CM: The Nerve of Your Life Cycle Management

NEUMA

Copyright Information

CM: The Nerve of Your Life Cycle Management

by: Neuma Technology Inc.
#51 - 5450 Canotek Rd.
Ottawa, ON K1J 9G3
Canada

Tel: (613) 749-9450

http://www.neuma.com
email: sales@neuma.com

About Neuma Technology Inc.

Neuma Technology Inc. provides an integrated tool suite that helps manage the automation of the software development lifecycle. Customers can rely on Neuma's wealth of experience and understanding of the complexities of software lifecycle management to assist them in delivering quality products to their markets.

The company's flagship product, CM+ is a high-performance Software Configuration Management System which provides an automated environment for the management of quality software projects. Based on a process-oriented database and workflow technology, CM+ provides reliable configuration management capabilities, within a tightly integrated set of applications. Unique approaches to managing software releases and change packages, and a fully scaleable suite of applications differentiates CM+.

The integrated applications of CM+ include:

- Version Control
- Change Control
- Configuration Management
- Build and Release Support
- Problem Tracking
- Activity Tracking
- Requirements Management
- Document Management

Contacting Neuma Technology Inc.

For additional information, e-mail Neuma at *support@neuma.com* or visit our Web site at *www.neuma.com.*

This article was originally published in the April 2005 issue of *CM Crossroads Journal.*

CM: The Nerve of Your Life Cycle Management

Most CM professionals, I think, would agree that the CM database is not just another component of Application Lifecycle Management (ALM), it's really at the heart of the matter. I'd like to say that the CM function shares the same stature - it's not just another component of your ALM - it's the nerve. In fact, I'd like to go one step further and say that the integration of CM and Data Management (DM) capabilities, when done properly, can transform your ALM environment into a next generation engine that will empower each component of your environment.

Perhaps you view CM as just another component of your ALM suite. Let's explore a different approach by understanding CM as a capability that can be applied to the various components.

The ALM requires management tools spanning a wide spectrum. From "Request Tracking" and "Requirements Management" through to "Deployment Management", "Problem Tracking" and "Customer Tracking", there are a range of component tools required to manage the lifecycle. Internally, we use our own tool [CM+] for:

- Request Management
- Requirements Tracking and Traceability
- Projects and Activity Work Breakdown Structures
- Document Management
- Software Version Control
- Change Management
- Baseline and Build Definition
- Problem Tracking
- Test Case and Test Run Tracking
- Build and Release Management
- Deployment
- Customer Tracking

That's quite a list of tool components. So, where does Configuration Management fit in? Is it all of these, some of these, none of these? Configuration Management is a backbone technology.

We carefully choose the areas to which we apply Configuration Management. And how much CM we use for each area is carefully customized. Here's how we apply CM in our shop.

We use CM for Source Code Management - no surprise. Tracking thousands of files and directories requires version identification, baseline definition and context-sensitive ways to view the source tree. And we need control and auditing of the changes to the configuration. Because there are so many objects to track, we need a way of hierarchically collecting and navigating them. Because changes often involve more than one file and/or directory, we need a means of collecting newly created revisions so that we can deal with them as a group (i.e. a change). Because changes are so frequent we also think it's a good idea to be able to track several minor increments without having to redefine a new reference point, or baseline, nearly as frequently.

We also use CM for Requirements Management. In many ways it's much the same as for source code management. Again, though not necessarily files and directories, we have dozens, hundreds or perhaps thousands of requirements, hierarchically arranged. We need to track the various revisions of each requirement, and to collect together revisions which fall out of a given request or focus. Baseline definition is critically important, as is change control. But we don’t have the same need for increments in between baselines. We do have the need to be able to specify a given baseline, or even a rule-based alignment, and to view and navigate that easily. We generally don’t use files to define each requirement (apart from attachments), but even so, if our mission were only to manage requirements, we’d certainly require a strong CM capability.

