Features of the SAP NetWeaver Development Infrastructure

Athur Raghuvir Yuvaraj
SAP Labs India Private Limited
138 EPIP, Whitefield
Bangalore, India 560 066
yuvaraj.athur.raghuvir@sap.com

Disclaimer
Some components of this product are based on Java™. Any code change in these components may cause unpredictable and severe malfunctions and is therefore expressively prohibited, as is any decompilation of these components.

Any Java™ Source Code delivered with this product is only to be used by SAP's Support Services and may not be modified or altered in any way.

Table of Contents

Abstract

This paper explores the features offered by the SAP NetWeaver Development Infrastructure that support the next generation of SAP business solutions. To handle the current technologies and trends in software development, the best practices discovered in SAP R/3 systems are applied, along with the latest thinking in the SCM area, to create a unique offering to SAP customers—one that preserves existing development process knowledge while it migrates the customer base to a new technology stack.

1. Software Configuration Management and Lifecycle Management

Development of software for large enterprises is a growing challenge. On one side the software is complex due to variety of real life situations that need to be handled and on the other hand, software development is manpower intensive. Typical development of enterprise strength software involves hundreds of developers working together. More complex software development would mean groups of developers working together to develop smaller pieces of software that participate in a "stream" of development where the development groups "downstream" use the software created by groups "upstream." Further more, the downstream groups can be dependent on many upstream developments. Quickly this leads to a huge dependency among the software parts being developed that finally explodes during integration testing and creation of software for final deployment.

The way out of this complex problem of creating software has been well explored in the academia and the industry and goes by the name of *Software Configuration Management (SCM)*. The principles of Software Configuration Management revolve around four key components [1]:

1. *Version Control*: This is simply an automated act of tracking changes of a particular file over time.
2. *Build Management*: *Build* usually refers to a version of a software program under development. Good Build Management has a huge impact on the productivity of software developers.
3. *Release Management*: The primary purpose of a release is to make the software available to the end user. Release management is about packing the production build of the software, the deployment process and the update of metadata that goes into tracking a given version of a software application.
4. *Process Control*: This is like an umbrella over the above three concepts and is a combination of business processes defined for a certain team and the implementation of these concepts within the configuration management tools.

The impact of research in the academia on SCM in the industry has been studied in the paper by Jacky Estublier et al, titled *Impact of Software Engineering Research on the Practice of Software Configuration Management* [2]. To discuss the issues of SCM, this paper refers to many sections of [2] paper as it summarizes the findings of experts in the advances in SCM at the industry and academia.

1.1. Lifecycle Management as More than Classical SCM

Current Business Software is more sophisticated due to the complexity of the domain it handles. However, there is a corresponding simplification that is demanded by the customers of the Business Software since the customers are not software developers. Since the domain of interest of the customers is business, it becomes imperative that the Business Software Vendors discover ways and means of presenting the customer with a solution that satisfies the twin needs of being able to handle complex modifications in a way that is user friendly and reduces the cost of ownership for the customer. So, although SCM

promises to solve many of the problems faced in software development, there are several viewpoints that need to be revisited, particularly in the context of Business Software Development.
In [2], the authors concisely summarize this issue as the departure from the SCM's fundamental assumptions. Traditional SCM is based on two fundamental assumptions:

SCM_1: Focus on managing the implementation of software
SCM_2: Basic philosophy of SCM is programming language and application independent.

The current research, both in the academia and the industry, is now sorting out how to better fit the overall picture of software development rather than always assuming to be able to provide stand alone application that somehow seamlessly fits with existing tools, approaches and practices. This research is breaking the above fundamental assumptions of SCM in the following sense – (a) the first assumption is broken when management of artifacts produced earlier in the lifecycle (e.g., requirements and design) and later in the lifecycle (e.g., deployment, dynamic deployment and reconfiguration) is required, and (b) the second assumption is broken when SCM is integrated into dedicated environments (for e.g., integrated development environments) and representations (for e.g., product line architectures).

Some of the issues that remain critical for providing the differentiating features of a development environment are:

1. Refinement of the System Model to handle Components
2. Simplification of selection for user workspaces
3. Support for Central Test Environments
4. Support for Layered Development
5. Support for Customer Delivery
6. Process Migration

1.1.1. Refinement of System Model to Handle Components

Business Software has a requirement of handling Component based development. This is required to allow for configuring the software as per customer requirements. There is a strong wish to allow for reusability across business software components. This requires stronger structuring than the generic SCM systems. Formal basis for Component and their interdependencies can help in defining design time, build time and deployment time automation. Clearly, there should be a defined way by which external software can be used in the component world. However, to control arbitrary use of components from external sources at a local level, the formal model needs to be closed and centrally administered.

1.1.2. Simplification of Selection for User Workspaces

As complex system models are introduced into the development processes, it becomes necessary to define selection of resources into the user workspace. Generic mechanisms for these often lead to a spectrum of problems ranging from every developer's workspace being different from everyone else's to fear of use of the features of selection mechanisms.

