United Business Media

Dr. Dobb's

ALM Sourcebook

Entire contents Copyright© 2010, Techweb/United Business Media LLC, except where otherwise noted. No portion of this publication may be reproduced, stored, transmitted in any form, including computer retrieval, without written permission from the publisher. All Rights Reserved. Articles express the opinion of the author and are not necessarily the opinion of the publisher. Published by Techweb, United Business Media Limited, 600 Harrison St., San Francisco, CA 94107 USA 415-947-6000.

The Requirements Payoff

Getting a project's requirements right from the start can speed development and ward off problems

By Karl Wiegers

Investing the time to create better requirements for a software project takes a major leap of faith, since people tend to think the extra up-front work will just delay development. That's generally true, but getting requirements right also prevents problems later that can not only delay projects but lead to their failure.

Incomplete requirements and specifications, as well as changing requirements, are a main cause of project distress. For example, the FBI abandoned its Virtual Case File case management software project in 2005, along with $170 million and 700,000 lines of code, in a large part because of poorly defined and slowly evolving design requirements, according to a scathing report by Department of Justice's Inspector General Glenn Fine. "It was a classic case of not getting the requirements sufficiently defined ... from the beginning," Fine says in the report. "And so it required a continuous redefinition of requirements that had a cascading effect on what had already been designed and produced."

An error, omission, or misunderstanding in the requirements forces developers to redo work they've done based on the incorrect requirement. Thirty percent or more of the total effort spent on most projects goes into rework, and requirement errors can consume 70% to 85% of all project rework costs, according to a 1997 study by Dean Leffingwell. Development teams often implement functionality that someone swore they needed, only to find that no one ever uses it. Techniques that can prevent this wasted effort are a solid investment.

What's To Be Gained

Good preliminary requirements help CIOs make effective business decisions regarding which projects to fund. Well defined and documented final requirements are critical to letting developers devise the most appropriate approach, estimate the effort needed to execute an iteration cycle or complete the project, incorporate changes that will deliver maximum customer value, and develop accurate test cases to verify the implemented functionality.

Putting more effort into requirements development can actually accelerate software development. At a large insurance company I worked with, increasing the front-end requirements effort from 19% to 33.6% of total effort reduced projects' overall effort and cost by an average of 4%. Another company trained its development teams on requirements engineering and implemented improved requirements practices. In a year, the share of projects that were behind schedule dropped from 21% to 12%, and the total days those projects were behind dropped from 1,738 to 518.

While no one can predict what ROI you're going to get from your investment in better requirements, consider these questions as a first step in determining payback:

- What fraction of your development effort do you expend on rework? Some rework is unavoidable and adds value, but a lot of rework is nothing but wasted effort.

Dr Dobbs

EDITOR-IN-CHIEF
Jonathan Erickson

MANAGING EDITOR
Deirdre Blake
COPY EDITOR
Amy Stephens
CONTRIBUTING EDITORS
Mike Riley, Herb Sutter
WEBMASTER
Sean Coady

VICE PRESIDENT, GROUP PUBLISHER
Brandon Friesen
VICE PRESIDENT GROUP SALES
Martha Schwartz

AUDIENCE DEVELOPMENT
CIRCULATION DIRECTOR
Karen McAleer
MANAGER
John Slesinski

DR. DOBB'S
600 Harrison Street, 6th Floor, San Francisco, CA, 94107. 415-947-6000.
www.drdobbs.com

UBM LLC

Pat Nohilly Senior Vice President, Strategic Development and Business Administration
Marie Myers Senior Vice President, Manufacturing

TechWeb

Tony L. Uphoff Chief Executive Officer
John Dennehy, CFO
David Michael, CIO
Bob Evans Senior Vice President and Content Director, InformationWeek Global CIO
Joseph Braue Senior Vice President, Light Reading Communications Network
Scott Vaughan Vice President, Marketing Services
John Ecke Vice President, Financial Technology Network
Beth Rivera Vice President, Human Resources
Fritz Nelson Vice President and Editorial Director

techweb™

- How much does a typical customer-reported defect cost your organization? A system-test defect? One of my clients spent an average of $4,200 to deal with each customer-reported defect. That was 21 times more than it cost them to discover a defect through formal inspection.
- What fraction of user-reported defects and what fraction of defects discovered through system testing stem from errors in requirements? Defect root-cause analysis is an excellent technique to gauge how much you could gain from improving quality.
- How much maintenance cost, from defect correction and unplanned enhancements, can you attribute to requirement defects such as missed requirements?
- How much could you shorten your delivery schedules if your project teams could reduce requirement defects by, say, 50%?

Practices that result in fewer requirement defects will reduce the amount of development rework your teams must perform. This provides immediate payback through reduced development costs and quicker time to market. Techniques that get your analysts and customers working closer together also lead to products that better meet customer needs and need less reworking. All of these approaches have the potential to increase customer satisfaction.

Steps To Improve Requirements

In the quest for good requirements, first, make sure you have appropriately skilled and trained business analysts who can guide the requirements development and management activities on each project.

As your teams get up to speed and requirements become more sophisticated, having the right tools can be a big help. Lots of requirements management tools are available, including IBM Rational RequisitePro, Micro Focus CaliberRM, Microsoft Visual Studio Team System 2010, MKS Integrity, and Ravenflow Raven 2010.

With new projects, I begin at the top by clearly establishing the business objectives. Defining the product vision and project scope help ensure that all the work done aligns with achieving those objectives. This includes assessing your current practices and identifying areas that aren't delivering desired results. Improvement might require writing new processes, modifying current processes, and selecting new templates for key requirements deliverables.

Next, identify distinct communities or classes of users and determine who will serve as the voice of the customer for each such user class. To engage appropriate customer representatives, I recommend designating "product champions" who are key customer representatives engaged on the project on an ongoing basis. This is much like the Agile development concept of the on-site customer.

The "use case" technique, which focuses on expected usage rather than on product features, is an excellent way to explore user product requirements. It should be clear how the use cases will align with business objectives. If they don't align, there's a problem. But use cases aren't sufficient in every situation. Your analysts also need to derive the functional requirements that developers will implement to let users perform the use cases.

Don't just assume the requirements are correct: validate and verify them

In addition, explore and document nonfunctional, quality-related requirements such as usability, security, reliability, and robustness. These attributes are vital to customer satisfaction. Ongoing prioritization of the requirements also is crucial. Think of the backlog of pending requirements as a dynamic list, not a static snapshot frozen in time.

Don't just assume the requirements are correct: validate and verify them. You can begin "testing" your software as soon as you've written the first requirement. On one project I was involved in, I wrote a dozen or so functional requirements, then another dozen or so test cases based on my vision of how the code would operate. In writing the test cases, I discovered an error in one of my requirements. I generally find these sorts of errors, omissions, and ambiguities in my requirements at this point.

A well-structured and rigorous peer review or inspection of requirements documentation is also a good investment. And finally, an effective change management process will help ensure that your project delivers the product that customers need.

Of course, the greatest investment you can make is the time you spend eliciting, analyzing, specifying, validating, and managing requirements. Time spent on these efforts is likely to accelerate a project and make it run more smoothly.

These practices aren't free, but they're cheaper than waiting until the end of a project or iteration, and then fixing all the problems. The case for solid requirements practices is an economic one. With requirements, it's not "You can pay me now, or you can pay me later." Instead, it's "You can pay me now, or you can pay me a whole lot more later."

Return to Table of Contents

Keep your team on track with Agile Tools and Techniques from CollabNet

From the start line all the way to the checkered flag, we can help your dev team:

- Ignite your agile knowledge with Certified Scrum Training
- Kick your development team into high-performance mode with TeamForge™ and ScrumWorks® Project Management tools
- Be first to the finish line by enabling collaborative agile development

Join the race with a free trial at: www.collab.net/winwithagile

INFO@COLLAB.NET | WWW.COLLAB.NET | +1 650-228-2500 | 888-778-9793

COLLABNET®

Software Builds and the Virtual Time Machine

If you are not yet using virtualization in your build environment, then it's time to get moving.

By John Graham-Cumming

If you are not yet using virtualization in your build environment, then it's time to get moving. Applying virtual machine technology to a build system, just as many have done in the test environment, means being able to give instant answers to questions like "Can you give me the log files for build XYZ?", "Which system headers were used for project ABC?" and it means always being able to say "Yes" when someone asks for a rebuild of an ancient software version.

Fundamentally, software build management is about the process of getting all the right software components—a particular version of the source, the specific tools (such as compilers and linkers) and the right third-party code (such as system libraries)—on to a machine with the right operating system and running the build script.

Once the build script has run, the object code generated has to be extracted, source code tagged, and precious log files and other output saved for later use. Frequently, a complete record of the configuration of the build machine is needed (including tool, library, and OS versions) as well as a bill-of-materials listing all the source code that went into the build.

Complete, detailed records are necessary because the build machine's configuration will inevitably change as time goes by and other builds are performed on it.

But what if that were not the case? What if every time the build finished, the entire machine was placed in a vault in case the build had to be reproduced? Each time the build manager performed a build, they'd have to go out and buy a new machine.

Sounds ridiculous? Yes. But wouldn't it be great?

There would be no problem reproducing an old build when an important customer demands a bug fix, or a security headache means rereleasing old code. No problem grepping an old log file. No problem asking questions about what exactly went into a build. No need to guess what the configuration was, or sweat trying to reproduce an old build.

Happily, virtual machines turn physical hardware into files. And like any other file they can be backed up, versioned, and reloaded when needed.

If builds are performed on a virtual machine, that machine can be saved and tagged with each build. Or a virtual machine "snapshot" can be taken and tagged with the build number. You could even check the snapshot into version control when the build is tagged.

Now the configuration of an old build is on hand at any time. Just fire up the right VM to go back in time. You'll be taken right back to the moment the build finished, with the complete machine state available, and a cursor flashing at a shell prompt ready for the next command.

But keeping the complete state is not the only advantage of virtualization. Old operating systems can be made to run on new hardware by virtualizing the interfaces that the OS sees. It's no longer necessary to keep ancient hardware around—and working—if that Windows 98 VM can run on the latest virtualized Intel box.

Virtual machines also mean that pristine configurations are always available. The ultimate "make clean" is booting from a newly initialized virtual machine. A repository of configurations can be built and run on any machine available to the build team. The problem of maintaining specific machines for specific builds goes from being a hardware issue to a software one.

And the same hardware can be used to support different OS configurations, making more efficient use of build infrastructure.

All these changes mean that software production management products need to understand virtualization and help the build team manage virtual assets. If the management system is virtualization-aware, then it can control the entire process of starting the right virtual machines, getting sources, running builds, archiving output, and saving virtual machine snapshots.

The latest software production management products (including my company's ElectricCommander) integrate with the classic suite of build tools (such as SCM, scripting languages, make or ant tools, and test systems) and have been enhanced to be virtualization-aware.

In short, virtualization makes more efficient use of existing build hardware, simplifies the management of different build configurations, and gives the build manager a virtual time machine capable of taking them back to the moment when any given build was completed.

Return to Table of Contents

Software Configuration Management

Software configuration management is fundamental to top-flight process

By Pablo Santos

Software configuration management (SCM) is all about organizing, controlling, and managing change in software as it evolves. As a discipline, SCM defines a process for change control. Unfortunately, SCM is often thought of as only necessary for really big software shops undertaking really big development projects. I say "unfortunately" because, in practice, the principles of SCM can—and should—be implemented in software projects of all sizes.

For instance, a couple of years ago I moved from a large-scale, distributed development team making software for digital TVs, to a small group responsible of the maintenance of an ERP system. With not much more than a compiler, I found myself responsible for a 400,000 lines-of-code Delphi 5 project that was the code base for five different products. In a perfect world I could have hired some large group of software engineers, bought management, estimation, version-control, defect and task-tracking, and automated testing tools. Unfortunately I didn't have resources, so I had to be careful in my selection. If there's one thing I've learned in the programming trenches, it is that you can survive with very few tools—a compiler, editor, and strong version-control system. Why version control? Because you can take advantage of it to not only put your files under control, but also manage the efforts of the whole team.

So I pushed management and ended up getting Rational Clearcase LT, an entry-level SCM tool from IBM.

SCM In a Small Shop?

At first glance, it seems SCM is a big shop toy. Why bother with versions, releases, branches, and the like, if there are only four developers on your team? Doesn't that bring more overhead than benefits? The answers are: "yes," "you need it," and "no, it doesn't." Even small size teams can produce really good software if you can count on the right professionals, use the right development technologies, and organize your efforts correctly.

The main difference from a SCM point of view between a four-developer and 100-developer group is that the smaller group produces less code in the same time. But the quality must be the same, and the efforts have to be even better handled, because the resources are scarce.

In a small shop you can't afford losing some code because it was "unexpectedly overwritten"—your team has better things to do than rewriting it.

Making It Work

Before putting the SCM tool to work, I tried to order the source tree, coming up with something like Figure 1.

What was placed under SCM was already in the source tree—lots of directories with tons of files like .pas and .dfm together with some .dpr. I modified each project so that their binaries would be placed under the bin subdirectory, and the .dcu would be located in obj, so that we were able to keep the tree structure clean of intermediary files.

