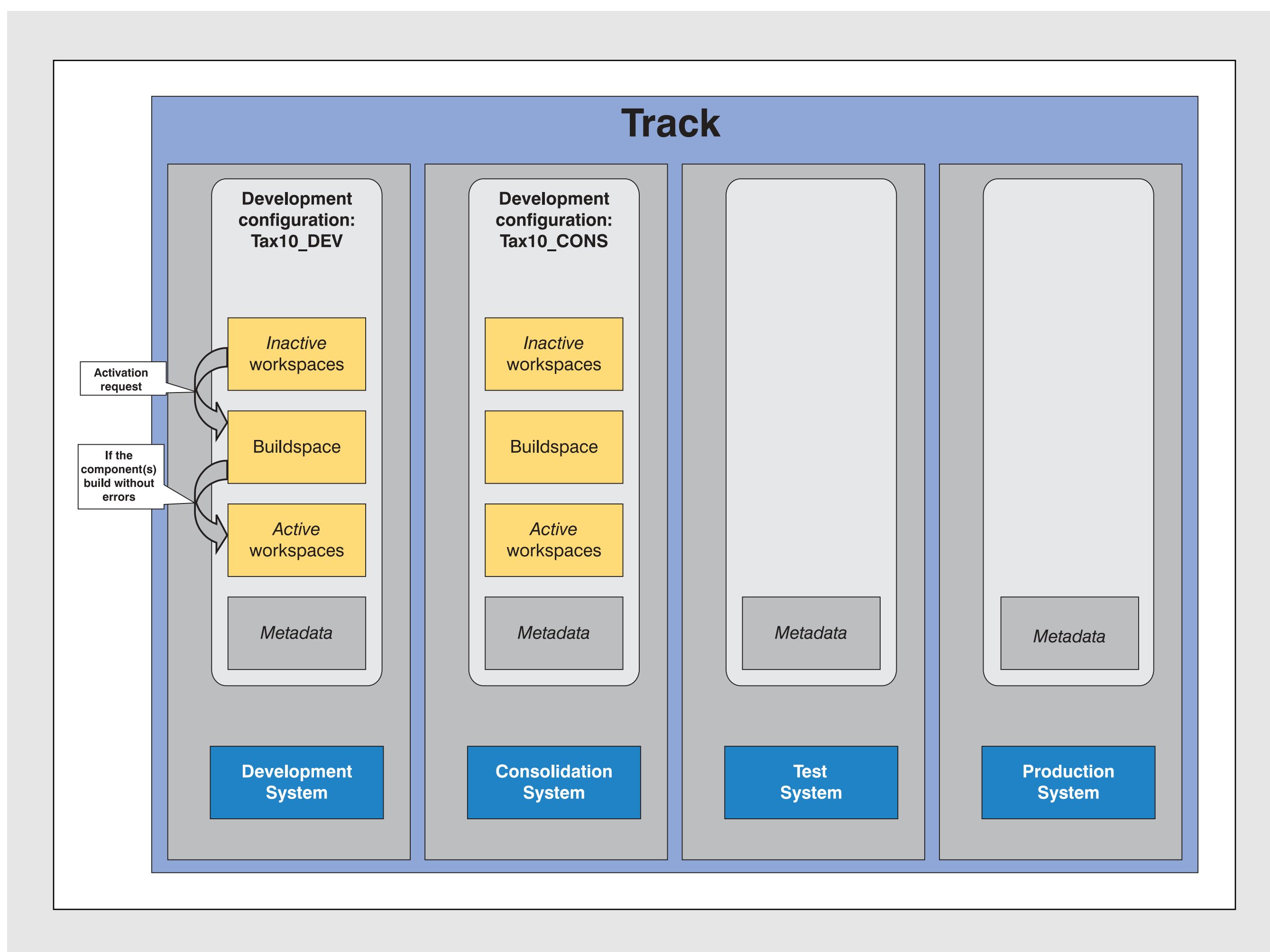

Figure 9 Key NWDI administrative constructs

ment and consolidation, a "development configuration" is automatically generated. Development configurations contain DTR workspaces for the sources and CBS buildspaces for the archives (more on development configurations in a moment). Developers must import these development configurations into NWDS to begin development work. In contrast, deployment-only systems (test and production) simply store the information needed for deployment into their runtime systems. These systems do not have workspaces and buildspaces — software is never changed directly in them. Each track also includes the system's location and whether sources should be included in the resulting Java archive files (the latter on a software component level).

Figure 10 shows the types of settings you'll encounter when defining a track for a product on the Track Data tab in the CMS Landscape Configurator. As you can see, you specify the repository type, which defines whether you want to use a DTR for Java-based development (the setting you will typically use) or create a track to manage Exchange Infrastructure (XI) — now known as Process Integration (PI) — transports. You also define the track ID, name, and description, which are used to identify and name the track and resulting workspaces and buildspaces, and the URLs for the DTR and CBS, which define where the workspaces and buildspaces will be automatically generated.

You select the types of system roles you want to

 ©2007 SAP Professional Journal. All rights reserved.