The question that is relevant is that in a defined group of developers working as a team on a software component, how different should the workspaces be. And the answer is that the workspaces of the different users have to be the same! Surprising though this result is, this simplification immediately lends itself to more control of the user's workspace and also allows the central build systems to make consistent assumptions on the contents of the workspace. The immediate next point would be who creates these centralized selection definitions, how can it be maintained and effectively used. If the system landscape definition can have one central system that is highly available and is designated as the unique one in the entire organizational system landscape, the answer is easy. This unique system would be a central store of the different selection criteria and the development environments would be enabled to connect to this central system to list for the user the various selection definitions allowing the user to choose and thereby customize the development environment to work with a specific selection definition.

Thus the configuration for the development can be centrally defined and users choose the most relevant configuration to work with. This vastly simplifies the user workspace content maintenance.

1.1.3. Support for Central Test System

Most server application development requires server installations to work with. And on typical developer machines, the machine configurations are not designed to host high end server applications. The way out would be to define Test Systems as part of the development landscape. Then, one of the process steps before a change is submitted would be to check the application after splaying it onto the Central Test System. The advantage of a Central Test System is that the software state can be precisely controlled reducing the issues of local tests that work while integration tests that fail. This works on the simple principle that earlier an issue is detected in the development cycle lower the cost for fixing it.

Now that the requirement of Central Test Systems is clear the software state in these Central Test Systems need to be controlled through policies and automation. Thus in the development process it pays to think about how a Central Test System can be used for testing and how the Central Test Systems are updated with the latest development state.

1.1.4. Layered Development Process

Another important process in the construction of software that is layered as technology, application and specialization is to support development that happens in layers as well. This layering is common in Business Application development. For example, the technology deliverables are more related to frameworks and hosting environments. On these environments applications that address domains like material management, human capital etc are developed. Later, these domain applications are tuned or specialized according to industries. This stacked application architecture demands a support during development that also includes the necessary validation steps.

By organizing the development into layers that mirror the different parts of the application, and setting up transport of software between the layers, it would be possible to define a flow between different development teams. At the handover points, there would be qualifications as relevant for that software before it is applied onto the next layer. This stabilizes the environments for development during development. Eventually, if there are clear policies and automated mechanisms, the layered development optimizes the development time by making changes available to the higher layers as soon as possible resulting in better integration of the software.

1.1.5. Custom Delivery

Many of the commercially available SCM systems are designed to solve the issues with-in an organization. The organization itself could be multi-site and the development could also be distributed. However, the issues of cross organization SCM level collaboration needs to be solved for Business Applications. This is acute in case the Business Application is constructed in such a way that the semantics of the application is captured through metadata that can be modified at the customer’s site. The moment modification of the same software extends beyond organizational boundaries, the issues of parallel-edit detections across the SCM systems becomes critical for the customer.

Business Application Vendors typically have a long term software flow relationship with the software consumer. There are upgrades and patches to the software metadata that need to applied on the customer systems. Further, the Vendor cannot insist on any particular SCM system for the customer since the customer might have already invested in different SCM systems.

To control the cost of ownership and not losing customers due to pre-defined IT landscape of SCM, organizations tend to have their own SCM technology that can be offered along with the Business Application.

From the SCM system architecture and design, what is important is that SCM needs to be across organizations and the semantics should allow for global version histories that are independent of transport order.

1.1.6. Process Migration

As mentioned earlier, Business Application Vendors have longer relationships with their customers since the applications are typically robust for longer times and the need to move to latest Application version is not critical for the business. This is clearly indicated in the number of shops that are still using mainframe systems running legacy databases and Business Applications.

However, the solutions and technologies keep evolving and the Business Application Vendor is faced with the challenge of how to migrate the existing customers into a new technology base. Although this looks like a seemingly minor point in the SCM area, this could be a major deciding factor since change of technology could result in re-training the entire IT unit so much so that the migration cost could work against the options of moving to latest technologies.

To minimize the impact of migration to a new technology, a similar process in the new SCM systems that support the Application Modification could persuade the organizations to migrate to the newer applications better. This way the organization is able to appreciate process continuity even if there is a technology discontinuity.

2. SAP and SCM Technique

SAP's flagship ERP solution is based on a proprietary technology that consists of an R/3 Server and ABAP as the programming language for the applications. Very early it was discovered that if ABAP programs are to be delivered to the customers, there is a need for versioning mechanism. R/3 supports an integrated design and runtime environment and a versioned repository.

The SAP R/3 system is described concisely in the following sections to indicate the discovery of some points that were found relevant in Business Application Development and Delivery. The theme that emerges has implications on any new Development and Delivery strategy that SAP would like to innovate beyond the SAP R/3.

2.1. SAP R/3

The *SAP R/3 Change and Transport Management* [3] details the architecture of the R/3 Database as almost everything the users can see: transaction data, program source code, text, menu options, screens, and even printer and fax definitions. The R/3 Database can be logically divided into *R/3 Repository* and *Customer Data*.

R/3 Repository provides the data structure and programs that are needed to maintain data in the R/3 System. The central part of the R/3 Repository is the *ABAP Dictionary*, which contains the descriptions of the structure and relationships of all data. These descriptions are used by R/3 to interpret and generate the application objects of the runtime environment – for e.g., programs or screens. Such objects are referred to as *Repository Objects*.