Our product had been under development for a long time, evolving through versions 1 to 3, so at outset we used Delphi 1. By the time I joined the company, we were compiling with Delphi 5. Since many of the .dfm files were in binary format, I decided it would be a good idea to convert them all to text format so that the SCM tool can keep track the differences correctly.

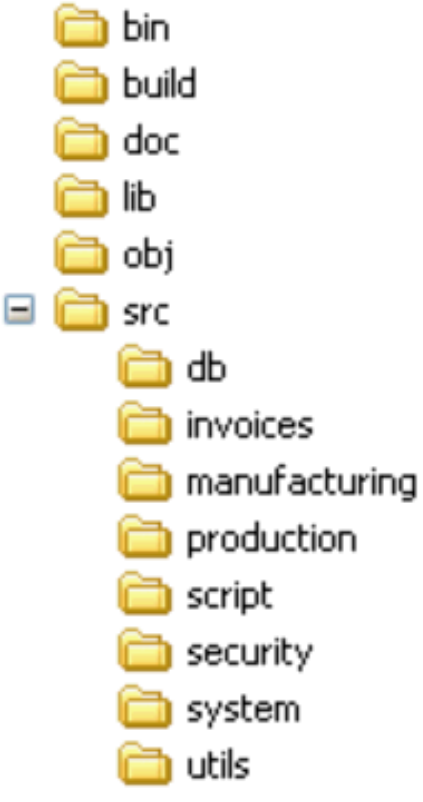


Figure 1: Project's directory structure.

Next, I tried was detecting all the dependencies that the projects had on third-party installed components, even in-house ones. I wanted the source tree to be as independent as possible from installation issues, so moving the whole src tree (and I was thinking on workspaces) wouldn't be a concern. So I placed all the .dcu from the different components (each group on its own subdirectory) under lib, and updated the projects directory configurations so they used these .dcu to compile.

In doing so, I implemented the following policy: The bpl would be copied to a bpl subdirectory under Delphi installation directory, and installed in the IDE from there (the same for every developer in whatever machine they are). The .dcu files were not an issue anymore, because they were located using relative paths. This approach not only introduced a way to handle compilation, it also provided a way to deal with external elements: Whatever can be precompiled will be used in its binary form. We ended up with a tree like Figure 2.

Figure 2: The project and component VOBs.

Alas I forgot (and still regret) to reformat the code. Before creating the repository on a SCM tool, you are handling the prehistory of your system. Whatever happened there remains obscure. Once it is under source-code control, every change is controlled. Minimizing changes is good to keep a consistent view of modifications. I you need to fix a bug which involves modifying one line but you reformat the whole file, it won't be clear what you needed to change afterwards. This will be quite confusing. Unfortunately the same happens with a controlled reformatting, the code is better, but it is almost impossible to trace back changes easily. Consequently, reformat the code before or make the reformat just after the initial code import into the SCM tool.

The Initial Code Import

To make the initial code import, I first created two VOBs (versioned object base, the project repository)—one for the products and another one for the components. (Again, we were using Rational Clearcase LT, and the terminology varies from tool to tool.) Then I imported both all the code for the project and for the different components.

Once everything was there, I compiled each of the applications (having a project group or makefile-based build system helps). If it goes well, then you have your first controlled release. So what I did was label the entire source tree with the name of the release, V151 in this case.

Once the code was there, we prepared some documents about setting up views and filling in the configspecs.

And The Process Will Follow...

Putting all the codebase under source control was the first step to bring some order into the project. But then it was necessary to introduce a whole working process together with the tools.

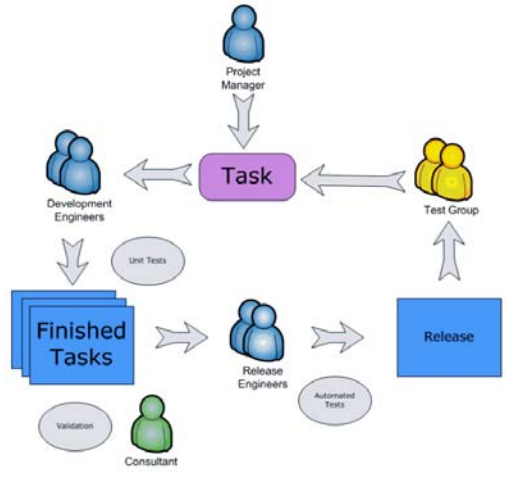


Figure 3: Task-oriented development cycle.

Our process was quite simple: Every single change to the source code had to be registered into a homemade task control application. It would assign a number to the issue (being a bug, new feature, or whatever), and with this number (and in our case a prefix) we would create a branch in Clearcase. We are using a branch per task pattern, so every single development is isolated in its own branch. It would be a big burden for several SCM tools, but not for clearcase.

So each task has its own branch and is assigned to one developer. Hence each developer gets completely isolated from the others, his own changes won't affect his colleagues, and neither he will be bothered by incomplete modifications from the rest of the group. You will get this benefit using any SCM tool, is just a consequence of workspaces, but if you add branches to the picture then developers get extra facilities like version control on their ongoing work. This is really useful once you get used to it. Figure 3 shows the development cycle we used.

Basically the project manager is responsible to introduce tasks in the in-house task control tool. Tasks are assigned to developers. Each developer works on one task. Once a developer finishes his work on a task, he undertakes unit tests. The task won't be finished until all the unit tests are passed. When we started with SCM we didn't have unit tests in place, but as we were progressing we started to introduced this technique, which proved to be helpful. We use Dunit. Once a task is finished the developer moves to another one, creating a new branch and so on.

After some time, you end up with a pool of finished tasks, with their associated branches containing all the modifications.

Implications

Since we are dealing with a business application, it is sometimes hard for developers to really know the implications on the rest of the program. Consequently, we introduced the following policy: Every single business-related task has to be validated by a business expert, a consultant in our case. So to finish a task the developer has to complete the following steps:

- Run and pass the unit tests.
- Build the application with version information included.
- Rename the executable to the name of the task.
- Place the executable in a common directory for finished tasks.
- Mark the task as finished in the task control tool.

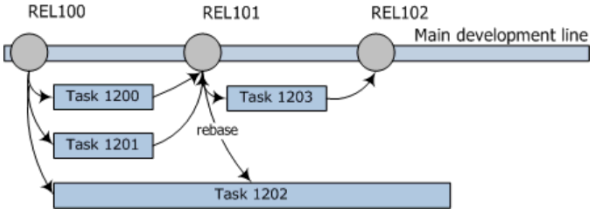


Figure 4: A task that lasts for several releases.

Periodically a selected group of consultants check the finished tasks in the tool, run the related executables to determine whether the modifications are correct, and if they agree on the changes, they mark the task as verified in the task control tool. Once there are several verified tasks, the tasks are integrated to create a new release. There are several possibilities about integration periodicity. Some developers argue in favor of daily integration, others for integrating continuously. We decided to integrate once a week. The period depends on the amount of work your team is able to produce. If your team is big enough to complete a large amount of tasks every day, then integrate once a day. In my case I need a week to have a good set of verified tasks, so we create a new release every week.

A more agile approach says that a continuous integration is better. In my opinion it depends on the nature of the software you're developing. If stability is your biggest concern (and it is in my case), then I prefer to have somebody controlling every change that is made to the software. If velocity is the main concern, then is better if every member on the team can integrate its own changes. Also having somebody responsible of the integration process hides the complexity of the task to the rest of the team, than won't be bothered with the internals of the SCM tool.

Once a new release is created, the source code is labeled and a new baseline is created. Recently we have created a test group integrated by both consultants and developers. The group has created a big set of automated tests under Rational Test Suite that checks most of the functionalities of the application. Nowadays we are not yet covering all the functionality, but we're pushing to have a quite complete test suite. So before labeling a release we run the automated test suite (now it takes about 12 hours), and if it passes then we label the release.

Once the release is labeled, the ongoing tasks are asked to be updated to the new release so that developers work against the most up-to-date release.

Planning and Release Periods

Choosing a certain periodicity for integrating has some consequences in planning. Since we have chosen to integrate once a week, ideally the biggest task a developer works on is a week long. But we always have tasks that need more time than a week to be finished.

Having a task that lasts for more than one release is not a problem, because you can always rebase your work, updating it with the latest changes from the release. But is clear that doing so introduces some overhead and complexity in the process. It is better if you can avoid it. Figure 4 shows a task that is not completed in one release period and needs to be updated when the release is created. So if possible would be better to split long tasks so that they can be completed inside the release period.

Build Automation

Again, during the finalization of a task the developer has to add version information to his executable. I started doing that modify-

```
1 VERSIONINFO
FILEVERSION 4,3,211,0
PRODUCTVERSION 4,3,211,0
FILEOS 0X4
FILEFLAGS 0X04
FILETYPE 0X1
{
    BLOCK "StringFileInfo"
    {
        BLOCK "0C0A04F4"
        BEGIN
            VALUE" CompanyName",        "CIM, s.1."
            VALUE "FileDescription",   "V4.3"
            VALUE "FileVersion",        "4.3.211.0\255"
            VALUE "InternalName", "Xmas\255"
            VALUE "LegalCopyright",    "Copyright \251 1993-2004."
            VALUE "LegalTrademarks",   "marca registrada de\255"
            VALUE "ProductName",        "BL211"
            VALUE "Fecha",               "16/12/2005\255"
            VALUE "OriginalFilename", "exe\256"
        END
    }
    BLOCK "VarFileInfo"
    {
        VALUE" Translation", 0x0C0A 0x04E4
    }
}
```

Figure 5: Contents of the .rc file.

ing the version properties of the project with Delphi. The problem is that using an SCM tool you have to check out all the project files (not only the dpr) to modify and save the version information. So I decided to create a separate version file and include it as a resource (version information is a given resource). So during compilation a .res is created with brcc tool from the .rc file, and version information is added to the executable. Figure 5 shows the contents of the .rc file.

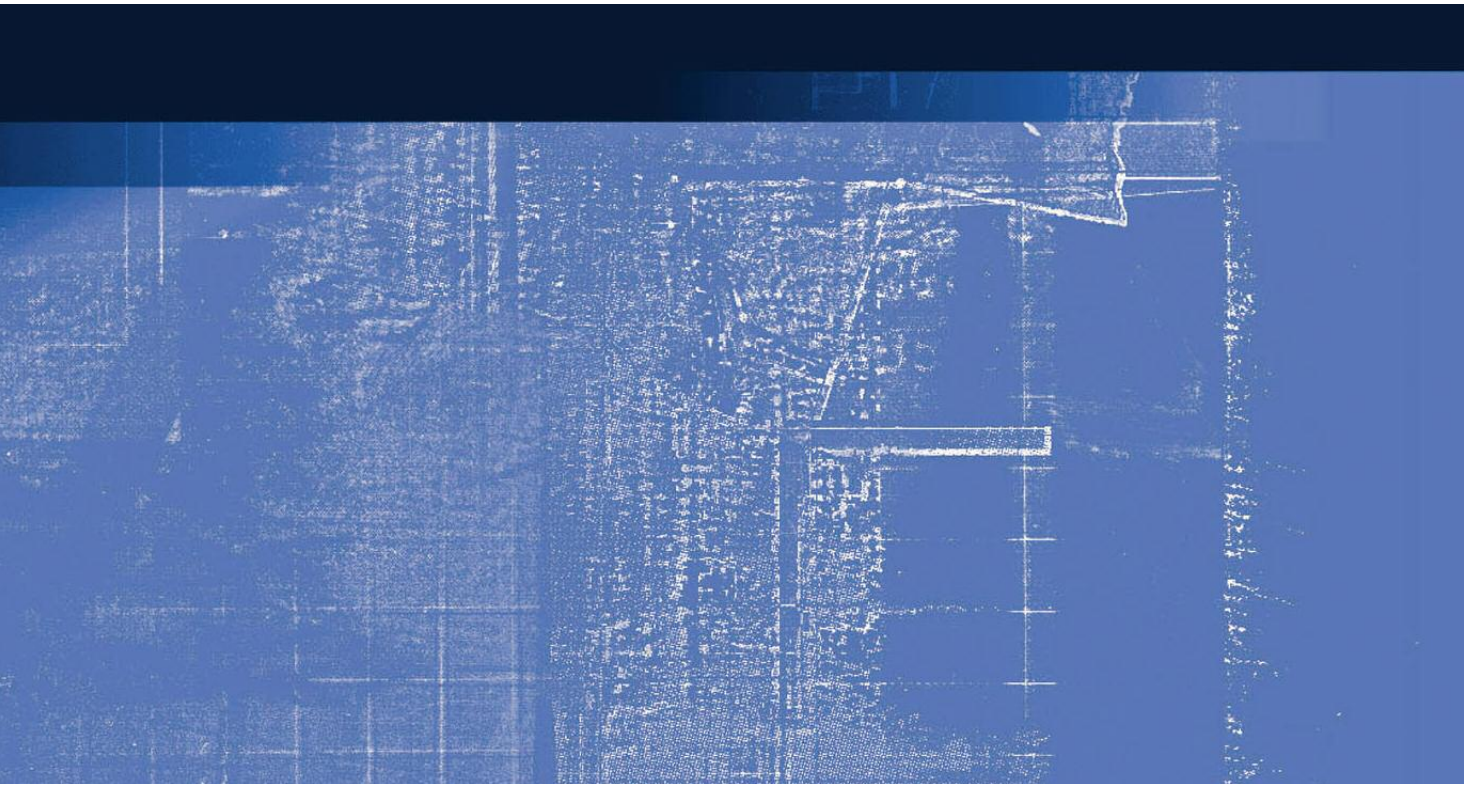
To compile the .res resource file I created a makefile, and I ended up using the makefile to compile both the finished tasks and the releases. So we compile under Delphi for normal development, but as soon as we finish a task we go to the build subdirectory and type something like: make TESTUNIT to run tests or buildtask to execute an script that will extract the task name using clearcase commands, create a .rc version information file with the task information and build our executable with that information.