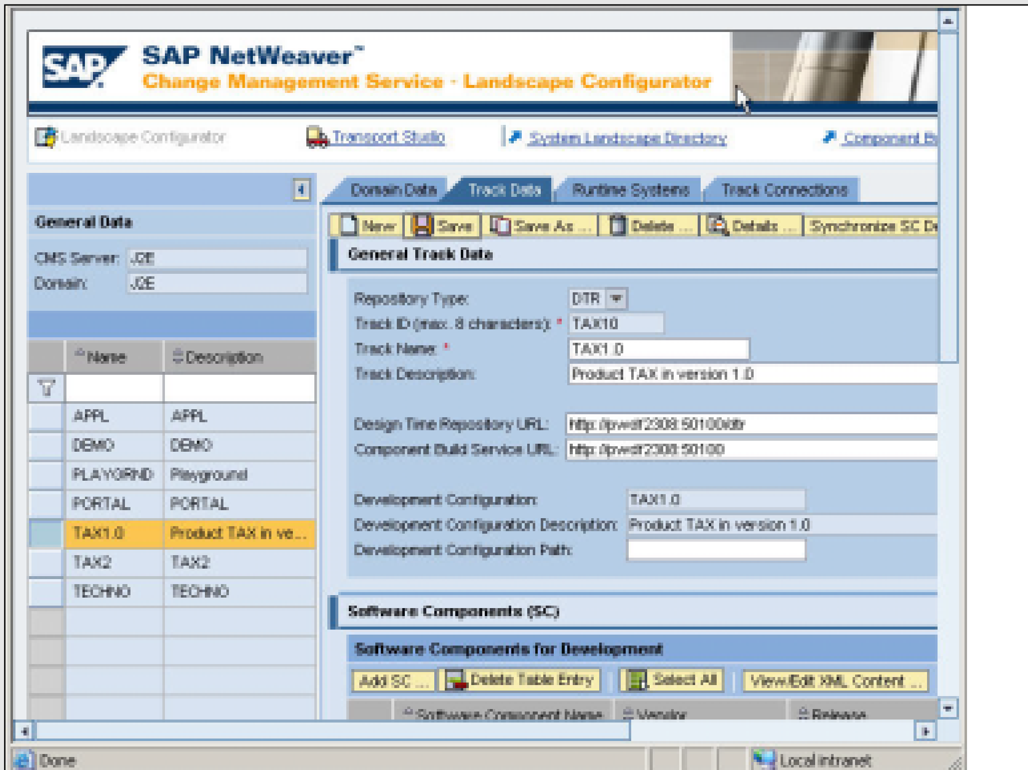

Figure 10 Defining a track

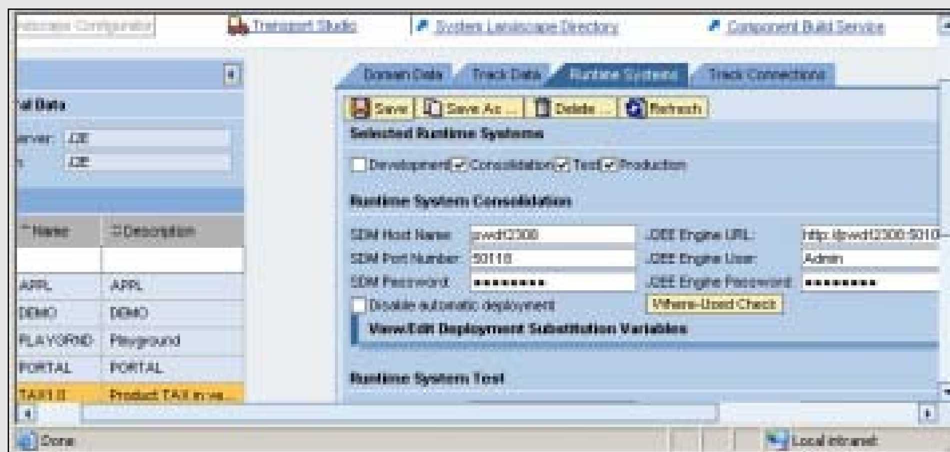

Figure 11 Specifying the runtime systems to be used

include in the track (development, configuration, test, production), and specify connection information for each system on the Runtime Systems tab in the CMS Landscape Configurator, shown in **Figure 11**.

Development configurations

A development configuration is an XML-based description of a system in a track where coding is changeable — that is, either a development (dev) or consolidation (cons) system. A development configuration houses three key pieces of data needed to automatically configure NWDS to access the systems, sources, and archives to enable developers to perform their duties:

- The location of the CBS, DTR, and name service, which are expressed as URLs[8] where workspaces and buildspaces will be generated
- The software components that are part of the product being developed (whether they are existing components that will be leveraged or new components that will be developed), including the dependencies between the software components, which the CMS must take into account during preparation of the systems in the Transport Studio and which the CBS build (activation) process requires
- Runtime systems for the deployment

Development configurations can be thought of as "guiding" the work developers do by predefining specific software components to which the developers can append development components, and by giving them restricted access to existing software components they may need to reference in their projects. As mentioned previously, development configurations are what developers actually import into NWDS to configure the access to NWDI for the development process.