During R/3 implementation, the *ABAP Workbench* is used for creation or modification of the R/3 Repository Objects.

Customer Data defines any kind of data entered by the Customer including customizing data, application data and user master data. *Customizing data* is generated when R/3 is configured to meet the particular needs of the customer through customizing. *Application data*, also known as *business data*, is the data required for or generated by day-to-day business processing in R/3. *User master data* is the records of R/3 users' password and authentication.

R/3 allows for setting up subdivisions called *clients* that help separate customer data into different groups. Among the data in the R/3 database, the application data is client specific. Some customizing data could be shared by different clients and thus is client independent. Repository objects are client-independent.

The software SAP delivers to its customers is referred to as *SAP Standard*. It contains over 1000 business process chains and their associated functions. Before working with R/3, there is a step called *implementation*. Implementation is about capturing the specific requirements of the company through choice of process and variations in them. The implementation of these decisions is supported through *customizing* and *development*. The customizing procedure basically adds customer specific data to tables corresponding to SAP-standard objects.

To satisfy business needs beyond the scope of SAP standard, customer development is required. Customer development is possible through creation of new Repository Objects, enhancements or modifications.

Addition of new Repository Objects have to follow recommendation of SAP to ensure that further changes to SAP standard do not disturb the customer development. These recommendations are a) customer *development class* that groups the repository objects that are created by the customer, b) *customer name range* to define the names of the repository objects and c) customer *namespace* when the development is larger and decentralized.

Enhancement is through defined mechanisms of *exits*. Exits are places where SAP standard objects allow for changes. Exits are defined on SAP Programs, menus, screens, tables, fields and text. The merit and purpose of SAP enhancements are to enable addition of functionality to SAP-standard objects by creating new objects than modifying the SAP-standard objects. By following enhancement, the customers benefit through a) receiving SAP customer support is easily, b) fewer problems on application of SAP's periodic correction to SAP software and c) release upgrades can be performed more quickly. SAP further guarantees that through enhancement techniques, there would be no loss of functionality provided through enhancement on the application of an upgrade or supply package.

Modifications are also possible to the SAP-standard objects. However, modifications are not guaranteed to work after upgrades or application of support package. To make customer life a little better, a *Modification Assistant* guides modification. Further when the modifications have to be adjusted due to an upgrade or support package application, the modification assistant is of help.

2.2. System Landscape and Change Process

A customer has to customize the R/3 for adapting the business process for an organization's needs. For this, SAP proposes using few special clients – *customizing and development* client (CUST), *quality assurance* client (QTST) and *production client* (PROD). The recommendation here is to do Client-dependent changes in the different clients and use the *promotion* mechanism to move the changes across the different clients. In this way, the changes are controlled and carefully applied to the PROD client.

However, real adaptation to business would touch objects that are client independent as well. In order to handle these changes, SAP recommends distribution of the CUST, QTST and PROD clients among several R/3 systems. The critical clients distributed amongst R/3 systems with *transport paths* defined constitute a *system landscape*. The objective of a system landscape is to provide an implementation environment where a) Customizing and development can be done without affecting the production environment, b) business processes can be validated before using them in a productive environment, c) R/3 Release upgrades and support packages can be simulated and tested before they can impact a production system and d) work on customizing and development for future business requirements can be done without influencing the current production environment.

System Landscape can be a simple 1-system or an n-system global landscape. The landscape is designed with the organizational needs and requirements. The interesting point to note here is that the landscape for progressive application and testing of change before it reaches the productive system is a best practice that has been successful for many R/3 implementations. For in-house or extensive development, there would be need to change the client independent objects as well. Thus, the recommendation is to have a 3–system landscape of CUST-QTST-PROD.

Although it is possible to manually reenter the changes in successive systems in a system landscape, this is not a desirable option due to the quantity and complexity of the changes required. For this the R/3 enables recording of the changes in a *change request* that can then be distributed to other clients in the same R/3 system or in another R/3 system. Change requests can contain client dependent changes – in *customizing change request* – or client independent changes – in *workbench change request*.

Basically, change requests are simply a collection of tasks that list the different objects in the R/3 database that have changed. This information can be used to update any other client in the system landscape with a new copy of the changed objects. For this the possible technical procedures of transport are *promotion* and *import*. Promoting change involves releasing changes and then exporting them out of their R/3 system and onto the operating system level; in a further step, they are imported into another R/3 system. Before promotion and import can occur, *transport routes* between R/3 systems needs to be defined in a strict way.

2.3. Development Process

The development process starts in the CUST client in the landscape. The settings of such a client typically would allow for automatic recording of changes done on the objects. Collection of the changes would result in a change request.

To release a change request, each separate task must be documented and released. It is also recommended that *unit testing* is done before the release of objects recorded in a change request. To achieve this, testing can be done in the development client or in a separate client that can be created through a process called as *client copy*. After suitable quality is ensured, the change request can be released.
The SAP R/3 system allows for activating object checks at the point of change request release. These checks identify and display errors such as program syntax errors. If errors are detected, the user has the possibility of continuing the release process, viewing the errors or canceling the release process.