Using a makefile is probably not the easiest or more fancy way to work, but we can easily create executable without debug information, run unittests or compile and include certain resources (now we include SQL sentences for different databases as resources, and an internal manager class loads the appropriate ones depending on the database you're connected to). It was specially helpful to create releases without debug information at the beginning, when we had five different Delphi projects (now we have only one dpr). Otherwise we would have to open each one of them, uncheck the debug information boxes, and compile.

Conclusion

Introducing a powerful SCM tool in a project is one of the main steps to follow if you want to increase the quality of your process. Activities will be better managed using such a tool. Finding bugs is easier because you can trace back any change done to your code base. And finally, controlling releases in a formal way introduces a work discipline that is helpful for the whole team.

Return to Table of Contents



Extending Continuous Integration Into ALM

Traditional Continuous Integration has been constrained so that it provides only a partial picture of software quality

By Maciej Zawadzki

Continuous integration (CI), the practice made popular by Agile methodologies, has seen tremendous adoption by development teams in the last few years. This wildfire-like spread of the practice has been the result of a common-sense approach to automation and information sharing. But traditional CI has been constrained so that it is localized within the lifecycle and provides only a partial picture of software quality. In this article, I examine the reason for these constraints and suggest approaches for working beyond them.

Continuous Integration Foundation

Continuous integration is a practice made up of two components:

- Team members integrating their work frequently.
- Integration not degrading code quality.

The practice is rooted in the observation that the longer developers go without integrating their work, the more painful the eventual integration. However, frequent integration is only one part of continuous integration and is not sufficient on its own to constitute the practice. To practice continuous integration, the second part of our definition must be met: Each integration should not degrade code quality. Thus, critical to the implementation of continuous integration is being able to determine code quality. This feat is typically accomplished via testing as determining quality is not black and white, but rather comes in shades of grey. The more tests we run, the more accurate our determination of the code quality will be.

There is a force that counterbalances the desire for a clearer picture of code quality. Within continuous integration, it is important to determine quickly whether quality has been degraded. Consider what must happen when we find that the latest integration decreased the code quality—the offending code change must be either backed out or corrected. In either case, we're talking about another integration and the longer we delay that integration, the more painful it is.

So we find that a balance is needed when practicing continuous integration. On the one hand, the more tests we run, the more accurate our determination of code quality will be. On the other hand, more tests mean longer test times and a longer wait before correcting any offending code change. Typically, the balance is struck by running unit (or fast running) tests as part of the continuous integration build process. This leaves a lot of testing on the table (functional tests, performance tests, regression tests, integration tests, and the list goes on). Can anything be done with these remaining tests?

What About the Remaining Tests?

Any tests not performed as part of the CI loop are typically performed manually or with the aid of scripts or tools. Even when scripts or tools are used, they usually perform only a portion of the work and have to be coordinated manually, leading to a semi-automated process. The end result is that there is a long delay between the time the code changes enter the additional testing stage (beyond CI) and the time that the results of those tests are available and acted upon. Presumably, the closer in time the discovery of a bug to the time of its introduction, the smaller the effort and cost required to fix it. Thus, there is value in reducing the feedback loop in the tests beyond continuous integration.

If we didn't have to worry about keeping the CI loop so tight, we could include additional tests as part of the CI process. Continuous integration has the promise of providing the automation framework that is needed to decrease the turnaround time on the longer running tests. More importantly, many teams that have started out with continuous integration and a fast CI loop have implemented automation of additional tests to provide a more thorough view of the quality of their code base. Running these additional tests is not as fast as running tests used in the tight CI loop and does take a substantial amount of time. The turnaround time can range any where from two hours to more than eight hours. In addition to running slow unit tests, some teams automate functional tests (which require that the application be deployed into a test environment), integration tests, system tests, and more. The possibilities to a large extent depend on the approach taken

toward the automation and the model for automating these processes outside the tight CI loop.

In the rest of this article, I present some alternative ways in which this can be accomplished. I will cover a build-centric approach in the section about Staged CI, a process-centric approach in the section about Chaining Processes, and a lifecycle-centric approach in the section about Build and Release Pipelines.

Staged CI

Staged CI is a practice where a tight loop is used for the typical CI build and one or more additional loops are used to automate a more thorough determination of the code quality. The reason for the multiple loops is that you want to keep the CI loop tight to provide feedback to developers as quickly as possible. In the tight loop, you're willing to sacrifice accuracy of the quality determination for speed. But once you have the quick feedback, you can take a little more time to get more detailed feedback from the additional loops. The end result is that for each project, you have multiple build loops in your system, one loop for each stage. Each stage is progressively more thorough and thus includes longer sets of tests.

The nice thing about this approach is that it makes it easy to deal with limited hardware and test resources. Since each stage runs in a loop, there's never more than a single instance of a stage running at any one time. If that weren't the case and you could have multiple instances of a stage running at any one time, then you'd need to have a scheduler that has knowledge about available hardware resources and manages the allocation of hardware to stage instances. Furthermore, there would need to be a way to balance the pace of stage instance creation to the throughput of the available hardware. Once common mechanism to do this is request coalescing. But the point is that by keeping each stage in a loop, you can avoid a lot of these complexities.

In a Staged CI type of setup, it is typical to have a tight CI loop that takes 15 minutes or less to run, followed by a longer loop that takes an hour or two to run, followed by a nightly build that can take five or more hours to run; see Figure 1. The tight CI loop runs quite often as indicated by the runs CI1, CI2...CI6. The second and fourth runs of the CI loop failed as indicated by the red color. The longer loop that includes long-running tests is depicted by runs L1 and L2. Notice that this longer loop runs concurrently with the tight CI loop. The entire system being used for CI and the other loops

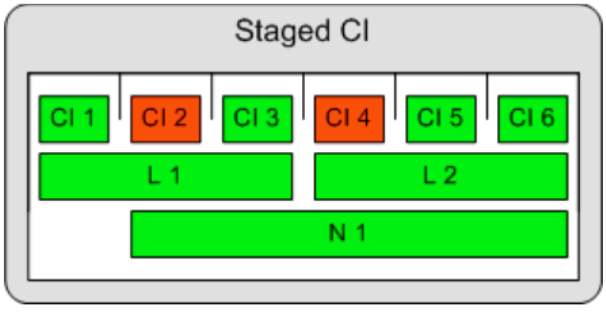


Figure 1: Staged CI.

may be made up of multiple machines that include a central server and many agent machines. All the heavy work is typically performed on the agent machines so that the system scales horizontally. The nightly build (N1) takes the most amount of time to run.

Staged CI is not really any different from nightly builds, although this depends on the reason for the nightly build and on what happens during the nightly build. If the reason for the nightly build is simply that the team does not see any value from having a tighter feedback loop, then there are some differences. But if the reason for the nightly build is to let the build run over several hours doing extensive testing of the code base to arrive at a very detailed quality determination, then the nightly build becomes an example of Staged CI. Rather than running in a continuous loop, the nightly build runs once during a 24-hour period during the night. Typically, the decision to run at night is related to the usage of resources. Presumably, if the nightly build were to run during the day, it would require access to some resources that are unavailable or being used for other purposes during the day.

Staged CI makes use of multiple build types. Let's take a look at what we mean by build types, then look at why Staged CI uses them.

To understand build types, you need to understand that most of the time when we use the term "build" we're not exact in what we mean. Usually, when we talk about a "build" we are actually talking about something more than just a build. For example, when we talk about a "continuous integration build," we're talking about a process that extracts source code from the source-code manager (SCM), compiles it, packages it, and then runs some tests on the resulting artifacts. In contrast, a nightly build may extract source code from the source-code management system (SCM), compile it, package it, then deploy the artifacts to a QA server and run functional tests. The CI build and the nightly build are just two examples of build types.

The defining feature of a build type is that it is a combination of multiple processes. Of those multiple processes, one is a build process and the remainder is made up of one or more secondary processes. So what do we mean by a build process? A build process takes source code, dependencies, environment settings, and configuration as input, and transforms them into the output. The typical output of the build process is made up of artifacts (typically a compiled binary), log files, and reports. The transformation of the input into the output typically involves compilation and packaging. However, this varies with the technology being used, as native languages include a linking step, whereas scripting languages don't have the compilation step.

Let's take a look at a CI build type and a nightly build type in light of this definition of the build process. Recall that a CI build type extracts source code from the SCM, compiles it, packages it, and then runs some tests on the resulting artifacts. I can now restate that so that the CI build type extracts source code from the SCM, performs a build, then runs some tests on the resulting artifacts. And the nightly build type extracts source code from the SCM, performs a build, then deploys the artifacts to a QA server

and runs functional tests on them (Figure 2). Each build type is a combination of build process along with one or more additional (secondary) processes.

One of the defining properties of Staged CI is that each loop (or stage) is a different build type. This means that each stage builds the source code in addition to running one or more processes. This may seem like a natural thing and you may wonder why this is worth pointing out. The reason is that this is very different from the two approaches I address next.

Chaining Processes

The Staged CI approach results in multiple stages where each stage is a different build type. As such, each stage performs a build as well as one or more secondary processes. A more process-oriented (as opposed to build-centric) approach draws sharp boundaries between the processes, so that there are no overlaps. One process retrieves the source code from source control, compiles it, packages it; in other words, "builds" the software. A second process runs the quick tests. Additional processes may provide even more functionality. This separation in processes offers greater efficiency and allows the same source code to undergo progressively more exhaustive testing.

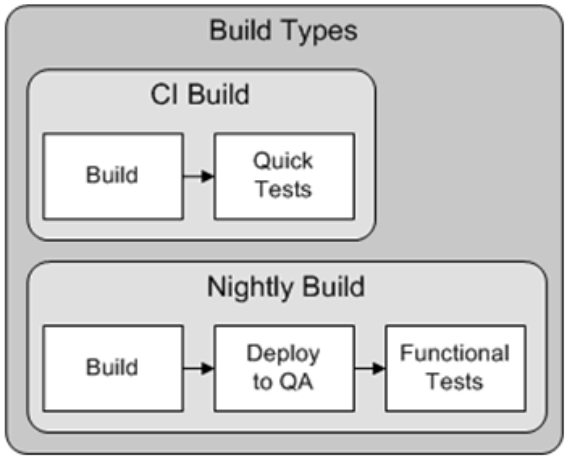


Figure 2: Build types.

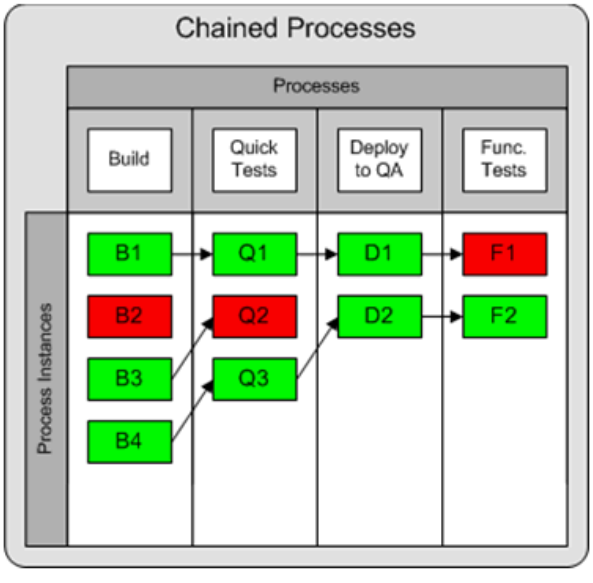


Figure 3: Chained Processes.

The idea is that all of these separate processes are executed in a chain. First, the build process is invoked. Once the build process completes successfully, the next process (Quick Tests in Figure 3) is invoked. It is at this point that we run into our first complication. The secondary processes, such as the Quick Tests process, are not build types and do not produce their own artifacts to be tested. This separation between the build and the test processes is the key to the Chained Processes approach and the main differentiator between it and the Staged CI approach. And since the secondary processes do not produce their own artifacts for testing, they need to get the artifacts externally. The artifacts are produced by the Build process. And typically, when the Build process runs, it places the artifacts it produces in a well-known location. This can be an agreed-upon directory on the filesystem, an SCM, or an artifact repository. The secondary processes need to obtain the artifacts from the well-known and agreed-upon location.

This type of artifact passing is trickier than it sounds. It is easiest to have the artifacts from the latest build overwrite the artifacts of any previous build. The alternative to having a separate location for the artifacts of each build requires that secondary processes be able to locate their intended artifacts. A simple naming convention where the artifacts of a build are stored in a directory with the build number as its name would require that each secondary process be passed to the build number so that it can find the artifacts. While this is not difficult to do, it is another detail to keep in mind.