Test cases are yet another area requiring CM. Test cases need to be arranged in functional groups, but benefit from automatic number assignment rather than requiring that each test case be given a name, as is typically done for source files. Test cases evolve in much the same way as source code - multiple revisions until the test case is properly defined, and branches when the test case is different in one major release than in another. Although change grouping might be useful, it is less of a requirement because test cases are typically modified individually or according to their functional grouping. Baselines are important so that we can relate the correct test case revisions to the test runs performed on particular builds. We need to be able to track things like which test cases were performed in a given test run, and when did the test case last fail, or last pass successfully. The test case has to be traced to a requirement, a feature, or a problem report. How CM is applied to Test Case management varies more widely from project to project than its application to Source Code management. So more flexibility is required for a project to implement its CM plan for Test Case management.

Document Management and CM

So what about documentation? Isn’t it just another form of source code? Well yes, and no. Let me clarify that: it depends. Documentation is complex. In fact, I’d say that we take all of the things that we’re not quite sure how to characterize and call them either documents or data, and of course XML is even helping to blur the distinction. We have Product Documentation, Functional and Design Specifications, Status Reports, Process Documentation, Technical Notes, Customer Communications, and so forth. And what do we call these - documentation. So, what do we do at Neuma?

First we have our **Product Documentation**. This is a lot like another form of source code, other than the fact that it has to be massaged so that it can be presented in several different formats. So we treat it like source code. Change control, version control, hierarchically arranged, baselines, etc. In fact, we don’t distinguish, from a CM perspective, the handling of source code and Product Documentation. They even share the same source tree. Just as with source code, we have to deal with branching at some point because the release 1 documentation is different from release 2 and may evolve separately, or will, at least, require separate maintenance.

What about **Process Documentation**. This is much the same. We treat our Process as a product. The difference is that it has it’s own product tree, separate from the product(s) we’re developing. But it still needs to be tracked much the same. It will branch on different time lines than the product(s). The "process product" road map will tell us how and when it will branch. Each application product will have it’s own road map for branching. And that will be different than for the process product road map, which may well apply across all development projects.

Incremental Specifications

So, the same for Functional and Design specs? No. Whereas product documentation has a life of it's own, specifications are used to transform the product. Let me clarify something here. I'm talking about the functional and design specifications which describe Changes to the product: incremental specifications.

The full Functional Spec for the product is part of the Product Documentation. The full Design Spec for a subsystem or component should be treated much like product documentation as well. However, rather than having 25 designers, with different writing skills, trampling through the design specs as they make changes, we have them write incremental design specifications which indicate how the design is changing. That's what they know about the best. Technical writers are much more able to knit the design changes into the full Design Spec. And they can do it in batches. So if 30 changes affect a particular component, in a given release, we don't need to describe the component at each of the 30 states. Perhaps it is updated after the first 20 changes and then later on when design changes are fairly much completed for a release. That way, if a change has to be rolled back because it was decided that it was a threat to security, or to stability, we don't have to do double the work. Although this would never do in some organizations, it suits us well.

How do we handle these incremental specifications? We attach them directly to the activity/task record. Take a WBS (Work Breakdown Structure) for a specific project, typically a new product release. All of the activities are clearly identified. Ideally they have a clear set of objectives/deliverables and are typically assigned to a single person or small group for implementation. Some organizations break the activities down into fine granularity activities (often called tasks). Others leave them in bigger clumps. We number our activities using a dumb numbering system (a.1200, a.1201, etc.). The WBS identifies the activities that will be executed to transform the product from one release to another. The incremental specifications document the transformation from one release to another.

So we attach our incremental specifications to these activities. In effect, the WBS becomes a live growing super-change specification for the release. If you want to know what the change is, or will be, as a result of a give project, take the appropriate part of the WBS and extract the documentation for it. From a design perspective, extract the incremental design specs. From a functional perspective, extract the incremental functional specs.

We typically will work in one of two modes - different activities for creating the specification and for doing the design and implementation, or one activity to cover both. In the former case, we just select the "design" or "feature" activities to get the design or functional changes respectively. In the latter case we either have a single document with both feature and design sections, or we have separate functional and design documents attached to the same activity.