Successful release of a change request initiates the *export process* and the corresponding recipient system gets an entry in its *import queue*.

The import process uses the information in the import queue to start the import. On successful completion of the import, the change request is entered into the downstream client of the system that has received the import. This is necessary since the transport order is important for the change propagation. Any missed transport can result in system inconsistency. The transport mechanisms of R/3 are built to prevent any missed transport in the system landscape.

During import there are two options – *import all* or *preliminary import*. Import all would import all the waiting change requests. However, it is possible to set an *end mark* to define the scope of the import. Preliminary import is to be used in exceptional situations where a change later in the import queue needs to be imported first. This is only for exceptional situations and is not recommended for regular development procedures.

Post-import issues include reviewing relevant logs, resolving any errors that occurred during import and notifying people who will perform testing and business validation.
Summarizing, during development a developer performs the following steps to make changes in the R/3 system [*R/3 Dev Process*].

1. Logon to SAP Development system that defines the whole environment of own and foreign libraries.
2. Edit and Lock development objects. Since the development system would be configured for automatic recording of changes, this would result in a change request.
3. Document the tasks and perform unit tests.
4. Release the Change request that could optionally trigger the object checks.
5. Successful release triggers the export process.

The object checks that run on release is an important aspect of detecting compatibility issues early. In standard source control systems, there is no linkage to the build mechanisms and so syntax validation at the point of submit of a change is not possible. In Business Application development, having such checks early means early integration and hence higher productivity.

2.4. Versioning for Repository Objects

In order to track the history of changes on Repository Objects, a versioning mechanism is associated with the Repository Objects. There are two types of versions identified – *temporary* and *active*. *Temporary version* can be created at any time. These versions are stored in the *version database*. A version is created when a change request is released. *Active version* is the version that is the current active state of the object. Active versions define the current state of the repository objects. *Development Database* contains the active versions of an object.

The version database is closely associated with the development system. If a development system is discontinued, all the version history for the custom developments and modifications to SAP objects made in that repository would be lost.

By default, R/3 Repository Objects are not versioned upon import. This restricts version histories to the development system. If there is a need to maintain the versions in all systems in the landscape, that can be achieved too.

R/3 supports linear versioning of Repository Objects. And this version history can be transported by enabling versioning on import. However, parallel edits are not supported allowing only one user to modify at a time. Preservation of repository states is done by copying clients or systems or both.

2.5. Extended Development Landscapes

For better control over the development in-house, at partner sites and at customer sites, R/3 lends itself to extended landscapes. Although the end deliverable to a customer is referred to as SAP Standard, internally, the software is organized as *Core, Industry Specific* (verticals) and *Localization*. This is required to satisfy diverse business houses that require ERP solution. For example, the Core consists of Human Capital Management (HCM), Finance and Controlling (FiCo), etc. modules. Over the Core modules are the modules that are customized based on industry verticals. For example, the Industry Specific Modules could be for the hi-tech industry, or aerospace and defense, or apparel and footwear. The verticals are defined based on shared business practices and require domain knowledge of the business house. Finally, before delivery to customers, the location of the customer would decide what human language would be used to interact with the application and location specific details like taxation laws and legal issues of accounting. This last part of tuning the application is referred to as Localization. Thus, even before the SAP software reaches the customer location, there are various *layers* superimposed on one another that together satisfy the requirement of the customer.

The layered software development requires landscape support that allows separating the development phases of the software of the different layers. Similar constructs can also be found in partner development where layers can be added onto the SAP delivered software and finally at the customer site that has a central development followed by regional specializations.

With SAP R/3 the layered pattern of software development can be quickly achieved by repeating the CUST-QTST-PROD pattern for each layer. Thus, extending the landscape to n-system landscapes allows for development within a layer or organizing layers of development.

2.6. Best Practices

The SAP R/3 and the development process that has evolved contain some best practices that add value to Business Application development. The following is a list of practices that have been discussed so far:

1. The R/3 has an integrated data base of the runtime system and the design time objects. The design time objects are interpreted at runtime for providing a customized implementation of a business process.
2. The Integration Guide and ABAP workbench together encompass all the possible changes that can be done on the system.
3. There are clear recommendations on what kind of implementation practices that a customer should undertake so that SAP upgrades and Support Packages disturb the business process minimally.
4. There is a clear software production process that is based on a system landscape
5. The grouping of changes and transport of changes are well defined.
6. There is a versioning system built into the change process to track the changes. The propagation of the version information across the landscape is configurable.
7. The software development is layered to allow for specialization of software that can be handled by different organizational groups.