You can view all the imported development configurations for an NWDI development landscape by opening the Development Configurations perspective in NWDS. **Figure 12** shows the imported development configuration for the example Tax product. The Active DCs view displays all of the software components of the product (both those that can be changed and those that are read-only) with their activated content — that is, all of the software components that are contained in the active DTR workspace and the CBS buildspace (more on workspaces and buildspaces in the next sections). In the example, the read-only SAP_JTECHS software component is expanded, so that you see the development components it contains. The Inactive DCs view displays only those software components that can be changed — that is, the software components that are contained in the inactive DTR workspace — along with the built state of the software components in the CBS buildspace. The Inactive DCs view is where you create development components within the displayed software components. This example shows you a first glimpse into a development component's structure: You can see the metadata of the tech/tax development component contained in the TECHNOLOGY software component. Other views that you can see are the Properties window and the DTR Console.

[8] In a typical setup, the DTR and CBS run on the same host as the CMS, and the URLs are filled in automatically. However, NWDI is easily scalable: In a very large landscape with multiple DTRs and CBSs, you can specify the particular ones you want to use by manually entering the appropriate URLs.

Note!

The source versions displayed in the Active DCs and Inactive DCs views are identical after successful activation, because the contents of both views are based on the central buildspace and the corresponding workspaces will point to the same database objects. Once you make a change to an object in the Inactive DCs view, the contents will no longer be identical until you build and activate the changed object.

Workspaces

If development configurations can be thought of as guiding the work developers do (that is, identifying the specific component versions they'll be working on and the NWDI services they'll use), workspaces are where developers actually *do* the work. A workspace contains the file versions for a software component in a specific track. Only software components under

 ©2007 SAP Professional Journal. All rights reserved.

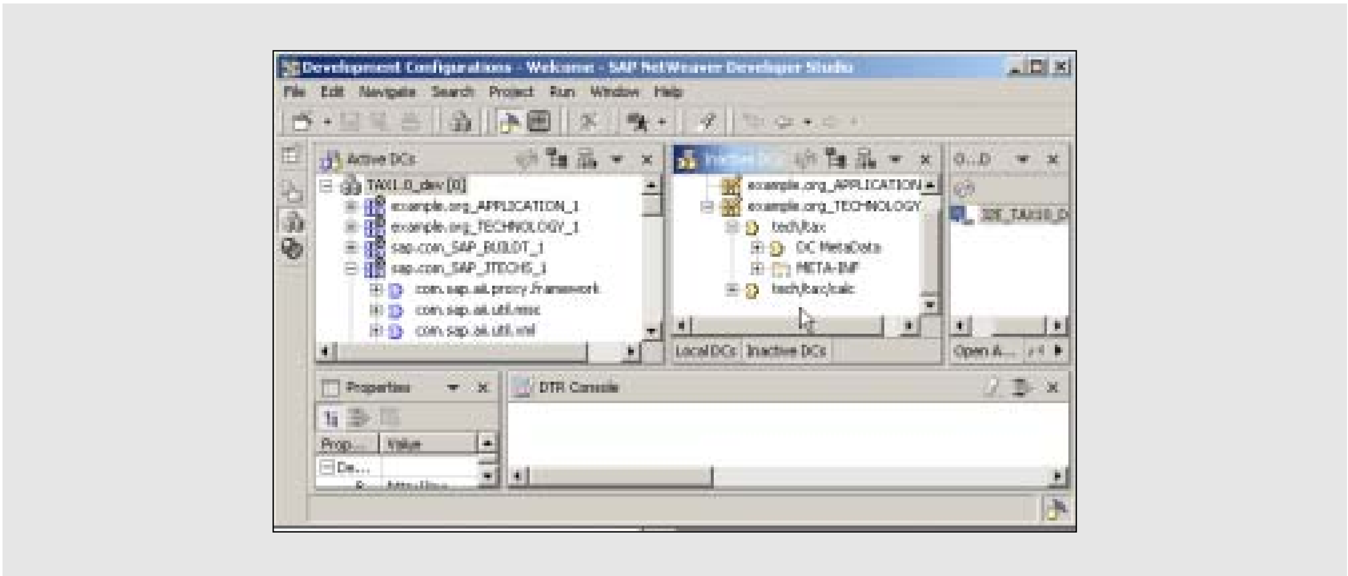

Figure 12 Imported example development configuration

development are present in a DTR workspace.[9] These can be preexisting components developed by SAP, a third-party,[10] or in-house, or they can be new components that the developer is currently working on. Because only a single version of a given component can be placed on a developer's workspace at any given time, old versions are stored in the component's history once the new version is checked in. You can display an old version at any time by directly selecting it from a Version Graph view, which displays the version history (you'll see how to do this in the next article). DTR workspaces can be used as "snapshots" of a project at a given state — for example, of the stable, published-to-production version of the application, of the version being tested in consolidation, and of the version in which development is being done.

By default, the system generates two workspaces for each SC under development in each development configuration (**Figure 9**): An "inactive" workspace, in which developers develop source code or make modifications to existing objects, and an "active" workspace, into which sources are included for use by all developers after they are successfully built in the CBS — these successfully built objects are known as "activated" objects.[11] This approach to separating

> ***Note!***
>
> The automatic definition of inactive and active workspaces is a great convenience to developers.[12] Developers can open the Active DCs view to access activated sources in the DTR workspace and safely use foreign archives without compilation or runtime problems, because only objects free of these issues are activated.