Now that we know how the secondary process (such as Quick Tests) is going to locate the build artifacts, we can continue the walk-through. When the Quick Tests is invoked after the completion of the Build process, it retrieves the build artifacts and runs the quick test on them. These results can then be communicated to the development team in order to provide the fast feedback required by continuous integration.

Aftr a successful execution of the Quick Tests process, we would like to run the Deploy to QA process. However, if you recall from our discussion of Staged CI, a limit in the available hardware resources may mean that you can only run a single combination of the Deploy to QA and Functional Tests processes at a time. The scheduler used to schedule the execution of these secondary processes should be robust enough to provide the desired behavior.

This approach does a good job of facilitating a common Build process that provides both the steady CI feedback and feeds longer running processes like tests. One of the benefits of this approach is that downstream processes always have a build that successfully completed all preceding processes. To illustrate this point, consider that at the time we invoke the Deploy to QA process D1 in Figure 4, the latest run of the Quick Tests process is Q2 corresponding to build B3. But since Q2 has failed, and we would like our execution of the Deploy to QA process to deploy the artifacts of the latest build without any detected problems, the D1 process is linked to Q1 and B1 instead of the latest available Q2.

The Chained Processes approach does present the challenge of traceability. As each process stands alone, the linkage between successive process executions is based on parameters being passed in. If we want to determine the build and code responsible for a failed Quick Tests execution, we need to get the build number that was passed into our process execution as a parameter. Process-oriented systems built around this approach typically do not provide any built-in traceability mechanism for navigating the process chains. Likewise, process-based systems typically do not provide any built-in way of passing artifacts from one process to another in a traceable manner.

Build and Release Pipelines

While the Build and Release Pipelines approach is similar to the Chained Processes approach in that both rely on non-overlapping processes, the difference comes down to the central concept. The Build and Release Pipelines approach uses the pipeline as the primary concept, while it is the process that is the central concept in the Chained Process approach. In the Build and Release Pipelines approach, the processes are executed within the context of a pipeline.

As with a simple continuous integration system, a commit to the SCM can trigger the build process after the quiet period is met. At this point, a new pipeline is created and the invoked build process runs within the scope of the pipeline. In this approach, all processes are executed within the scope of a pipeline. The artifacts produced by the build process are stored within the context of the pipeline. It is the pipeline that provides traceability to the artifacts (meaning that the artifacts are traceable to the pipeline and via the pipeline to the build process invocation that created them).

A successful completion of the build process invokes the Quick Tests process within the context of the same pipeline. The Quick Tests process has no problem locating the build artifacts to be tested since they are available from the pipeline context; it is the pipeline that is responsible for storage and retrieval of all artifacts and other data related to the pipeline. The results of the Quick Tests process can be communicated to the members of the development team to provide the fast feedback required by continuous integration.

As in the Chained Processes approach, the process scheduler should be responsible for ensuring that the hardware is not saturated by too many concurrent instances of the Deploy to QA process. One common strategy employed by schedulers to ensure this is to combine requests for the Deploy to QA process. If two separate pipelines made requests for the Deploy to QA process but only a single process could run, the scheduler would honor the most recent request and discard the older one. But any such scheduler needs to be robust enough to provide all desired functionality. For example, the scheduler should probably not discard process requests made by users manually.

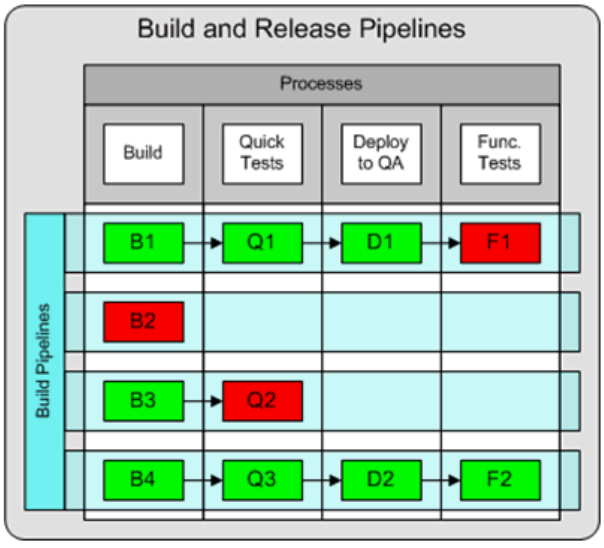


Figure 4: Build and Release Pipeline.

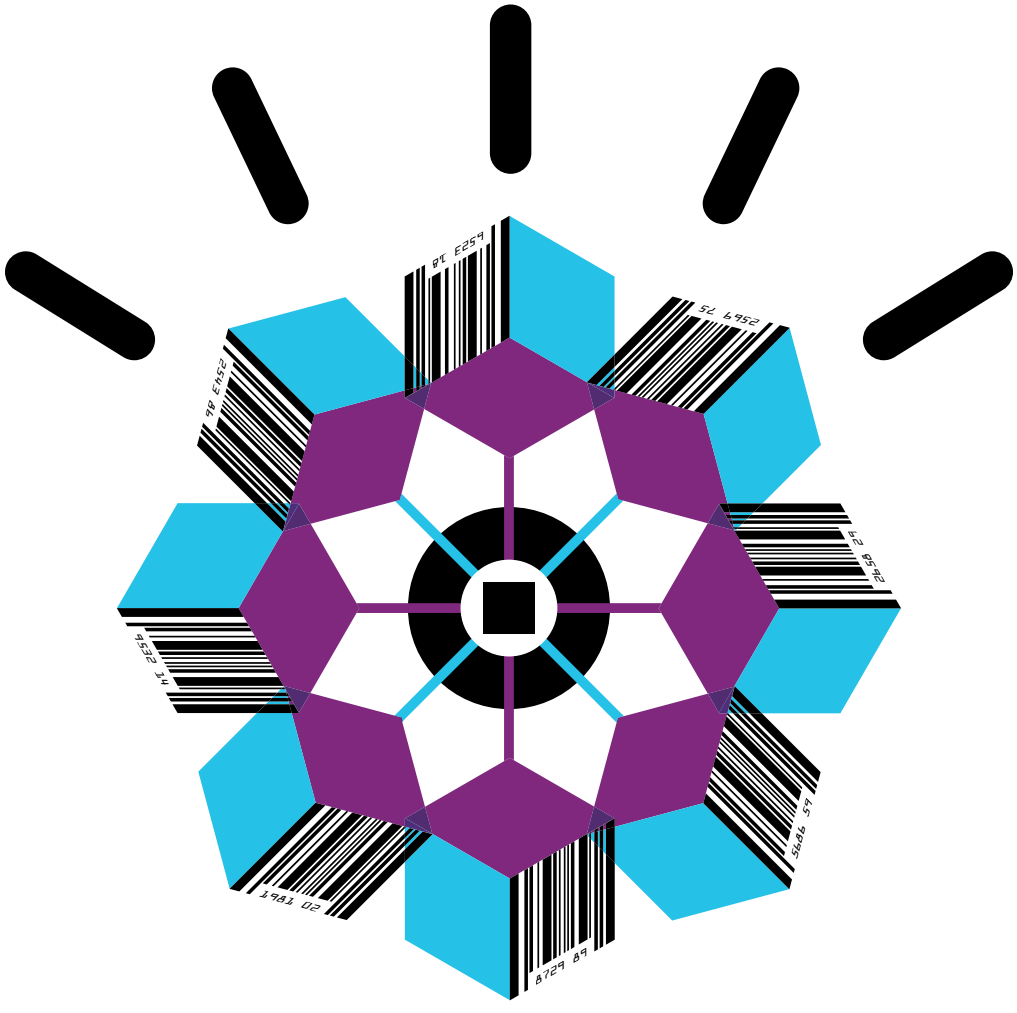
As with the Chained Processes approach, the current approach ensures that downstream processes always have a pipeline that successfully completed all preceding processes. For example, if at the time of invocation of the Deploy to QA process, the latest pipeline contained Build B3 and Quick Tests Q2, the Deploy to QA process would run within the context of the latest pipeline to successfully pass both preceding processes, the pipeline containing B1 and Q1.

The Build and Release Pipeline approach brings us the efficiency and flexibility strengths of the Chained Processes approach and the traceability advantages of the Staged CI approach. There is more complexity in this approach; however, this complexity can be safely hidden in a system or tool so that the end result is easier and more intuitive to use.

Conclusion

Given that the more tests we run, the clearer our picture of quality is and the sooner we get the feedback the more valuable it is, there are plenty of reasons to extend automation beyond the development team. With approaches I've discussed here, we've seen QA and even release management teams derive the same benefits that development teams are getting. I hope that some of the ideas in this article will allow you to obtain similar results.

Return to Table of Contents



By making things smarter, life gets better.

Fully 66% of products developed in the past year included some kind of embedded information technology. We're living in a world of smarter products, and that brings myriad benefits—not to mention a lot of interactive fun—to all of us.

But what does it mean for the creators of those products? What are the implications for *them* of infusing computing into things we would never think of as computers: phones, cars, appliances, clothes, medical devices and more?

One reality is clear: software is becoming a strategic business asset for every company. The makers of smarter products have to become just as proficient in complex systems integration and software development as they've been in conventional product design and manufacturing.

This shift is profound, and urgent. For instance, how does a company manage and integrate software across a complex, ever expanding supply chain when more and more components are sourced from different locations, arrive with software already embedded and are assembled in various combinations? Aerospace industry leaders are responding by transforming the way they manage a constantly changing set of design and manufacturing requirements.

At the same time, product development cycles are being challenged by the integration of mechanical, electronic and software technologies into the design process. This requires new levels of interoperability. BMW, for example, reshaped its engineering processes, modeling them on the development processes of leading software providers. This has dramatically reduced testing and manufacturing costs. Overall, best-in-class manufacturers that have built software systems or virtual prototyping into their business practices are meeting 90% of their product deadlines and 87% of their budget targets.

With smarter ways to make all the things we want and need, we now have the opportunity to do for product creation in the 21st century what mass production did in the 20th.

Let's build a smarter planet. Join us and see what others are doing at **ibm.com/smarterplanet**

IBM, the IBM logo, ibm.com, Smarter Planet and the planet icon are trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at www.ibm.com/legal/copytrade.shtml. © International Business Machines Corporation 2010.

Software Development In the Cloud

Cloud management and ALM

By Nick Gulrajani and Darryl Bowler

Similar to how virtualization began to take hold in the engineering lab, cloud computing is taking root in software development. The reason for this is obvious: Development teams are quick to jump onto any leading edge technology that solves their challenge of needing dynamic, flexible tools and processes. Development computing environments require more adaptability — systems specifically must accommodate shorter project sprints, be less static and more configurable-on-the-fly, and support collaborative principles — in short, on-demand resources that can be shared across teams, managed by development, and have traceability across projects.

Access to flexible, on-demand cloud computing resources, either from a virtual private cloud within the corporate data center or from public clouds, can provide just such a flexible environment. While clouds are interesting on their own (who doesn't want all of Amazon's computing resources available to them?), having tools for managing cloud resources (such as tooling and workflow) is key. 'Cloud management for development' brings teams the level of control and visibility necessary in this new landscape.

With cloud management for development, teams ultimately drive the allocation and provisioning of their systems — on-demand — as they need them. They can utilize the pooled physical and virtual machine capacity of a cloud for more flexible automation and reuse across projects. This means less time spent configuring and finding errors when software moves from one stage to the next. The benefits of thinking of servers as 'clouds' or 'pools' of virtual resources and version-controlled configurations include:

- Managing and controlling entire development, build, and test processes from one interface means capacity can be monitored and optimized.
- Storing and managing profiles (configurations) as reusable assets across a project or across multiple projects, easily accessible by any developer in the project or organization.
- Visibility from the project— and management-team levels as to how the resources are being used, with the ability to charge back resources per project if desired.

In this article, our focus is on how to manage virtual private clouds for Application Lifecycle Management (ALM). We generally define virtual private clouds as groups of public or private server pools from your corporate data center or from public clouds like Amazon EC2. To these server pools, developers would apply some software stack or configuration required for their task at hand. A typical enterprise use case for managing their cloud might be:

- To control costs, manage security, or supplement resources during peak use, engineering or project managers set up what clouds a project may have access to.
- Individual hardware resources can be assigned to a 'cloud' that now has the capacity of their combined resources. Dell class servers could be allocated to a development cloud, while HP blades with a higher service-level agreement and security could be allocated to a production cloud. The cost associated with a machine in the former is about $0.10 per hour while the latter is $0.25 per hour.
- A project manager (or admin) allocates portions of either cloud to projects as the development or production configuration. The manager can limit projects to certain clouds; for instance, services from the development cloud can be used for development purposes only.
- Finally, cost accounting data can be derived by tracking usage by project and user, so managers can optimize assets at varying stages and projects.
- Amazon EC2 can be used to extend resources temporarily (and at a very low cost).

To complete the concept of cloud management for development, let's also introduce the concept of Development Services or Build and Test Services. Development Services consist of code, build and test tools, applications, and infrastructure stacks that can be stored and managed as configurations or profiles, and applied to an available server. These profiles can be accessed and used globally and version controlled for consistency across the application development lifecycle. So Development Services are simply software configurations or profiles applied to an available server in the cloud, on demand.