We have no need to implement a naming policy (the activity id suffices), and no need to make sure activities are referencing the right document identifiers. But, we still have a need for some CM. We need to track revisions of each activity document, through to approval. But we don't need to group activity documents together into change packages - each document pretty much stands on its own. We don't need to group them into a separate navigation hierarchy - the WBS does that already. We're not concerned about baselines - the latest version of the WBS documents will give us the most accurate incremental documentation, and last week's collection is really not of much interest to us. So we're concerned with version control, some approval management and automatic identification of the documentation, through the activity id. But we're not concerned about branching.

We’re not going to create a release 1 version of an activity and a different release 2 version. We may have two different activities dealing with the same area in two different releases, but each describes a separate feature or design. No need to deal with branching.

More Documentation

Some of our documentation falls into another, more generic, category. Things like status reports, technical notes, sales feedback, etc. We call this general documentation and have a specific Document Management component to deal with it. We’re concerned here with document identification, and in some cases with version control. A sales feedback document may be a one-time contribution. We want to identify it as sales input and so we use a prefix such as SAL- and then ask our CM tool to number all such documents sequentially. Similarly, we may have a TNO- prefix for technical notes and a PSR- prefix for project status reports. We add a new prefix as we find necessary and let the CM tool do the number assignment. We perform revisioning of the documents, but generally hide most of this from the user in the normal course of events, other than the version identification tag. We allow hierarchical organization, perhaps for documents pertaining to a specific sale, but we don’t require it.

We also have **attachments -** things like diagrams, debug output, email attachments - which need no version control, no hierarchical organization and no real identification other than to be able to reference them as an attachment.

Finally, we have "short documents" or **notes**. Things like problem descriptions, an email thread or chat script from a customer interaction, that sort of thing. These we attach directly to the corresponding database records as "note" fields. We don’t do version control on them at all. But we still need to organize them and the database capabilities of our CM tool support that adequately. I guess we’re starting to blur the distinction between data and documentation at this point.

We have all sorts of documents, and very different CM requirements for each type. If the CM tool were to force us to use "source code control" on all of these, we’d be treading water. CM must be provided as necessary, but it really needs to exist as a set of capabilities which may or may not be applied to the various documentation sets.

One last set of documentation that we’ll often refer to is **generated documentation**. We don’t need to do CM on this type of documentation unless we are unable to re-produce it at will. It’s the CM and other capabilities for the underlying objects which are important here. Release note summaries, requirement baseline documents, status reports, etc. If we can’t easily reproduce these, we may choose to generate them and then store them as a generic report document, similar to the Project Status Report (PSR-) documents above. Otherwise, we only need to ensure that we can reproduce them at will.

How much CM is too much?

So how much CM do we need to do? Should we version control all of our data? What about baselining it?

We could, if we wanted to, do version control of all of our requests so that we could go back to any point in time and tell our customers this would have been the status of your requests at this point in time. We don’t find that useful so we don’t do version control of our requests. Instead, at a particular point in time, say quarterly, we’ll generate a report for our customer and then save it as a CUS- document, perhaps collecting all of the reports for a particular customer under a specific DIR- container. Then we have every report we’ve ever sent to our customers and we don’t need to do any CM on their requests.

I’ve seen tools that will even do version control of a Problem Report rather than just maintaining a running log of events for the problem. (I’m NOT talking about tracking different instances of the problem for different products or even different releases.) For me, this is overkill. For you it may be a different story. But that’s why we need CM as a capability which we can apply to various objects in different ways, according to our process requirements.

Similarly, we don’t "baseline" all of our data. But we do have the capability of creating a "checkpoint" file which captures the exact state of the repository at a given point in time in just a few megabytes. That’s good - if we want to prove that our data has not been tampered with after the fact, we can view our repository from the perspective of a given checkpoint file. But that’s what it is. It’s a snapshot in time. We don’t switch back and forth between checkpoint contexts and do queries between checkpoints. They’re different from baselines. Perhaps a baseline is to CM as a checkpoint is to data. But having the checkpoint capability does take away the reasons that we might otherwise have for applying CM to all of our data.