[9] These workspaces are defined on the server and accessible to the whole team. DTR workspaces are different from Eclipse workspaces, which reside on the developer's PC.

[10] A third-party component can be used if it is delivered with sources; if not, it can be put directly into the CBS buildspace wrapped in an "external library" development component type, so that its functions can be used, but not changed.

[11] Active (a.k.a., activated) components are ones that have been successfully validated and built into Java archives. As a part of the activation process, interactions and interfaces of a component are validated against the "latest" compiled versions of dependent components.

[12] If you need a place to store non-IDE items, such as a product specification in a Microsoft Word document, you can simply add a workspace to the DTR using whatever naming convention you want (because there is no activation involved, there is no need to name the workspace "active" or "inactive").

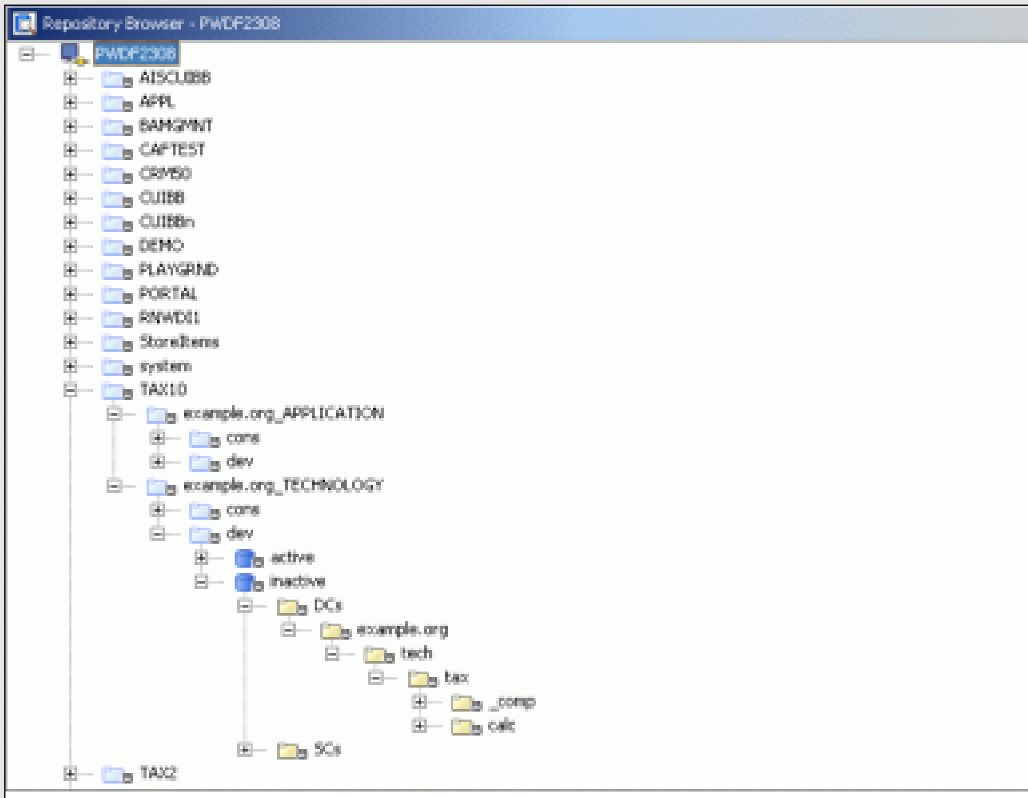

Figure 13 Example workspaces containing the Tax product sources in the DTR Repository Browser

active from inactive objects helps the system keep "known-good" versions of objects separate from objects under development.

You can view the files stored in the DTR by opening the Design Time Repository perspective in NWDS. **Figure 13** shows the Repository Browser view, which displays the DTR content of the NWDI server (PWDF2308) used for the example project. For each software project, there is a folder with the associated track's name. TAX10, the track name for the example project, contains a folder for each software component developed in that track (APPLICATION and TECHNOLOGY). Each software component contains the cons and dev phases, which contain the workspaces (remember that only development and consolidation systems which permit code changes, contain workspaces). As you can see, the inactive workspace, where new versions are created, contains the software components and development components needed for the track. To work on a component in a workspace, you simply go to the Inactive DCs view in the Development Configurations perspective in NWDS

Note!

The DTR perspective in NWDS is used for only very specific actions, such as conflict resolution. I use it here in **Figure 13** to show you the entire unfiltered view of DTR content in the Repository Browser, which can be used by administrators to manage adjustments across workspace boundaries. All the DTR functions needed in everyday work are integrated into the other NWDS perspectives.

 ©2007 SAP Professional Journal. All rights reserved.

(**Figure 12**). Once you have completed your work and are ready to move the component to the active workspace, you simply “activate” it, which is when the CBS build process (discussed in the next section) comes into play.