Development In the Cloud: A Practical Use Case

One use case for software development in the cloud combines the agile best practice of continuous integration (CI) and some common collaborative development tooling that has been used across companies of varying sizes, locations, and industries. This

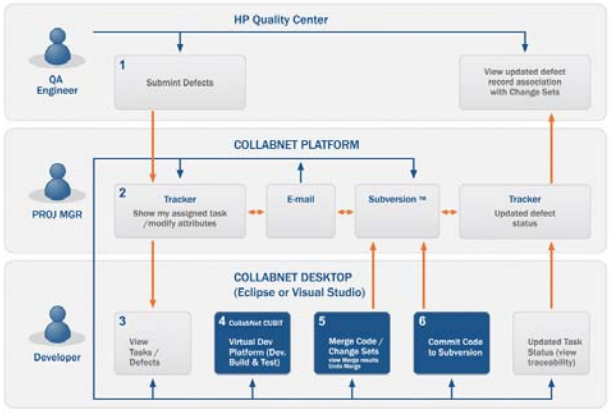


Figure 1: Software development in the cloud workflow and stakeholders.

approach is scalable and flexible around the size of your development team and type of products you are building.

The cloud development use case encompasses the flow of defects/requirements through phases of development/builds/tests and back to submission of new requirements or defects by various stakeholders. Automation at any point possible is a key capability, including the ability to 'turn on' and 'rip down' virtual or physical systems as needed, in a cloud.

Figure 1 illustrates the workflow for development in the cloud through the perspective of the various contributors, along with their collaborative and cloud management tools.

1. Business Analyst/Quality Engineer (BA/QE) submits defects/requirements.
2. Project Manager (PM) picks tasks, sets priorities, and assigns them to development.
3. Developers open their favorite IDE and view their tasks. They begin to work on the defect and write code.
4. Developers use the cloud-management platform to build and test code.
5. They merge code and change sets.
6. They commit code to a source-code management tool (here, Subversion). This triggers a continuous integration that takes place using the cloud management and build automation tool (Figure 2).
7. CI tool monitors for code changes (Figure 2).
8. Upon build failure, defect tracker is updated and notification is sent to the development team (Figure 2).
9. Upon successful build, the defect tracker is updated automatically. If the test succeeds, test results are e-mailed to PM/QE. If the test fails, QE is notified (Figure 2).

Virtual Private Clouds

So what part of this happens in the virtual private cloud?

- Step 4 is where the developer may configure his own cloud system on demand for build and test.
- Steps 7, 8, and 9—the CI tool and build automation — run on the cloud systems.
- Once a build is successful, the artifacts are uploaded automatically to the Project Build Library on the cloud systems. The Project Build Library (PBL) stores files created and used by the CI process, which can be shared with others who may need to access the build results.
- A test system can be dynamically provisioned and build artifacts are downloaded and tested automatically.

Figure 2 illustrates what the CI in your private cloud might look like, with virtual and physical machines and tools for source code, tracking, build, and test.

Setting Up a Cloud Development Environment

We now present an overview of some of the tools and principles to consider when modifying your own environment for development in the cloud. Basic principals include:

- All stakeholders should be able to see and make progress on the code and executables in real time without having to replicate data or change ownership.
- A version-control system should be integrated with build automation tools.
- Simple branching strategy (parallel development) for development, integration, and release work.
- Make incremental changes and test often.
- Focus on small releases that provide the most business value.
- Utilize aggressive milestone management, such as the agile imperative of time boxing (for more details, see the text box entitled 'Time Boxing').

Time Boxing

Time boxing involves splitting projects into separate time periods of 2-6 weeks, each with their own deadline and budget. Deadlines are fixed but deliverables are adjusted. To meet the fast turnaround times and project splitting, engineering must allocate and config-

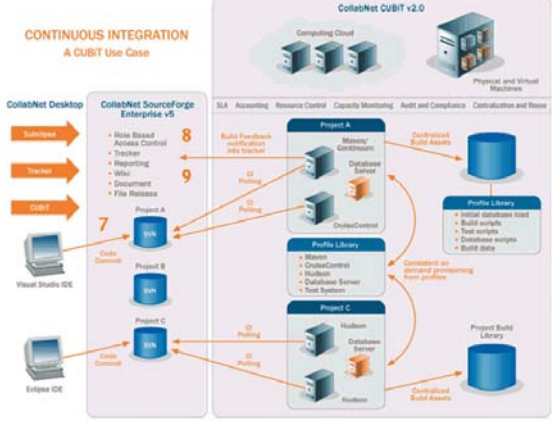


Figure 2: Continuous integration in a virtual private cloud.

Tool	Activity	Example
Project Activity Tracking/ Collaboration	An integrated suite of web-based issue tracking, project management, and collaboration tools such as wikis and document management.	CollabNet SourceForge
Automated Build Tools	Build management and acceleration tools.	CruiseControl, ANT, Hudson and Maven
Automated Test Tools	Defect submission, automated tracking, etc.	HP Quality Center, PushToTest
Source Code Management	Commit Code, Branch and Merge.	Subversion
Integrated Development Environment (IDE)	Developer desktop tool that interfaces with source code management, debug, unit testing, and other tools.	Eclipse, Visual Studio
Feedback Mechanism	Notification system for status of builds, tests, issues, etc. across the lifecycle. E-mail and SMS are good, however an integrated tracker system enables more auditability of changes, can use knowledge threading, and can associate build defects with source code and/or other artifacts.	CollabNet SourceForge Tracker, Email, SMS, wikis
Virtual Private Cloud Management	Physical and virtual machines that can be flexibly managed by software development teams. Profiles (configurations) can be version controlled and managed.	CollabNet CUBiT
Project Build Library (PBL)	Centralized area for daily builds, as CI is not always sufficient for comprehensive testing. In addition, daily or weekly build, integration, and test on a clean 'production-like' system, configured with a standard production profile. PBL helps to automate this task.	CollabNet CUBiT

Table 1: Tools for software development in the cloud.

ure systems quickly. If constrained by access to these systems or incorrect configurations, short iterations are impacted. On-demand cloud management server pools can mitigate this.

Taking advantage of software development in the cloud does not require massive retooling—that's one of the benefits. You may already be using many of the basic collaboration and change-management tools commonly used by software teams worldwide. (As a disclaimer, many of the tools we talk about are those that CollabNet sells.) Table 1 lists the tools.

In general, requirements for tools should include:

- Support for heterogeneous environments, so teams can have flexibility across the organization, based on specific project requirements.
- Adaptive to current workflows and tools. Developers and QA engineers want to stay in familiar environments.
- Access to complementary tools and configurations across the development lifecycle.

For instance, CUBiT (CollabNet's Cloud Management for Development Services) can be configured to manage a virtual private cloud with your own corporate computing resources, expanding to a public cloud such as Amazon EC2 if desired. This type of capability is key for automating any development environment. With such automation, overhead for software teams to obtain and manage computing resources is reduced with the Development Build and Test Services previously described. New systems can be provisioned from the predefined profiles within minutes while maintaining corporate security, auditability, and traceability.

Design goals for the cloud environment include:

- Teams must have autonomy over their own resources, and must have minimal delay in provisioning their own systems.
- Strict access controls so that resources are dedicated to project teams or to a dedicated purpose (such as build or test).
- Charge back for resources used. Even if the organization doesn't charge back, teams can begin to understand the resource costs associated with their projects.
- Using agile processes such as continuous integration (CI).
- Automated build and test.

Figure 3 illustrates how CUBiT manages development clouds of build and test services — combinations of physical and virtual machines with profiles.

Software developers within their own projects have access to their own dedicated systems that are dynamically sharable among project members. CUBiT management features include visibility of all system resources within cloud, auditing, capacity monitoring, role-based access, and accounting with charge back. Profile management enables version control and traceability for configuring predefined development stacks within minutes.

Step 1: Configuring a virtual private cloud. Using CUBiT, you can configure multiple clouds for different use cases, and have the ability to control the usage of those clouds. For example, we could create clouds for continuous integration build systems and assign them as an exclusive resource to that project, or share it among any number of projects.

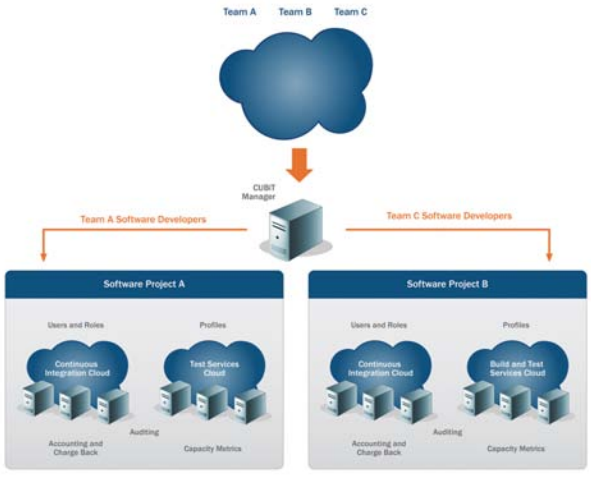


Figure 3: Cloud management with CUBIT.

Step 2: Giving developers easy access to cloud resources through their IDEs. CollabNet provides a free plug-in to IDEs such as Eclipse and Visual Studio that lets you seamlessly access the source-code management tool, collaboration platform, and development services (Subversion, SourceForge Enterprise, and CUBiT, respectively) without having to leave your IDE. From the IDE, you can browse all system resources within a cloud or CUBiT domain, physical or virtual systems. The management interface can be viewed directly from within the IDE; in addition, the systems can be securely accessed.

Step 3: Creating reusable profiles for on-demand build and test services. By creating reusable predefined configurations explained as profiles earlier, systems can be consistently provisioned on demand, hence the concept of build and test services. Profiles are defined in an XML format and are maintained in a Subversion repository under version control, so systems can be restored exactly to a previous state if needed.

Example 1 shows how a continuous integration build system can be provisioned from CUBiT. This example installs Java SDK, CruiseControl, Apache Beehive, Apache Derby, and Tomcat — a full development and build stack. Typically, such a system can be provisioned in less than 10 minutes, thus meeting the demands of an agile development team.

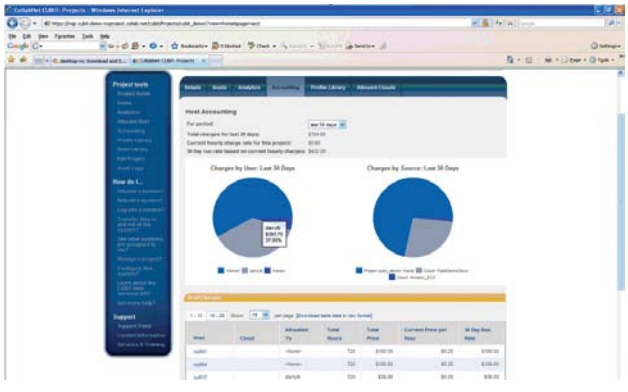


Figure 4: CUBIT report.

Example 1:A continuous integration build system.

```
<rpms action="install"path=
"pbl://mgr/pbl/cubit_demo1/pub/cruisecontrol-apache/">
<rpm>jdk-1_5_0_15-linux-i586.rpm</rpm>
 <rpm>cruisecontrol-bin-2.7.3-1.i386.rpm</rpm>
<rpm>apache-beehive-1.0.2-1.i386.rpm</rpm>
<rpm>db-derby-10.4.2.0-1.i386.rpm</rpm>
<rpm>apache-tomcat-5.5.27-1.i386.rpm</rpm>
</rpms>
```

Step 4: Using CUBiT Web Services to automate cloud provisioning for testing services. CUBiT has an extensive set of REST-based APIs that let you automate what would normally be complicated procedures. For example, you can dynamically provision build systems in the cloud, then destroy those systems once testing has completed. This is useful for testing milestone builds that require a 'clean' system in a known good state.

Step 5: Determining how cloud resources are being used and charged back. Finally, tools like CUBiT give teams and managers the ability for each system or profile to be used on an allocated/hour basis. Even if this feature is not used to charge development teams, it is still valuable for determining the usage and allocation patterns of your resources. In Figure 4, CUBiT graphically reports (in either a pie chart or table) who is using the resources in any given project and the associated costs. This ability becomes more important as organizations begin to dynamically allocate computing resources across projects that require managing their own virtual private clouds.

Conclusion

Software development in the cloud brings adaptability and flexibility to any size project team. By utilizing some common software development tools and a new one that introduces the concept of cloud management and development services, teams can build their own cloud development environment with continuous integration automation and "roundtrip" feedback to provide on-demand access and visibility across the development lifecycle.

Return to Table of Contents

Q&A: Agile Meets ALM

In an Agile world, the traditional "phases" of ALM are much shorter and much more tightly dependent upon each other

By Jonathan Erickson

Cyndi Mitchell, managing director of ThoughtWorks Studios, recently talked about Agile development and application life-cycle management with Dr. Dobb's editor in chief Jonathan Erickson.

Dr. Dobb's: Does Agile make it more difficult to achieve effective ALM?
Mitchell: No, Agile engineering practices, when applied effectively, make it easier to know what's really going on in a project or within a code base. Agile does place heavier demands on ALM tool providers. Most tools are too prescriptive to support the adaptive nature of Agile once the application life cycle gets under way. Still others cover only one aspect of the application life cycle, leaving end users to cobble together disparate tools to support the entire application life cycle.