Abstracting out these best practices for application to other system architectures and technologies, we get the following:

1. Out-of-the-box solutions have to have a following customization and modification to achieve a better fit to customer needs. For this, the application architecture should be such that there is a *framework* that is not modifiable and *metadata* that can be tuned through various techniques including configuration at the minimum and modification at the maximum range of variability.
2. Production of software can be better handled through staging of software. For this a multi-system landscape is a best practice that helps in real life development situation. System Landscape requires capacity to handle system landscape patterns, strict definition tools and fault-tolerant transport mechanisms. System Landscape must also scale for Layered development so that many organizational groups can collaborate in the development and delivery of software.
3. There is a constant need for software to flow between the supplier and consumer. Thus the landscape that supports the software lifecycle extends beyond organizational boundaries. This calls for mechanisms and process on how the changes can be handled on import such that the customizations and modifications on the customer site are not lost and hence, the productivity at the consumer end is not compromised. This results in the tension between the levels of change that the software permits versus how the changes can be handled efficiently.

With the proprietary ABAP language, the SAP R/3 system was an innovation in its time incorporating mechanisms of versioning, change management and system landscapes. Moving over to the open era of software development, the best practices of R/3 system get a re-look for a graceful migration of the process discovered over decades. Keeping these important aspects in mind, SAP has created the SAP NetWeaver Development Infrastructure.

3. SAP NetWeaver Development Infrastructure

The maturity of an SCM system can be illustrated well with Susan Dart's model. This model was introduced way back in 1991 to capture the requirements of a SCM system. The important parts of this model are:

- Components. SCM systems must support their users in identifying, storing and accessing the parts that make up a software system. This involves managing multiple versions (both revisions and variants), establishing baselines and configurations, and generally keeping track of all the software systems and overall project.
- Structure. SCM systems must support their users in representing and using the structure of a software system, identifying how all parts relate to each other in terms of, for instance, interfaces.
- Construction. SCM systems must support their users in building an executable program out of its versioned source files, and doing so in an efficient manner. Moreover, it must be possible to regenerate old versions of the software systems.
- Auditing. SCM systems must support their users in returning to earlier points in time and determining which change have been performed, who performed those changes, and why the changes were performed. The SCM system should serve as a searchable archive of everything that happened.
- Accounting. SCM systems must support their users in gathering statistics about the software system being developed and the process being followed in so doing.
- Controlling. SCM systems must support their users in understanding the impact of change, in allowing them to specify to which products a change should apply and providing them with defect

tracking and change request tools such that traceability exists from functional requirements to code changes.
- Process. SCM systems must support their users in selecting tasks to be done and performing such tasks within the context of the overall lifecycle process of software development.

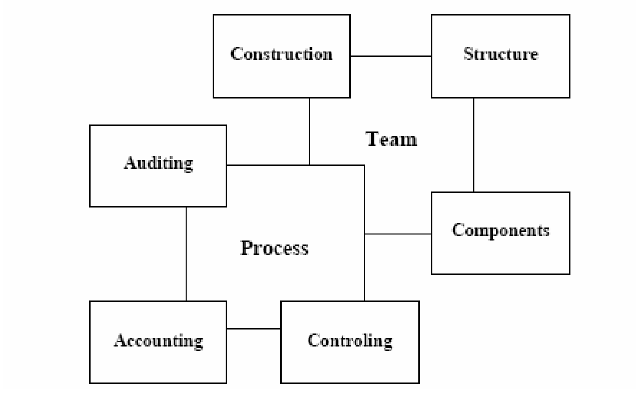

Figure 1: Functionalities of SCM Systems ([4])

SAP as a business software provider has the need to create the next generation SCM support for Business Application development. So, all the criteria as defined by Dart do not directly apply. As noted earlier [2], the industry direction is more towards encompassing other artifacts during the software lifecycle and building dedicated environments that are tuned to application and even language in which the application is developed.

However, the Dart model enables evaluation of the level to which the SAP NetWeaver Development Infrastructure (NWDI) meets these criteria.

Given that one of SAP requirements for a new generation SCM system for Business Application development must also have a process migration for the existing R/3 customers, it becomes clear that instead of exploring the various options, the choices are fairly limited to the change process that the user is exposed to in the R/3 world. Due to technology discontinuity and incorporation of other advances in the various areas of SCM, the challenge is to keep the change management process for a developer in the new system as close as possible to the R/3 Change Management Process.

In the following sections the various components of the SAP NetWeaver Development Infrastructure is explored using the Dart model as a framework for SCM systems while keeping in mind the process migration impact for SAP R/3 users.

3.1. Version Management

For auditing requirements of the Dart Model, the requirement of a versioning system is clear. There are many flavors of versioning system available in the market today from the generic versioning system providers. The versioning systems can be categorized based on the interface/access mechanisms, the version graph for single versioned resources, the scope of user isolation for change and support for multi-site development. The versioning system developed by SAP as part of the SAP NetWeaver Development Infrastructure is called *Design Time Repository* (DTR).

As far as Business Application development is concerned, there is a higher emphasis on early integration of changes. This results in a design that allows for what is also referred to as *rolling baseline*. A baseline is the current active state of the software being developed. When a user makes a change and submits the change, the change is published to the baseline and thus all the users can see the latest state. There is no choice for the user to work on local changes for a long time. This forces all the users to validate their changes immediately and work towards a consistent software much early in the software development lifecycle.