Technically, what the workspace really contains are references to objects in the database, so using one object in more than one DTR workspace does not mean you are copying it, and there are no unnecessary duplicate objects in the database. In **Figure 13** you can also see that a single DTR can host any number of tracks containing different product versions with multiple DTR workspaces. With the content of track TAX10 expanded, you can see a second product version (TAX2), which contains its own set of workspaces. With this architecture, you only need one NWDI for many development projects, including support for development, deployment, and maintenance.

Note!

To keep the workspace display manageable and specific to a developer’s tasks, each development configuration is automatically generated with a filter, so that the developer sees only the dev or cons phases in a particular track (e.g., the TAX10 track). If you would rather not use this filter — e.g., if you are an administrator who needs to perform transports across workspaces in different tracks and you want to display the workspaces of all tracks — you can simply create an access to the DTR called “Client” without a filter, which will display a view similar to the one shown in **Figure 13**.

Buildspaces

Buildspaces are the final stop in our mini-tour of NWDI administrative concepts. A buildspace is an area in the database that the CBS uses to manage the archives in a development configuration. A buildspace is created automatically by NWDI during track creation and then filled with the archives of required SCs (and the contained DCs) by the administrator. During activation, the archives of SCs under development are added. It contains all archives of a development configuration — those of existing software components and newly built ones. When a developer attempts to activate changes, the CBS loads all of the components needed for the build into a local temporary folder — specifically, the object you’ve modified, plus the active versions of any existing software components upon which your component depends — and executes the build.[13] The build result — the “archive” — if successful, is written to the buildspace in the database, which then contains the archive state of the development configuration’s software components. At this point, the object is considered “activated” and made available in the active DTR workspace (see the sidebar on the next page for more on activation).

The ability to execute builds on the development component level before activating them, rather than having to build the entire application, is another advantage of NWDI. It isolates the central buildspace content from any unsuccessful builds, so that a bug in a single development component does not break the build of the complete product, forcing everyone to wait for a fix. It also ensures that only a successfully built component is activated as part of the product. With the local development component build, the developer can adjust the local environment, test any necessary bug fixes, and then try to activate the component build again. If the developer updated the local environment, the central build will work just as the local build did; if not, the developer simply updates the local environment, adjusts the development work, and tries to activate again, without hindering any colleagues in the process.

Keep in mind that there is one fundamental

[13] Keep in mind that modified versions of these used components may have been activated by other developers since you began your work, so pulling down the latest active versions of these components is essential to keep the platform consistent and in healthy working order. If the activation fails for any reason, the CBS will not activate the objects, and will advise the developer of the problem. If activation succeeds, the CBS will update the active workspace with the objects and rebuild the dependent DCs automatically, establishing a new baseline version.

The three rules of activation

As indicated earlier, one of the great features of NWDI is that developers can work with multiple versions of each development object — be it in development or consolidation, in the creation of version 2.0 of a product, or in the maintenance of version 1.0 — in parallel without incident. One of the key ways that NWDI manages to do this is through the concept of *activation* — the successful build of an object in the CBS — which operationally enforces three key rules:

1. In each (logical) system (development, consolidation, test, and production), there can only be one active version of each object at a time.
2. If you want to use objects created by other developers in your development work, you use the activated versions of these objects. (It is possible to use a checked in but not yet activated source version, but you should only do this in consultation with the original developer.)
3. For an object to be marked "active," it must build without errors when compiled against all other active versions in the system. After a successful activation, dependent objects are rebuilt automatically.

A check-in/check-out mechanism allows developers to make concurrent changes to development objects, but notifies them about the checked-out state of an object. The DTR detects any problems when the first conflicting version is checked in and requires a resolution of the identified issues before allowing the second conflicting version to be checked in. Thus, the state of the workspace is kept in a consistent state. It is possible to configure an "exclusive check-out" to prevent this, but using this option can be problematic — e.g., a developer might exclusively check out an object and then become unavailable (due to illness, for example) before checking it back in, preventing others from making necessary changes and holding up the process.

difference between the CBS buildspace and the DTR workspace: In the DTR workspace, old file versions are added to the history and the new file version becomes the active version — the "active" designation moves from the previous version to the new version. In the buildspace, the new version overwrites the older version. Technically, the DTR workspace only contains pointers to the older versions; the versions themselves are all stored in the DTR's database tables. Therefore, a workspace defines the source state of a software component version, and because it does not require much space, DTR workspaces are very seldom deleted. The CBS buildspace, on the other hand, contains the actual state of the software component's runtime objects. Because archives use a lot more space than sources, newly created archives automatically overwrite previously created ones.

Figure 14 shows the CBS Buildspace Information screen, where you can check the archived buildspaces to see if any problems occurred during the build. As you can see in the example, the buildspace of the development phase (D) of the example Tax product is in a "green" state — that is, there are no "dirty" development components (those that still need to be

Note!

If you need to restore any deleted buildspaces — for example, if you want to re-create a specific state of a product for maintenance reasons — you can easily re-create them from the stored workspaces. Simply export the state of the workspace, create a track, import the exported workspace state plus the required software components, and execute the build.

 ©2007 SAP Professional Journal. All rights reserved.

Figure 14 CBS buildspaces overview

rebuilt) and there are no “broken” development components (those that are not buildable, due to inconsistent changes in a referenced development component, for example).