Dr. Dobb's: Is it really possible to automate all or part of the ALM process?
Mitchell: Yes, though it may not always be desirable. Any place in the application life cycle where humans are required to repeatedly perform manual steps will create an opportunity to introduce errors and waste valuable resources, and these are all strong candidates for automation. Of course, there may be corner cases where the cost/benefit trade-off of automating a particular aspect of a particular stage in the life cycle doesn't make sense.

Dr. Dobb's: Do ALM and Web 2.0 butt heads, or are they simpatico in terms of life-cycle management?
Mitchell: They're largely simpatico, but one area where they often butt heads is around the rich user experience of many Web 2.0 apps. Many of the dynamic "Ajax-ey" toolkits are difficult to test with current automated functional testing approaches, and this can make automating some aspects of of the application life cycle very painful.

Dr. Dobb's: What's been the biggest change in ALM the last few years?
Mitchell: There is a growing recognition that the scope of ALM is far broader than just project, program, and requirements management; it must extend to development, deployment, support, and maintenance. Of course, ALM tools need to support this entire scope as well, including a holistic approach to good Agile engineering practices.

Dr. Dobb's: Are we missing any steps?
Mitchell: In an Agile environment, it's helpful to think of the ALM process not at as sequential phases or steps, but rather as a series of short work increments, each including just enough analysis, design, development, testing, and deployment to deliver a bit of business value. The right processes and best practices are continuously discovered and improved.

Dr. Dobb's: Is any one phase/category of ALM more important/critical than any other?
Mitchell: In an Agile world, the traditional "phases" of ALM are much shorter and much more tightly dependent upon each other. The application of good engineering practice to all aspects of development, testing, build, deployment and release management is highly critical as this is what makes projects and systems reliable and predictable regardless of stage in the life cycle. It is also critical that your chosen ALM tool holistically supports good engineering practice across each of these areas.

Dr. Dobb's: Does parallelism/multicore complicate ALM?
Mitchell: It doesn't have to. Parallellism/multicore brings huge benefits to the automated testing, build, deployment, and release management aspects of the application life cycle — the right ALM tool will allow companies to explicitly take advantage of this, while hiding the complexity.

Return to Table of Contents

Separating Duties to Meet IT Compliance

If your organization is facing strict audit compliance but struggles with distributed platforms, take a look at how the big boys do it

By Tracy Ragan

We used to call it "throwing it over the wall"—the process of turning over source code for production release. That was in my mainframe days when "throwing it over the wall" was a simple check-in of source code to CA Endeavor. Once we handed over our source code, the rest was handled by production control. The request for approval to promote to the preproduction stage was all that was required. Production control handled the promote, compile-and-link edit, and move-to-production. This simple process of separating production control duties between development teams and production control teams has been a standard process on the mainframe for the last two decades. But developers of today's distributed platforms—Java and Windows developers—just can't seem to figure it out.

But is the distributed platform really that complex, or are we missing something? To answer this, look at large UNIX-based legacy systems. Written in native languages such as C or even assembler, they resemble both the Java/Windows and mainframe worlds. Ask any systems administrator on the UNIX platform. They have the bragging rights to describe a process that clearly separates duties between development and production control. Developers check-in their source code to IBM ClearCase and the system administrator manages one large ClearMake makefile to build the code and move it to production. Different tools, but a similar process to the mainframe. If complex legacy UNIX application teams can do this, then so should Java and Windows application teams.

These legacy processes should not be forgotten. They have much to offer in terms of lessons learned from years of experience delivering critical software solutions to end users. They represent the most sophisticated systems ever designed. They serve as the intelligence in our Apache Helicopters and track our hard-earned cash through complex banking systems. There is nothing simple at all about the applications, or the code that these systems deliver.

Common Themes

There is a common theme in both of these "over the wall" processes—both hold true to a very clear separation of duties between development and production control. The key to the process is that developers never build the production executables, nor do they control the process or scripts that create the production executables—and they never release the production executables to production. In both of these environments (development and production control), binaries can be traced back to their source code of origin through footprint technology that captures the information at compile time. Developers are responsible for developing and unit testing the source code, then leaving the rest to teams that focus on getting the steps right.

Java and Windows development teams are now being asked to step up to the plate and hit one out to the cheap seats. Auditors and government regulations such as Sarbanes-Oxley are simply expecting from Java and Windows developers the same level of professionalism that has been demonstrated by developers who came before them. In many shops I visit, I hear CTOs and CIOs complaining that they could pass their audits if only they could put some controls around the Java and Windows development teams. They can't understand why, after spending money on versioning and sophisticated software configuration management (SCM) tools, that they still cannot provide a clear audit trail showing what source code was used to create the binaries running in production—and why they cannot restrict Java and Window developers from having access to production servers.

The answer is not too far in the past. CIOs and CTOs simply need to look at their legacy mainframe or UNIX applications to see the answer. Java and Windows developers struggle with letting go of the process of building and releasing their binaries in the same way as mainframe or UNIX developers can. Most importantly, Java and Windows developers have not sorted out a build process that can be repeated regardless of build location. Their build scripts are ad hoc, have static references, are highly redundant, and are anything but repeatable. For this reason, they "hit the wall" instead of going "over the wall."

Binary Management

The application build process is the final piece of the IT compliance puzzle for distributed teams writing Java and Windows applications. Ad hoc build scripts are the roadblock preventing Java and Windows developers from moving forward. Ad hoc build scripts

are the most common methods used to manage Java and Windows application builds, yet they don't meet the four essential requirements of IT governance standards:

- Traceability
- Auditability
- Validation
- Separation of workflow duties

To mature the Java and Windows development process, a method for managing binaries must be implemented. Managing binaries is a core component of the build-to-release process. If you cannot manage the process of creating binaries, then your build-to-release process becomes invalid. Similar to SCM, binary management lets you track, trace, and manage the details about the build. Your process of managing binaries lets you control what compile-and-link flags were used to build the deployable objects, restrict the use of debug flags used in production builds, and control the location and version of the compiler and linker used in the build. A wobble in any of these areas can cause drastic differences in build results.

There are four core competencies to a solid binary management process:

- Reduce your dependency on ad hoc build scripts. Both mainframe and legacy UNIX developers had to face this core requirement. At one time, mainframe developers managed their own compile JCL. They would release applications into production by running their compile JCL and creating load libraries right into production. Legacy UNIX developers wrote hundreds of make scripts until they realized that a single well-managed makefile written in a commercial make language such as IBM ClearMake offered them a method of standardizing the creation of binaries across diverse teams. Java developers are overly dependent on open-source languages to perform this critical step. Windows developers often rely on point-and-click processes inside Visual Studio to create the binaries. Neither method allows for a repeatable process that can be turned over to a production control team. As long as you rely on ad hoc build scripts, you will never meet the separation-of-duties requirement.
- Never take binaries from a developer or execute a developer's build scripts. The only reason production control would rely on developers to create binaries or to provide scripts to create the binaries is because there is no repeatable method between development and production. Therefore, all of production control must be done in the development environment. The requirement to never take binaries from a developer is clear, but understanding why production control should not execute the developer scripts is less obvious.

  If your production control team is simply executing an automated process to run a build script provided by the developer, the separation-of-duties requirement cannot be met. First, there is no clear understanding where the source code is coming from. Scripts point to directories that are often outside the local build directory. Second, the developer's build-machine configuration may be at higher versions than the production runtime environment. And finally, your production-control team needs the ability to validate if production-level build configurations are turned on/off, such as removing debug flags and adding optimization flags. Yes, it is these steps that mainframe and legacy UNIX applications have understood for years, but Java and Windows teams are completely forgetting.
- Identify the files used in the build that were not managed in the SCM repository. If you have gone through the expense of implementing source-code management tools throughout your development process, then it is critical to validate that the managed source code is actually used in the build. It is often critical to identify the components used in the build that are not under SCM control. These components are often critical third-party libraries, database libraries, SOA WSDL objects, and Java runtime libraries. Your process of managing binaries cannot underestimate the importance of these components. They serve to be as critical as any homegrown source module.
- Create a footprint showing which source code and libraries were used to create the final executables.

Managing Dependencies

Managing dependencies is the most critical requirement of binary management. If your build-to-release process does not manage dependencies, then you won't be able to clearly see what components were used to create the executables. Yes, you can generate traditional bill-of-material reports that show the files checked out of your SCM tool, but this does not expose what files were found outside of the local build directory. Dependency management and orchestration provides a complete audit trail showing what source code and versions were used to create the final deployable objects. Nothing else can do this. Dependencies can be difficult to trace and often impossible to understand with manual scripts. That's why you need to implement a build-management solution that offers binary-management services to ensure that when the build executes, a dependency-scanning tool watches exactly what is called and used by the compilers and linkers. This is familiar to the mainframe of legacy UNIX teams. Figure 1 is an example footprint that exposes not only the files managed within IBM ClearCase, but also the files not under ClearCase control; notice rt.jar is listed as "Not in VOB." This footprint also exposes all of the environment variables used during the build. This level of footprint goes far beyond what a simple bill-of-material report can show.

If you're a Windows or Java developer and think that the use of ad hoc scripts is the only way to complete the job, well, I understand. I thought the same thing years ago when I was told I could not use my own compile JCL to release my mainframe changes to production. But there are real benefits if you implement a build system that actually managed your binaries:

- Developers using continuous integration builds could benefit from performing truly iterative (incremental) builds. If you have a need for speed, the easiest way to get there is to stop building objects that are already up-to-date. The best way to accomplish a continuous integration build process is to build

Build Audit Report for metalworks.jar

```
Project Variables:
    Built on SONYLAN by steve at 04/18/2006 08:47:59
Environment Variables:
    APPL=METALWORKS
    CFG=RELEASE
    COMPUTERNAME=SONYLAN
    JAVA_HOME=D:\Program Files\Java\jdk1.5.0_06
    OPENMAKE_SERVER=http://sonylan:58080/openmake
    OS=Windows_NT
    Os2LibPath=D:\WINNT\system32\os2\dll;
    PERLLIB=D:\Program Files\openmake641 client\perl\lib
    PROJECTVPATH=.;
    Path=D:\PROGRA~1\Java\JRE15~1.0_0\bin;D:\Program Files\openmake641 client\bin;D:\Program Files\openmake641 client\perl\bin;D:\WINNT\
    ProgramFiles=D:\Program Files
    Pwd=D:\ClearCaseViewsDev1\dev1_dev\metalworks_sources\InitialComponent\Metalworks
    REFDIR=D:\Program Files\openmake641 client\examples\ref
    STAGE=QA
    USERDOMAIN=SONYDOMAIN
    USERNAME=Steve
    VPATH=.;$(REFDIR)/metalworks/qa/src;$(REFDIR)/metalworks/qa;$(REFDIR)/metalworks/release/src;$(REFDIR)/metalworks/release;$(JAVA_HOM
Dependencies:
 Version                                                    Date       Time     Size      Target Dependencies
                                                            04/18/2006 08:48:12 9         metalworks.javac
                                                            04/18/2006 08:48:01 52        metalworks.classpath
 rt.jar is not in a VOB                                     11/10/2005 14:19:19 37757974  D:\Program Files\Java\jdk1.5.0_0
 AquaMetalTheme.java@@\main\dev1_dev\2                      03/13/2006 17:33:40 2428      D:\ClearCaseViewsDev1\dev1_dev\m
 BigContrastMetalTheme.java@@\main\dev1_dev\2               03/13/2006 17:33:46 4220      D:\ClearCaseViewsDev1\dev1_dev\m
 ContrastMetalTheme.java@@\main\dev1_dev\2                  03/13/2006 17:33:49 4642      D:\ClearCaseViewsDev1\dev1_dev\m
 DemoMetalTheme.java@@\main\dev1_dev\2                      03/13/2006 17:33:51 3628      D:\ClearCaseViewsDev1\dev1_dev\m
 GreenMetalTheme.java@@\main\dev1_dev\2                     03/13/2006 17:33:53 2399      D:\ClearCaseViewsDev1\dev1_dev\m
 KhakiMetalTheme.java@@\main\dev1_dev\2                     03/13/2006 17:33:55 2837      D:\ClearCaseViewsDev1\dev1_dev\m
 MetalThemeMenu.java@@\main\dev1_dev\2                      03/13/2006 17:33:57 2963      D:\ClearCaseViewsDev1\dev1_dev\m
 Metalworks.java@@\main\dev1_dev\2                          03/13/2006 17:33:58 2975      D:\ClearCaseViewsDev1\dev1_dev\m
 MetalworksDocumentFrame.java@@\main\dev1_dev\2             03/13/2006 17:34:00 6129      D:\ClearCaseViewsDev1\dev1_dev\m
 MetalworksFrame.java@@\main\dev1_dev\2                     03/13/2006 17:34:02 9616      D:\ClearCaseViewsDev1\dev1_dev\m
 MetalworksHelp.java@@\main\dev1_dev\2                      03/13/2006 17:34:03 4909      D:\ClearCaseViewsDev1\dev1_dev\
```

Figure 1: Example footprint that exposes files not managed within IBM ClearCase.

only the changes each time a build is launched. The concept of continuous integration is not new. Just go ask one of the mainframe old timers. They've been building in an iterative, continuous method for years. They do it best—they check-out, make a change, and check-in. The check-in launches a build that only builds what has changed. They never need to perform a "clean all" build (and would laugh at the thought). If they did, some large systems could take days to compile. This point alone should pique your interest.