From an interfacing and terminology perspective, the DTR is based on the *WebDAV/DeltaV* protocol. This protocol defined *workspace* management through *activities*. Change management in the WebDAV/DeltaV protocol is about *creating* an activity, *check-out* of resources into this activity (this implicitly associates an activity to a workspace), *editing* the resources and finally *check-in* of the activity. The developer is expected to validate the compatibility of the changes (like build success) and functional correctness of the changes in the activity.

What the WebDAV/DeltaV protocol does not address is the multi-site development. DTR has to be fundamentally multi-site since the application metadata needs to be delivered to customers where customers can also make changes. This important extension was necessary to make WebDAV/DeltaV usable in the SAP world.

One interesting requirement is that source control systems must serve as a repository for a variety of applications. The global standard works on a file-folder mechanism for storage and many editor (or designer) applications are based on file-folder structures. However, many of the current applications are object based or object oriented. This poses an interesting question of how to store object-based applications in a flat format. The answer lies in embedding the object structure into the flat file! Any structured format can now be captured elegantly through an XML schema. The transformation to and from an object net to an XML schema has good performance in the current generation of processors and tools. So, the file-folder structure for storage allows for generality in terms of different operating systems while the format of data in the file in XML does not rob away the object advantage.

3.2. Components and Component Structure

Structuring software is a formal necessity if higher level goals of reuse and recombination of software have to be achieved. And it is practically impossible to impose a formal structuring after allowing for unstructured software development over some time. At SAP as well, in two different versioning systems – the R/3 and Mobile Application Repository – the idea of formal structuring was introduced as the second stage in software design. However it became clear that there was no automated migration possible since the semantics of interaction were quite often buried at different levels of software interactions – at tables in a database to method code in software.

The designers of Development Infrastructure took special note of this problem and decided to impose a *component model* as a fundamental structuring of the software. To start with are *development components* (DC) that are the smallest unit of software that can be independently built to create a self-contained binary. Development components depend on other development components through a variety of *dependencies* like design time, compile time, run time and deployment time. One important decision here is that the component model is *closed*. This means that dependencies can be created only among components that are well formed. This immediately leads to the question of handling external binaries on which the software depends crucially on. If uncontrolled dependencies are allowed, inconsistency of the builds of the software on a user machine and a central machine are inevitable. On the other hand there needs to be a mechanism by which external / third-party software can be integrated into the system. The designers of NWDI wore the thinking hat for enterprise application development and made a conscious decision that the component model would be closed and any new external dependency would have to be centrally introduced into the component model. This decision has the advantages that a) components are introduced in an administrative way centrally, and b) the user’s environment is always made consistent with the central environment. Generic SCM providers might claim a loss of independence on the user's side. However, to control a large-scale development project, simplification of selection of resources for development is an important issue and maintenance of consistency is an even bigger issue that out weighs the offer of full freedom to the developers.

To move towards a selection procedure, the development components are contained in a *software component* (SC) that is the complete deployable software. SC are higher level aggregations of DCs and are the level at which an organization plans it product release contents. A certain SC might or might not belong to a certain Product Release cycle. However, development planning starts with defining which SCs can participate in a certain release of a product. The key advantage of this artifact lies in the fact that it ties the organizational policy to development without in anyway disturbing the software structures. So, architects, designers and software developers have the freedom to create DCs within a SC that are eventually bundled into the SC for deployment.

The closure of the component model implies that SC dependencies should also be formally captured. Thus the high level definition of a SC also defines the levels of dependency between the SCs. Putting all these points together, the top-down development process would be like the following:

1. The organization decides on the products that would part of a certain release.
2. The SC that needs to be developed – in *source state* – is identified.
3. The dependent SCs[1] – in *archive state* – are also identified.
4. Dependencies between the SCs are classified into design time, compile time and deployment time dependencies.
5. The result of this process is a *development configuration*.

The development configuration is the selection definition of the resources that can be used for the development of a certain SC.

Before moving over to the construction of software, it is important to see how the Component Model definition can itself evolve. From a high level development configuration perspective, the evolution is at organizational level and happens per product release. And this definition must remain stable through one version development of the product. However, the development components are far more dynamic. To allow for recording of version history of the components, the NWDI designers placed the component model definitions into DTR! This allows the component model the same degrees of freedom a regular versioned resource has.

3.3. Build

Construction requirement of SCM system in the Dart model leads to the discussion of Build support. In NWDI the component model contains the build definitions. This is captured at the level of development configuration itself. And this information has to be used by a central build server to create binaries of the sources created.

Build is a performance intensive process. To ensure consistency with the definitions, the build process typically has to prepare the environment before the build of the sources can be done. This means that the

[1] *Dependent SCs* are to be interpreted as the SCs that are *used* by the SC identified in point 2.

compile time dependencies have to be recovered for that specific definition to set the state for the build of sources. Due to the fact that build tools of standard software work against a file system, a file system has to be prepared before the build can execute.

In order to handle the complexity of the build process the NWDI has *Component and Build Server* (CBS) that ensures the consistency between the component model definitions and the compilation process.