This rounds out our discussion of **Figure 9**, in which you can see that objects begin their lives as part of the inactive workspace, they are then stored in the buildspace and added to the active workspace, making them available system-wide to all developers. All developers’ objects must now compile against the active version of these objects in order to be activated themselves. Note that all of these steps and states can be easily “tracked” by each developer and monitored by the administrator.

Now that you have an understanding of the fundamental concepts underlying NWDI, let’s take a look at what this means for the Java development process. In the next section, we’ll walk through an overview of the enhanced process to provide you with a solid foundation for the detailed example in the second installment of this article series, and to prepare you for your own NWDI development endeavors.

Enhanced Java development process with NWDI

Earlier I stated that NWDI doesn’t drastically change the methodology that Java development teams use to build software. The process you’ll follow with NWDI (illustrated in **Figure 15** on the next page) has all the familiar earmarks of your current development process: defining what needs to be developed, checking out any required libraries or sources, doing your development, testing locally, checking in the changes, testing centrally, and transporting the final archives to quality assurance and production. The particular steps and

No portion of this publication may be reproduced without written consent.

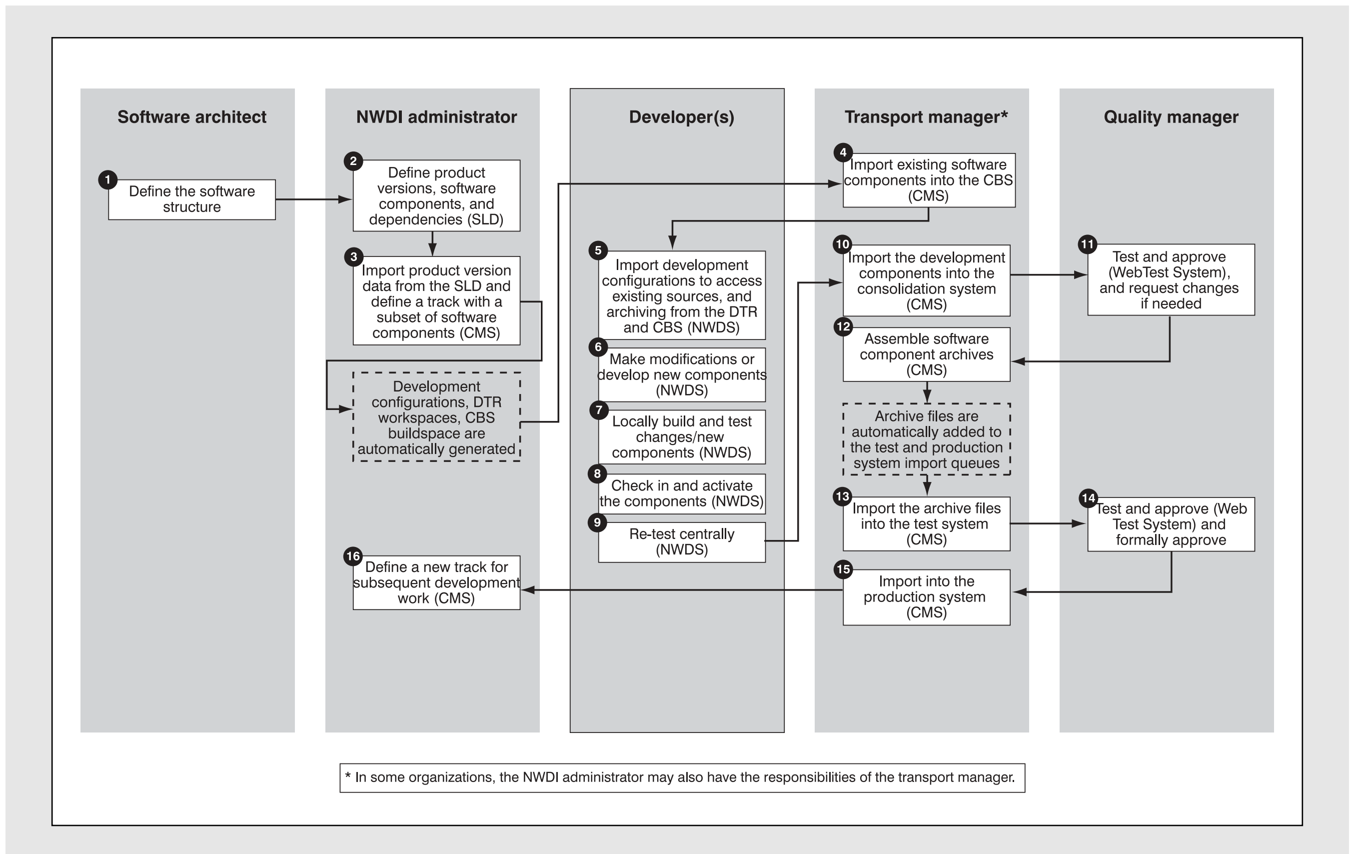

Figure 15 Complete NWDI-enabled Java development process

Note!