- Simplify changes outside your IDE as easy as inside your IDE. I see Java developers struggling with code and package refactoring outside the IDE all the time. It's incongruent that developers demand their IDEs handle processes such as code refactoring inside the IDE by pushing a button, but when building outside the IDE they are required to manually revisit dozens of Ant scripts to reflect the refactoring changes performed automatically by the IDE. Isn't it time for your build system to be dependent upon the IDE project file, preventing the manual update of Ant scripts anytime refactoring is required? There are better ways.
- Binary footprints expose the problem areas in a fast and efficient way. Distributed developers (Java or Windows) could benefit from what the legacy UNIX and mainframe teams have used as a critical tool for years—the ability to look inside the binary by running an "identification" program that shows the precise artifacts used to create the binary. There is no faster way to identify which source code or library broke the build or caused a runtime failure.

Conclusion

If your organization is attempting to meet strict audit compliance but struggles with meeting the separation-of-duties requirement when it comes to the distributed platforms, consider taking a look at how the big boys do it. Legacy UNIX and mainframe developers went through the same growing pains that distributed platforms are currently experiencing. Meeting that separation-of-duties requirement is as simple as implementing a build-to-release management system that supports binary management, and letting someone other than a few core developers build the application on only one or two development machines.

The benefits of addressing the binary management step of the build-to-release process will pay off in both time and money as well as the maturation of the Java and Windows development process. And don't be concerned that meeting this requirement somehow interrupts your lean development techniques—it can actually improve them. Both production control and developers have lots to gain by solving this critical component of the development-to-release process.

Return to Table of Contents

Application Lifecycle Management Meets Model-Driven Development

The combination of ALM and MDD gives you the connected workflow you need to handle the development of even the most complex applications and systems

By John Carrillo and Scott McKorkle

Application lifecycle management (ALM) has evolved into an ecosystem of integrated processes and domain technologies for the system and software development lifecycle. ALM establishes a framework that you can use to:

- Catalog and manage customer requirements.
- Plan a portfolio of development projects to address these requirements.
- Manage design, development, testing, and deployment.
- Manage change throughout the entire process.

MDD, on the other hand, lets you more accurately design, simulate, and validate the complex behavior of distributed, mission-critical systems. The model-driven process uses visual aids to accurately describe and define system objectives and solutions. Scientific and technical industries, such as aerospace, defense, and telecommunications, depend on MDD to increase the quality and efficiency of complex software and systems through modeling.

The combination of ALM and MDD creates a rich environment of connected processes and interacting solutions that are proving invaluable for successful systems and software development projects. This is a welcome advance, given the state of today's complex development environments. A major challenge for many organizations is finding a way of integrating all aspects of the development lifecycle in an intuitive, yet formal manner to deliver long lasting, business critical products, systems, and applications.

Moreover, in combination ALM and MDD provide exponential gains in the optimization of development lifecycle processes. MDD is a natural fit within the ALM framework. The integration of ALM and MDD lets you optimize products and services and serves as a framework for the fast, accurate, and coordinated design and development of architectures, applications, and products. The combined framework connects to an enterprise architecture at the highest level, helping organizations adopt more iterative development approaches. It also provides integrated products and best practices, as well as change management, so organizations can respond with greater agility as the pace of business changes.

Application Lifecycle Management

The concept of application lifecycle management (ALM) was introduced to help companies identify tools and technology to ensure software project success. Figure 1 shows the key components of the ALM ecosystem that focus on visibility, quality, and alignment in the development lifecycle.

Essentially, a defect, bug, story, feature, idea, or suggestion is a requirement if it results in a code change. Within the ALM ecosystem, requirements are the foundation from which all things evolve—they serve as the thread that ties together all phases of the product lifecycle. To maximize return on investment in the ALM environment, it is essential to align the organization's development efforts with internal and external customer requirements.

Because requirements are the key driver to align the business with engineering within software development lifecycle, a requirement-driven approach to ALM is fundamental. Whether you are developing a product, system, or an application, a requirements-driven approach lets you ensure that the goals of the business and your customers' needs are prioritized into requirements that remain persistent and auditable throughout every phase of the development process.

Another big piece of ALM is change management. Change requests, inevitable in any organization, can come from many sources and can take various forms; for example, bug fixes, customer proposals, enhancement requests, action items, platform changes, defects, and help-desk tickets. In addition, controlling changes to products, establishing relationships between them, managing different versions, assessing the impact of changes, and auditing and reporting the changes can overwhelm any organiza-

Figure 1: The fundamentals of the ALM ecosystem.

tion. Producing or reproducing a product with controlled incremental changes, comparing different releases, and analyzing the differences are all activities essential to success.

Small teams may be able to manage changes in an ad-hoc manner. But, as the team grows and its development becomes more complex, lack of effective change and configuration management across all phases of the development lifecycle will cause changes to spiral out of control and jeopardize the success of the project.

A more robust approach to change management in ALM integrates software configuration management. A change-oriented approach for configuration management ensures the consistency of the development team's deliverables and workspaces through a workflow driven by logical changes (such as fixes and enhancements) that often span multiple files. It accelerates development by identifying and resolving integration problems. When integrated with change management, software configuration management lets change requests drive development tasks.

Configuration management, a discipline that is essential for success, supports parallel development, distributed development, factoring, and versioning. It encourages re-use through a single repository that contains a single version of the truth that reflects what is actually happening in the development process.

An integrated requirements-driven process for ALM makes is easy to prove requirements are traced to their implementation requests, which in turn are linked to development tasks and objects. This ensures that developers are focused on the right priorities, have the latest updated information, and also have access to the full context of their assignments. Analysts, business managers, and auditors have full real-time visibility into the implementation.

By connecting the development process, the right tool, and the people, a requirements-driven approach unifies traditionally separate processes of the development lifecycle into a single connected process that drives the entire development lifecycle. Unifying requirements across the entire ALM ecosystem enables information exchange and collaborative communications. It also accelerates response to changes in customer requirements, process, organization, and technology.

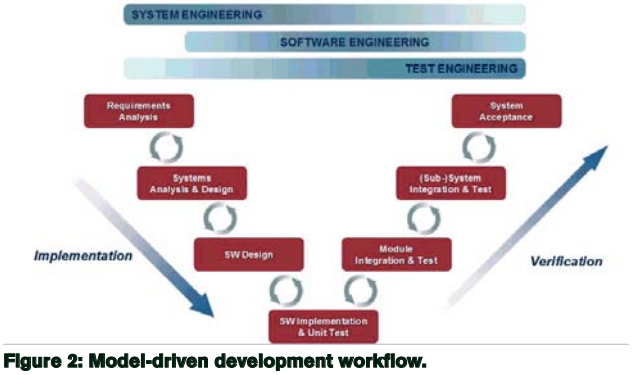


Figure 2: Model-driven development workflow.

When it comes to implementation and delivery of requirements, you can take either a hand-coding or a model-based approach to ALM. However, if you decide to take the model-driven development path, you have access to all the benefits that MDD provides.

Model-Driven Development

MDD lets you manage complexity by providing a way to make big picture decisions first, only adding details as needed. This solves the typical development conundrum of relying on implementation experts to create the architecture because they are the only ones who know how the pieces fit together. By letting architects focus on intended high-level functionality, MDD bridges the sizeable gap between the project requirements and the final implementation, and helps you realize a more manageable and practical application lifecycle platform (see Figure 2).

MDD provides a graphical language to define and develop systems and software. Based on industry-standards—for example, the Unified Modeling Language (UML 2.1) or Systems Modeling Language (SysML 1.0)—MDD lets requirements be assembled into a representative model that includes a complete use case and depiction of the application's functionality. Serving as a computer-based prototype, this model can then be "executed" in a manner similar to the way source code is compiled and run in order to simulate the system and validate and verify its completeness and intended behavior.

The results from the simulation are analyzed and used as the basis for developing more detailed specifications and requirements, iteratively extending the requirements into a complete application design. During each stage, the updated systems model can be executed to provide additional validation, verification, and detailed development. This iterative approach lets the application be constructed from high-level to detailed deployment, with continual validation of proper functionality and behavior at each step.

Once the application's systems architecture is fully specified, the model becomes the basis for all future activities, including application software development, testing, and implementation/deployment. The model can be used to specify and even generate application software, unit and integration tests, and published documentation. Because MDD is fully compatible with software development environments like Eclipse and Visual Studio, developers can debug and edit the application at any level—model or code—with their changes automatically synchronized throughout. For example, edits to code will automatically update the model and then be used to determine their impact on the entire architecture.

MDD provides a number of additional benefits. It lets you visualize system requirements and trace these requirements to design, code and test cases, formally extending requirements engineering through the entire development lifecycle. Because the requirements form portions of the model, they become an integral part of

the design itself. Design validation allows you to catch defects early in the development process, as well as highlight flaws of logic in the project requirements. Common design models eliminate the hand-off gaps between systems, software, and test teams. And, with the option of automated code generation from the design model, you improve developer productivity and quality while providing more time for design optimization and innovation.

With MDD, traceability is established throughout development. Each feature can be traced back through the model to its originating requirement, while extraneous features (those "thrown in" by well-intentioned developers) are quickly exposed, eliminating the expense and bloat of unintended feature-creep. You can also simulate and validate system behavior, which adds a whole new dimension to constructing complex applications like those based on service-oriented architectures (SOA). MDD improves development productivity, product quality, and the organization's overall competitiveness while reducing development cost and time to market.

Putting It All Together

As Figure 3 shows, the development lifecycle is best met by the combined benefits of ALM and MDD. ALM provides the discipline needed for governing and managing the project, while MDD provides the development muscle to handle even the most complex jobs.

The coordinated environment provides a consistent vehicle to handle change requests, bug fixes, auditing, and compliance mandates, all based on a single set of requirements. Project planners, requirements engineers, architects and developers all contribute to the common workflow, where even last-minute changes can be quickly inserted and evaluated for impact and relevance.

The combined ALM/MDD approach is very different from the usual methods of manual coding based on written specifications. It removes the collaboration gap between architects and developers by tying the high-level design directly to the final source code. Traceability is established from requirements to architecture to final code, while changes at any level can be evaluated at each level for impact and effectiveness. Developers can apply their expertise to ensure that optimal applications are designed, while architects and planners can ensure that the original intent is not "optimized out" in the process.

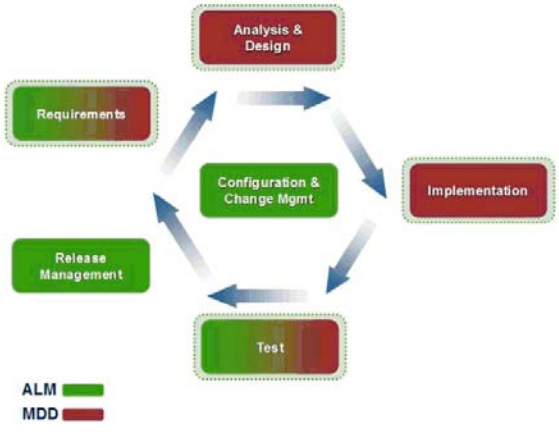


Figure 3: Connecting ALM and MDD.

Because these applications have been developed to a single set of coordinated requirements and subjected to rigorous change management procedures, they can easily be migrated to future applications throughout the enterprise. In addition, existing legacy applications can be reverse engineered and inspected for inclusion in the enterprise framework. Since they can be more easily adapted and incorporated into new applications, the organization's intellectual property is protected and applied in new ways as it moves into a services-based IT infrastructure.

Also, the combined framework includes configuration management capabilities that allow you to keep previous design configurations in an easily accessible repository—a handy feature if you need to roll back to an earlier configuration.

Accelerating ALM Tool Adoption

To reduce the risks associated with ALM and MDD tool adoption within your development environment, you must closely link people, best practices, processes, and tools. A recent report indicates that improving technology and process together is 10 times more effective at increasing productivity than improving technology alone. The reason is very clear: Process improvement (CMMI, for instance) enhances effectiveness, while development tools and best practices increase efficiency. Together, they can improve productivity and quality.

Reusing proven best practices is critical to ensuring that your teams choose to repeat a "best practice," not just any practice. Therefore, you should choose libraries that are built on years of experience with real projects. Also, it should be easy to tailor these libraries of best practices to meet specific organizational and project needs. For example, customers using Telelogic products are able to easily tailor their libraries of best practices content for ALM and MDD using the Eclipse Process Framework (EPF) composer from the Eclipse foundation.

With integrated process guidance you will promote best practices, processes, and tools across your organization and accelerate adoption of ALM and MDD. The result is systems and software that better meet your customers' needs.

Conclusion

The combination of ALM and MDD gives you the connected workflow you need to handle the development of even the most complex applications and systems. Particularly helpful is the capability to deal with changes in requirements, no matter when and where they occur during the development process. You can quickly analyze the impact of these changes to control costs and assure correctness. You can also balance business and technical requirements by performing model-based performance/security trade-off

analysis before deployment—after deployment, the cost of miscalculations escalates dramatically.