Taking a best practice from the R/3 world, the NWDI introduces an *activation principle* to define consistency of sources and archives. To start with there are two workspaces defined for a software component in source state – an *inactive workspace* and an *active workspace* in DTR. Inactive workspace is the place where developers make changes to the source in activities. During the development process, when the activities are checked-in, the developers request an *activation* of the new activities which is technically a build request to CBS. CBS syncs the sources from the active workspace, and then syncs the sources from the activities that are part of the activation request and triggers a build. Once the build is successful, CBS initiates an integrate operation of the activities into the active workspace. In this sense, the active workspace source state is maintained by CBS. In the active workspace, the sources are integrated only after a successful build. So, at any point in time the last successful build state can be reconstructed from the sources in the active workspace.

Thus, the activation principle achieves a two fold result – 1) the changes of a user are immediately verified to be in a state that allows for error free compilation and 2) the state of the active workspace corresponds to the latest archives that are available in CBS. This important principle is another outcome of the aim to achieve early integration to create stable software along the lifecycle of the development.

3.4. System Landscapes and Deployment

The SAP R/3 development introduced special clients and generalized the development in to system landscapes. It should be noted that the new NWDI does not decouple completely from the SAP R/3 system. In fact, a huge amount of development still needs to access functionality from the SAP R/3 system. This poses interesting software logistics question such as a) what should be the system landscape for NWDI? and, b) when can the software developed in the different landscapes be brought together for integration? NWDI solves at once the development issues of client side software and server side software. Clearly support for server side software is necessary since most Business Applications have a significant server side presence. Further NWDI also has to solve the issues of consolidation of two different streams of software development with in SAP and SAP's customers.

As discussed earlier, there can be central mechanism that allows for definition of Product definition from which the development configuration can be defined. The best practice of multi-system landscape comes when the SC dependencies are defined. Further, as the system landscape explodes on product lines and versions, it would be unmanageable to administrate every aspect of the development process.

To ease the administrative overhead, there is a central *Software Landscape Directory* (SLD) that contains information of all the systems in an organization. It is a lean highly available system that has the basic information of the systems in an organization. SLD is unique for an organization.
SLD is the location where SC and their dependencies are maintained. Development based on SC would first need a landscape to be defined.

The system landscape can have patterns that can be 1-system to n-system global landscapes. And there is a need to manage the software movement across the landscape pattern. To solve the logistics issues, NWDI introduces the *Change Management Service* (CMS) that defines and maintains a landscape in a consistent state.

The SC definitions from the SLD are imported into CMS and an appropriate landscape pattern is applied. The systems that are part of the landscape can include development systems, test systems, consolidation systems and even test servers for central testing. The pattern that is applied is referred to as *track*. Each of the system in a track that allows for change merits a *development configuration*. The development configurations that are relevant for a landscape are derived out of the track definition. Existence of an SLD allows for a central place where all the development configurations can be stored and accessed. Development configuration can have authorizations associated with them so that only few special users can modify content based on them.

Cross-technology development involving SAP R/3 and NWDI is solved by creating parallel landscapes in the R/3 landscape and the DI landscape. The designated runtime systems in the landscape would validate the cross-talk between the R/3 components and the components delivered out of NWDI. For now, the integration across the different development landscapes occurs at these points of test.

3.5. Development Process

To achieve the effect of an integrated development system like SAP R/3, effort has been spent to create the *NetWeaver Developer Studio* (NW DS) that is based on Eclipse Framework for a client based development environment. For regular Java developers using the NW DS, the development process would be as follows: [*DI Dev Process*]

1. NW DS connects to the SLD to discover the list of available development configurations
2. The developer chooses a development configuration of the SC for the development steps.
3. The NW DS uses this information to set the client machine's environment to reflect the server side environment along with the dependent components and the build tools.
4. NW DS uses versioning system of DTR to make the changes. The changes are collected in activities.
5. Check-in the activity.
6. The developer collects the set of activities that can participate in an activation request that would trigger an activation step on the CBS.
7. On successful build and integration into active workspace, the activities are ready to participate in logistics to the next system in the landscape. The developer can indicate the collection of activities and execute a release step.
8. The administrator has to execute the import step in the receiving system.

Intermediately, it is also possible that the CMS deploys archives onto central test systems in the landscape and the user can perform tests. Clearly, software from the active workspace is guaranteed to build but has to be tested for semantic correctness externally.

Concisely the entire process can be represented as the following diagram:

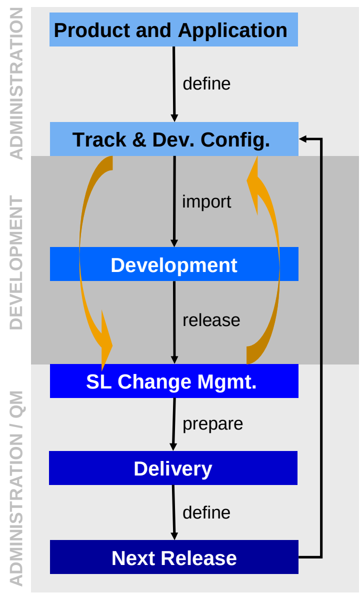

Figure 2: Lifecycle Management with NWDI

There are close parallels between the DI Dev Process and R/3 Dev Process (ref. Sec2.3). This facilitates migration of developers from the R/3 world to DI world. With application development becoming more declarative than programmatic, the R/3 developer is quickly able to appreciate and relate to the DI semantics of change management and so process-migration is achieved although the underlying technologies and application development are different.