Notice that the process steps in **Figure 15** are neatly divided across five discrete roles: software architect, NWDI administrator, developers, transport manager, and quality manager. This demonstrates a key advantage of NWDI: clear segregation of duties with defined hand-off points between roles, each supported by an easy-to-use toolset. Developer tasks are all done within NWDS. All other activities are done within the centralized CMS Web administration tool and the CBS Web user interface. For more information on what each role does, see the sidebar on the next page.

procedures you'll use, however, will seem a bit foreign at first, so in this last section I familiarize you with them a bit by briefly stepping through the process.[14]

[14] *Process* is the keyword for NWDI — every step is well defined, every software state can be reproduced, and the development enviroment is consistent for the whole team.

Starting in the upper-left portion of **Figure 15**, all development and enhancement projects follow these steps:

1. The software architect defines the structure of a product, including new software components to be developed, the target platform release, and any

 ©2007 SAP Professional Journal. All rights reserved.

NWDI team member roles and responsibilities

One of the great benefits of NWDI is that it boosts Java development team productivity by enabling clear segregation of duties. Each team member has a specific role in a defined process, as well as a dedicated toolset with which to execute each required action. As you can see in **Figure 15**, there are five key roles into which Java team members can be divided from an NWDI perspective:

- Software architect
- Developer(s)
- Quality manager
- NWDI administrator
- Transport manager*

The role of the software architect is to define the software structure shown in **Figure 8**. The NWDI administrator's job is to create the new product version and the new software components in the SLD and to define the track in the CMS Landscape Configurator according to the software architect's specifications (including the "initial load" of required software components or previous versions in workspaces and buildspaces). Using NWDS, the developer imports a development configuration and changes objects or appends objects to software components included in that configuration. After testing in the development runtime system, the developer checks in, activates, and releases the changes, at which time the transport manager uses the CMS Transport Studio to import objects into the consolidation system, where the quality manager organizes the tests. Finally, the transport manager assembles the changes in software component archives, which, after final testing, can be delivered or deployed into the production

* Depending on the size and needs of your organization, the NWDI administrator may serve the role of the transport manager as well.

Note!

Keep in mind that from a permissions perspective, there are only two roles in NWDI — NWDI developer and NWDI administrator — so technically, it's very simple to set up team permissions.

dependencies to existing software components that developers will (or might) need to use as a part of the development process.

2. In consultation with the software architect, the NWDI administrator creates the product version (e.g., "Tax product, version 1.0"), the new software components to be developed (that is, placeholders to which developers will be able to attach development components), and defines any existing software components to be used and any dependencies within the SLD (at build time and installation time).

Note!

Developers will only have access to development components within software components that are assigned to the product at this stage, so be sure to include any components that you suspect they might need to avoid delaying development. It is difficult to remove objects once they have been used, so if you are unsure, it's best to leave the component out until you can confirm it's needed, and then simply add it later and update the track.

No portion of this publication may be reproduced without written consent.

3. The NWDI administrator updates the CMS by importing the definitions created in the SLD. The NWDI administrator then uses the CMS Landscape Configurator to define the needed track, which describes the phases, systems, and transport path that particular software components will take on their way to deployment, thus defining the development landscape. As part of defining a track, the NWDI administrator then uses the CMS Transport Studio to check in the software components that were defined for the product in Step 1 to implement them in the development landscape for this software project. The NWDI administrator notifies the transport manager when this configuration is complete.

> ***Note!***
>
> As indicated by the dashed box between Step 3 and Step 4 in the diagram in **Figure 15**, the system automatically generates the development configurations, DTR workspaces, and CBS buildspaces needed by developers when the administrator saves the track.

4. Using the CMS Transport Studio, the transport manager imports the needed objects for all the software components included in the track (or tracks): sources and archives for the software components under development if available and archives only for the previously existing software components that will be used for developing the new ones. These objects are imported from the server's file system, where the available archives are stored in the NWDI's workspaces and buildspaces. This is the final required step needed before developers can begin their work, so at this point the transport manager notifies the developers that the system is ready for them.

> ***Note!***
>
> If Step 4 is skipped, developers will not see the necessary sources and archives, nor be able to download them.

In Steps 5 through 9, developers synchronize the needed files to their local file system; create new source file versions locally; build and test the components; check in, activate, and test the results, which are automatically deployed on a central development runtime system, if available; and, finally, release these objects if the central test is successful. Let's continue with our walkthrough by taking a closer look at these key development steps (remember that all of these development steps are performed using menu commands in NWDS):

5. At this stage, the developer imports the development configuration, which, as you'll recall, describes the structure of the product, the new and existing software components the developer can change or leverage, and the URLs of the various servers NWDS must talk to (servers for the CBS, CMS, etc.). The developer downloads source files and Java archives (from the consistent central storage locations) for development components of referenced software components to retrieve the latest active versions of sources and archives from the server. Source files are downloaded from the DTR, and Java archives are downloaded from the CBS buildspace, where all the archives reside.

> ***Note!***
>
> You will always automatically get the correct archive version related to the chosen sources, because the buildspace and the active workspace are always in sync. This also holds true if you sync a development component to change it from the inactive workspace to the active workspace — required archives will stem from the one buildspace of the development configuration.

 www.SAPpro.com ©2007 SAP Professional Journal. All rights reserved.