Configuration and Data Management

So let’s return to our original discussion.

There are objects that we perhaps will agree don’t need Configuration Management - problem reports, build records, change records, activities, users. I’m sure I could make a case for CM for some of these things, but I won’t until I really see a need. If the CIA needs to track revisions of a user so that it knows what permissions and brain implants the user had on a specific date, we’ll adapt our user management to permit that. In fact, you might find that version control of users is not so far out when you start to consider staff relationships. But it’s a capability that has to be addressed as needed, and in such a way as to keep things as simple as possible.

Application Life-Cycle Management depends not only on CM capabilities, but other generic data as well. At times, it’s hard to separate CM from DM (Data Management). A new revision of a requirement is really just another requirement record that is tied to it’s predecessor requirement revision record, isn’t it? We’re back to our yes and no response. Databases generally don’t understand concepts like revisions, history and baselines. Revisions of a requirement form a history. They can be collected into baselines. And although it’s ultimately the database that represents these relationships, it’s CM that understands them. Without the CM, we need a data interpreter.

Advanced DM tools (or Hybrid Databases, as I might refer to them) will let us express all sorts of data, data dependencies and data relationships. Good CM tools will let us look at data from a specific configuration viewpoint. It will let us specify revisions and branches, and so forth. An integrated CM and DM tool will do so much more. Try storing the "include" relationships of each file for each revision. Any database will let you do that with the appropriate schema. It’s just that when you get a half-million file revisions that you realize that the 50 million include relationships are causing a bit of a performance issue.

Take an integrated CM/DM tool that will automatically difference and restore the include list only whenever it changes. Or take an integrated CM/DM tool that will let you compute the "affected" source files, not for a specific configuration, but for an arbitrary rule-based configuration that changes over time, even when the include lists which dictate the "affects" relationship doesn't. This integrated entity understands data and CM.

Show Me the Power

Integrated CM/DM will be more prevalent in next generation CM systems. These tools will allow complex data queries on CM data. Here are a few of my favorites.

- **Problem Fixes Missing From a Release Stream**. Take a set of files and two development streams. Go back through the history of each file in each development stream and look at the change records used to produce each revision. Now identify the problem reports that were addressed in one stream (i.e. referenced by the changes) that were not addressed in the other stream. Now I know which problems which were fixed in the one stream for a set of files (say my whole product source tree), that potentially need to be addressed in the other stream.
- **Build Comparisons**. Take two build record definitions an old one and a newer one. Expand them and subtract the history of the older one from the history of the newer one. This should give me a set of intermediate (and final) file revisions. Convert these to a set of change packages and identify the features implemented by those change packages, the problems fixed by them, the developers who performed the changes, etc. Now I've got the basis for generating my release notes in terms of one build and an older one.
- **Layering for Re-use**. Take a layer of a product that you want to share in the development of another product. Identify all of the files not in that layer that the files of the layer depend upon, within a given context. That will show me the obstacles to re-use of that layer of software in another product. As soon as I eliminate those "upward" dependencies, I have a potentially re-usable layer of software.
- **Customer Request Progress**. Take a customer's current build and the build you're planning to deliver. Compute the set of problems reports and features addressed between these two builds, as above. Now trace these back to the requests that have been raised by that customer which have spawned problem reports and feature activities. This intersection identifies for the customer which requests have been addressed in the new build.

I've had the pleasure, over the past quarter century, of working with environment tools having a healthy interaction between CM and DM. There's no doubt that each of the ALM components has benefited. If your CM solution is currently just one component of your management suite, you may have room to make some quantum leaps. What forms the nervous system of your management environment?

Joe Farah is the President and CEO of Neuma Technology . Prior to co-founding Neuma in 1990, Joe was Director of Software Architecture and Technology at Mitel, and in the 1970s a Development Manager at Nortel (Bell-Northern Research) where he developed the Program Library System (PLS) still heavily in use by Nortel's largest projects. A software developer since the late 1960s, Joe holds a B.A.Sc. degree in Engineering Science from the University of Toronto. You can contact Joe at farah@neuma.com