For more information on the SAP NetWeaver offering, the public websites [5], [6], [7], [8] and education [9] can be explored.

4. Conclusion

The evolution of SCM techniques for Business Application development at SAP has come a long way. NWDI generalizes the development process for the current generation software development bringing along with it a synthesis of ideas that are from SAP R/3 along with the latest from WebDAV/DeltaV-based versioning systems.

4.1. Why an In-house SCM system?

Commercial SCM systems have a domain much broader than the requirements for SAP Application Development. And dependency on an external vendor impacts the cost of ownership of the solution for a customer and also impacts the variability of the system. Further, there is a major point of process migration of development from the established R/3 systems to the current technology that is not present in the current SCM systems. To shortly list the issues that SAP sees as important for which there is no clear support from the generic vendors are:

1. *R/3 Best Practice: Landscape for application development*. Business Applications need software to be staged for development and testing before productive use. Further global landscapes that are in practice multi-site need logistics support that are expensive in current generic SCM solutions.
2. *R/3 Best Practice: Object Checks*. The syntax checks realized through activation principle guarantees the existence of a software source state that corresponds to a build result.

3. *R/3 Development Process Migration*. The customers who are used to the SAP R/3 Development process look forward to similar processes in the new world.
4. *Highly integrated environments*. There is a tight integration between the design environments, the application development and the runtime environments. Customers prefer dedicated integrated systems rather than generic systems that introduce another level of administrative complexity.
5. *Cost of Ownership*. By giving a complete solution that handles many of the customer requirements through in-house offerings, the total cost of ownership reduces and is then a good selling point.

4.2. NWDI Answers for Lifecycle Management

In Section 1, the departure from classical SCM was discussed and differentiating factors were indicated. Revisiting those points, we have:

1. *Refinement of System Model to handle components*. The component model of NWDI addresses the formal mechanism by which the software structure can be defined that has implications on the design time, run time and deployment time.
2. *Simplification of selection for user workspaces*. The *development configuration* that gets defined in the process of landscape definition and centrally registered is the solution for giving users a consistent and simple mechanism for selection of resources for the user workspace. In fact the development configuration prepares the user's environment to more than just source code modification but also to include the build environment that is available on the CBS.
3. *Support for Central Test System*. CMS landscape definitions allows for definition of Central Test Systems that can have automatic deployment of the built archives.
4. *Support for Layered Development*. The DI Landscape patterns scale for multi-level software development.
5. *Support for Customer Delivery*. The versioning system is inherently multi-site enabled. The landscape thus replicates itself at the customer site to enable customer to make use of the DI features and to handle the changes that can come on upgrades or Support Packages.
6. *Process Migration*. Clearly, this is an important part for SAP. DI allows for a close mapping of the practices that were followed for the R/3 Change process. This means that existing customers do not have to learn an entirely new process to make changes in the new technology offerings from SAP.

Summarizing, the evolution of NWDI sets the stage for SAP to deliver next generation business applications. New technologies bring with it new challenges. As noted in [2], the readiness of the customer to accept the potential additional burden that comes with using new features is the place where SAP needs to invest in. By investing in an SCM system that is tuned to SAP and SAP customers' immediate requirements, SAP has made significant progress in setting the direction for current and future business application development.

References

[1] P.G. Daly Why Software Configuration Management?
http://www.intranetjournal.com/articles/200303/ij_03_12_03a.html

[2] Jacky Estublier et al. "Impact of Software Engineering Research on practice of Software Configuration Management." IEEE TOSEM 2005

[3] SAP R/3 Change and Transport Management by Sue McFarland Metzger and Susanne Roehrs, BPB Publications, 2000

[4] Susan Dart. “Spectrum of Functionality in Configuration Management Systems.” CMU/SEI-90-TR-11 ESD-90-TR-212. http://www.sei.cmu.edu/legacy/scm/tech_rep/TR11_90

[5] SAP: www.sap.com

[6] SAP Web Application Server on SDN: https://www.sdn.sap.com/irj/sdn/developerareas/was

[7] http://service.sap.com/NetWeaver → SAP NetWeaver in Detail

[8] http://service.sap.com/J2EE → Java Development Infrastructure

[9] http://www.sap.com/education/ → i.e. ADM200

[10] SAP, R/3, mySAP, mySAP.com, xApps, xApp, SAP NetWeaver and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries all over the world.

Author Bio

Yuvaraj Athur Raghuvir is part of the SAP NetWeaver Architecture Team at SAP. Yuva has 9 years of experience in Technology & Product development. Prior to joining the SAP NetWeaver Architecture team, Yuva was the Development Manager for the NW DI Organization at Bangalore. He holds three master degrees: Mathematics and Physics from Birla Institute of Technology and Science, Pilani, India, and Computer Science from Indian Institute of Science, Bangalore. Additionally, he holds a Post Graduate Diploma in Intellectual Property Rights from National Law School of India University, Bangalore.