6. The developer creates new objects or changes existing objects, all of which are stored locally on the developer's PC, and puts them in a change list called a DTR *activity* — the unit of transport in the development and consolidation systems. An activity is essentially a transportable change request that groups related files (for example, all source files of a new development component project or two files together fixing a bug).[15] NWDI automatically calls the name service when new development components are created to ensure that the names are unique across the landscape.

7. The developer then initiates a local build within NWDS using the local objects for the build. The resulting archives are stored locally and the developer is advised of any errors that occur during the build process. The developer deploys the archives to the local SAP NetWeaver AS Java engine provided with NWDS, and debugs the code locally.

8. The developer checks in the source objects to the central NWDI server, which causes NWDI to deposit the sources into the DTR. The developer activates the objects, at which time the server's CBS executes a fresh build of the changed development components against the latest active versions in its buildspace[16] of all referenced software and development components. The developer is again notified if any errors occur (which should only happen if the local environment hasn't been updated from the server recently); otherwise, the CBS updates its buildspace with the new version and new versions of dependent DCs afterwards, and triggers the DTR to flag the sources newly built into archives as the new active version. And, to keep the buildspace up-to-date, all development components using the interface of a changed development component are automatically rebuilt in the CBS. The developer has to take manual action only if problems are found (as indicated by **Figure 14**, any such problems can easily be detected).

9. Finally, the developer unit tests the application on the central server hosting the development runtime system, and then releases the activity (containing the new or changed objects) for transport to the consolidation system.

> ***Note!***
>
> The reason for the separation of the development and consolidation system is that, while the development runtime system is changing constantly, the import into the consolidation system is under the administrator's control, so a specific state can be kept and tested by the quality management team.

At this point, the developer's tasks are complete, unless problems are discovered during testing and there are repairs to be made. Let's continue on with the steps:

10. The transport manager imports the archive files into the consolidation system (if one is set up). Once the import is complete, the sources are automatically rebuilt by the CBS and the resulting archives are deployed in the consolidation system.

11. The quality manager tests the application in the consolidation system. Any problems are reported and corrected by the responsible developer (normally in the development system, or in urgent cases, directly in the consolidation system), redeployed, and then retested.

12. Once the testing is complete, the transport manager assembles the deployable software component archive (SCA) files from the development components for the application, and the files

[15] An activity is analogous to a change request in the ABAP world. Technically, an activity is a list of changed files as a transportable unit. In order to check in and activate new or changed code, developers must assign the objects under development to an activity within NWDS. When you check in an activity, all files associated with the activity are uploaded to the DTR. When you activate the activity, the system builds the activity. When the developer releases the activity, an administrator adds the objects to the import queue of the consolidation system for import. We'll look at this in more detail in the second installment of this article series.

[16] There is no way to put local build results into the buildspace directly, as this might lead to an inconsistent buildspace state.

No portion of this publication may be reproduced without written consent.

are automatically added to the test and then the production system import queues.

13. The transport manager then imports the files into the test system.

14. The quality manager tests the application in the test system.

15. Once the application is successfully tested, the transport manager imports the application archives into the production system.[17]

> ***Note!***
>
> Only software components as a whole (as SCA files) are deployed to test and production systems (obviously, deployment into these systems should be handled alike), because these are runtime-only systems (no changes can be made to development components in these systems), in contrast to development and consolidation systems.

16. The NWDI administrator defines a new track for subsequent development of repairs to the product. The result of the first development cycle (the SCAs) is the input for the next development cycle. The software architect could also define an entirely new product version in the SLD if significant enhancements are planned, starting the whole process again.

[17] Archives are not technically transported from test to production, as you might expect. As indicated by the dashed box between Steps 12 and 13 in **Figure 15**, archives are added to both the test and production systems upon SCA assembly. Thus, the transport manager must be careful not to import archives into production that have not yet been imported and adequately tested in the test system.

Conclusion

This article has provided a comprehensive introduction to the core terms, concepts, and steps involved in the NWDI-enabled Java development process. Although some of these terms can be a bit challenging to learn (e.g., tracks and development configurations), they will soon become second nature, especially once I show you the process in action in the next installment of this two-part article series. You should come away with the impression that NWDI delivers a well-organized, sophisticated approach and toolset to address the complex challenges faced by real-world Java development including organizing and managing new development and modification of software delivered by SAP as well. NWDI clearly divides tasks in the software development lifecycle into distinct roles, defines clear hand-offs between each, and supports each with easy-to-use tools. It also always provides a traceable development process with a unique approach to set up consistent development landscapes. The results of the development process can be safely propagated to any system, and it even allows for a modification concept for Java. The outcome is a practical, fully integrated, enterprise-scale solution for development, maintenance, and modification of Java software. With NWDI, you can finally enjoy the freedom of Java development with the ability to manage it.

> ***Note!***
>
> You'll find additional useful information in the NWDI Knowledge Center at the SDN Web site (http://sdn.sap.com → Application Server → Java → NetWeaver Development Infrastructure Knowledge Center), which contains in-depth articles, a forum on NWDI, a link to the official documentation, demo videos on the NWDI-based development process, and even a link to download a sneak preview version of NWDI.

 ©2007 SAP Professional Journal. All rights reserved.