Since you can simulate designs at any time during the process, there is no need to wait until the final stages of development to determine how changes will affect the application. By providing a common baseline, this integrated environment lets you propagate changes through multiple releases. Roundtrip traceability provides both bottom-up and top-down reporting to ensure compliance. Because the entire team is working from a known set of requirements, the integration of ALM and MDD promotes collaboration, even among teams that are widely separated functionally and geographically.

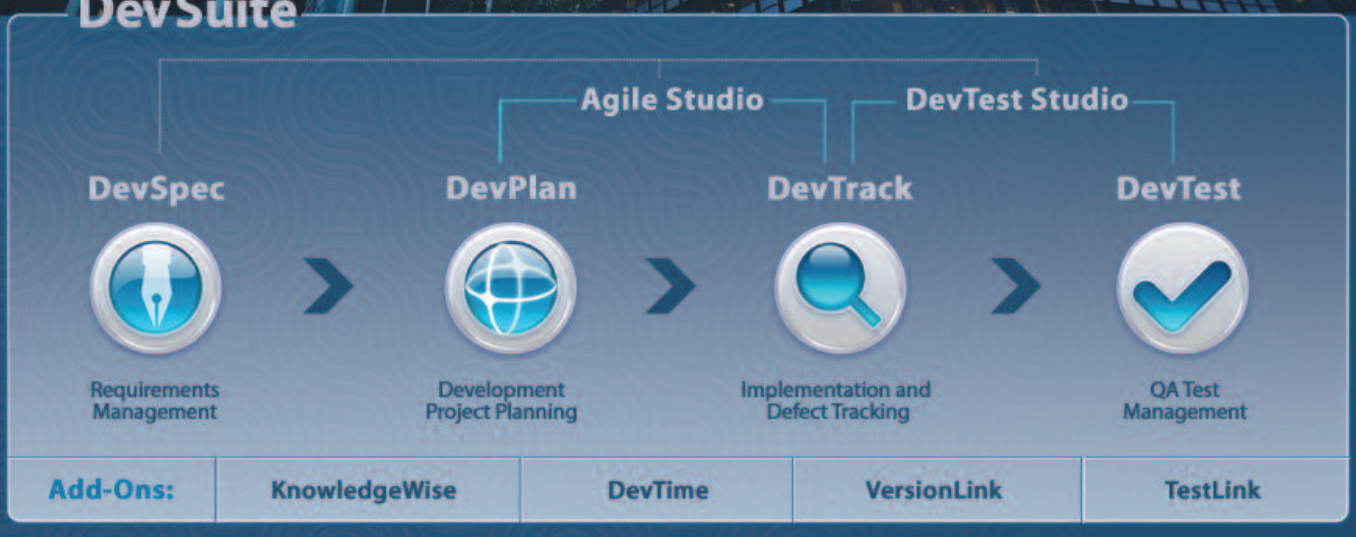
Independently, application lifecycle management and model-driven development have proven their value in a wide range of industries. No matter what industry you're working in, by combining these two systems and software development best practices, you significantly increase your ability to overcome the many challenges associated with the system and software development lifecycle.

Return to Table of Contents

Try DevSuite Live Today
Ask about our FREE trial programs
Call 1.800.439.7782 or visit www.techexcel.com

TechExcel

Knowledge-Centric ALM for Today's Enterprises



With DevSuite, you can deploy one module, a multi-module solution or the complete fully integrated ALM solution

1.800.439.7782

www.techexcel.com

Integrating ALM: Lessons Learned Deploying Tasktop at Nokia

Tool environments in technology companies require a stronger emphasis on treating engineers as customers

By Andy Boyle

This article is about lessons my colleagues and I have learned in deploying software development and ALM tools, and how those lessons are being applied in deploying a task-focused interface and backlog integration to Nokia engineers. As you'll see, we learned that tool environments in technology companies requires a much stronger emphasis on treating engineers as customers than what might be the case with traditional ALM deployments. This article is not intended to be a comprehensive guide to deploying software in technology companies, but rather it is a collection of tips that we believe would benefit others undertaking similar deployments.

The key driver for the Tasktop Deployment Project at Nokia is engineer productivity, with the specific objective of leveraging the opportunity of providing engineers with the (Carbide Eclipse-based IDE and Tasktop task-focus interface. Tasktop is the enterprise product built on the open-source Eclipse Mylyn framework. The goal of the Tasktop Deployment Project is to make it easier for engineers to switch between programming tasks (from stories on their Scrum product backlog) by remembering which artifacts an engineer works with in relation to a task, and by automatically filtering the information visible in the IDE to show only the code relevant to that task. For an engineer returning to a task, the task-focused interface answers the question: "Now where was I?"

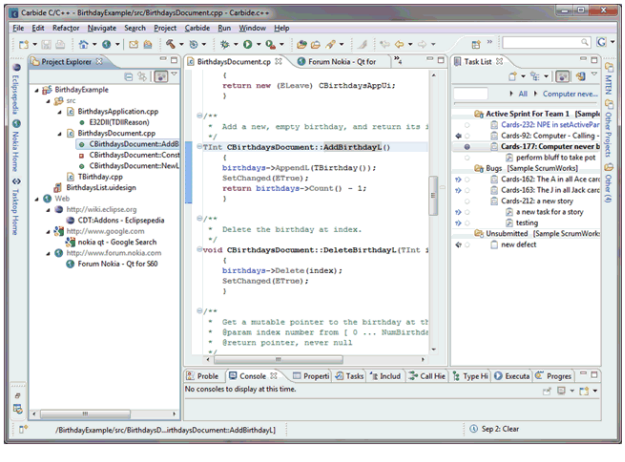


Figure 1: Focused Carbide IDE view on the left, backlog integration with active task on the right.

In terms of measuring the benefit that this technology brings, consider how much time a programmer spends searching within an IDE for programming artefacts versus actually editing them. Past studies of the task-focused interface have shown a statistically significant improvement of this ratio.

The Tasktop Deployment Project

The project to provide Carbide with Tasktop/Mylyn capability has two components:

- A development component undertaken by Tasktop Technologies which involves integrating the Carbide IDE and Tasktop.
- Creating Tasktop connectors to Danube's ScrumWorks Pro Agile change-management system and the SCM system used by Nokia engineers.

Tasktop's function here is to integrate the various development and ALM tools with the IDE, and to layer the productivity-enhancing task-focused interface on top of that integration.

The deployment process will take the product of Tasktop Technologies development component and ensure that it maximizes adoption of the software and its long term adoption within Nokia. We decided to use the Agile Scrum process to manage both development and deployment, since this allows us to quickly respond to the lessons we learn along the way. The deployment is managed in two week "sprints." Figure 2 outlines the deployment plan. At this writing we are in the midst of sprint 5 of 12 development sprints.

The risks and opportunities faced by a project is dependent upon the environment within which a tool or system is being deployed. Consider the deployment of systems that support a key development process. The deployment will be endorsed by management, users will have to follow the process, and there will be

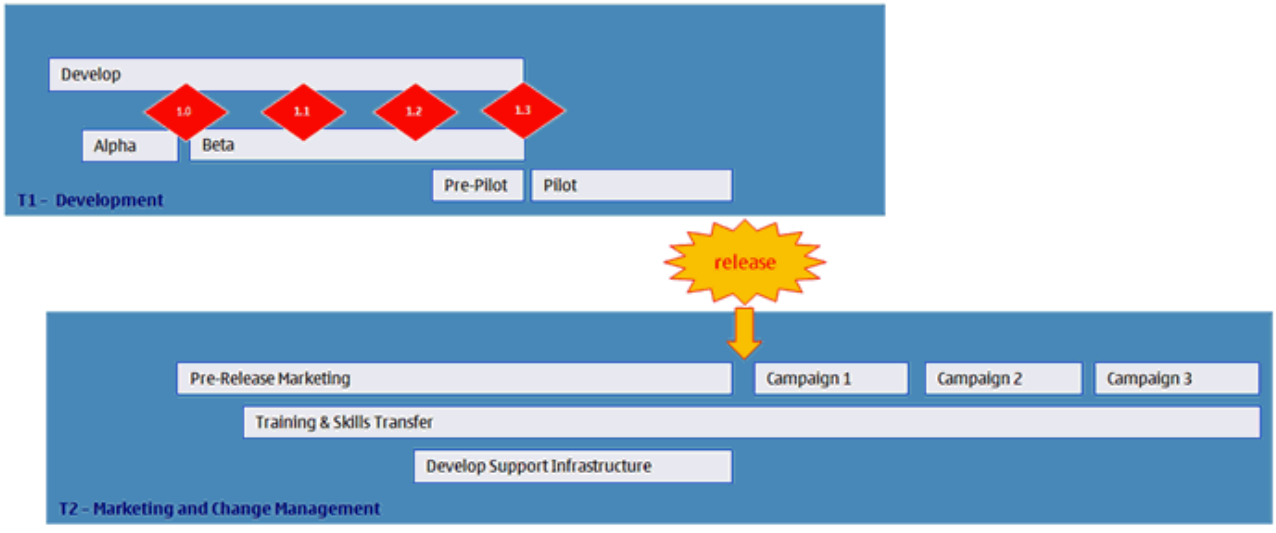


Figure 2: Deployment plan.

only one tool or system that supports the new process: Users are compelled to use the system. At the other end of the spectrum are tool deployments, where the continued running of a production process is not dependent upon the tool. In this case, there are often alternative incumbent tools or other open source options. Also, the environment will often allow users a degree of latitude in the tools they use. In this environment you are "selling in a free market"and your success will depend upon how well you focus on the needs of the end user. This is the sort of "change" environment where the tactics discussed in the rest of this article come in handy

Lesson 1: Incremental Deployment

While not always possible, incremental deployments are beneficial because they:

- Limit the scope of effect of the Day 1 gremlins
- Learn from early missteps of and adapt.
- Improve orientation and training assets based in input from early adopters.
- Sharpen marketing approach via early success stories.
- Grow the user base through early adopters without annoying the often vocal skeptics.
- Those early adopters will pass on their enthusiasm and you will benefit from a "viral" spread in usage.
- The feedback of the early adopters helps create the marketing collateral and tool configuration needed to ease adoption by the skeptics.

Lesson 2: Incremental Development and Early User Engagement

Incremental development engages key users early in the development phase. Our development is organized as a series of two-week sprints (driven by a shared ScrumWorks backlog), which deliver functionality to a group of "Beta consumers" who try out the software and provide feedback in the form of defects and new requirements.

The benefits of this approach are:

- We involve "real" users early in the development process who "own" a piece of the solution and can act as advocates of the solution amongst their peers.
- We benefit from a long period of feedback from users with "early adopter" mindset, which can polish the main deployment.
- We can react to user feedback and a changing deployment environment — adding, removing re-prioritizing stories.

The benefits of early engagement with key users is so compelling that I would look for a proven agile capability in any supplier undertaking development work on a project I am running. We were fortunate that Tasktop Technologies' background in the Eclipse development process, and their internal adoption of Scrum methods, made it easy for us to work with them in an agile fashion.

Lesson 3: Instrumentation

A key feature being built into the version of Tasktop being deployed at Nokia is the ability to track usage, not only of the tool itself, but also of the different features of the tool. The information (which is collected anonymously and cannot be traced back to the end user) helps us paint a picture of how users are using Tasktop, Carbide, and other Eclipse-based tools. This helps:

- Track where you are relative to your deployment targets ("Is team x really using this tool?").

- Tell you where you may not have been effective in communicating the value of a feature to users ("Why is nobody using feature when we think it's so great?").
- Put together models of productivity that you can use to secure buy-in from non-user stakeholders such as project and line managers.

In my experience, it is really helpful if the tool itself can provide usage stats — to the extent that I now view this as a requirement for any tool I am going to deploy.

This is one of those situations where an incremental approach with early user engagement pays off

Lesson 4: Put Yourself In Their Shoes

Our guiding philosophy is that our users are customers, their time is important, and we need their first experience of using the software to be positive.

For these reasons, an understanding of what it is that users find confusing in their initial interaction with the "whole product" is valuable and needs to be factored into development plans or your training materials. I say "whole product" here because it's as much about installation and training material as it is about the actual software itself.

This is one of those situations where an incremental approach with early user engagement — where you have a chance to adapt installation or orientation materials — pays off. If you pay close attention to the sorts of things that cause people grief during development, you can deal with them by creating either a development story (which changes the UI in some way) or a training story (for which you develop or enhance your training materials).

It's worth noting that something which is patently obvious to you, may not be to someone else; and if you are deploying to 1000+ users, there may be a significant number of them who will respond differently to the "product" than you would . And they may be a lot less forgiving and a lot more vocal. The moral here is to consider carefully every case of new user confusion, as it may be a warning of things to come.

Lesson 5: Have a Clear View of the Users' Learning Path

Software that supports complex tasks will probably require an investment in time for users to become adept. A well thought out UI and training material will allow users to perform basic tasks quickly and gain immediate benefit from their first steps onto the learning curve.

With that in mind, one useful approach to organizing and presenting your training assets is to structure them into a series of levels from new user (white belt) through expert (black belt). In doing this, you need to:

- Identify those pieces of knowledge that users must pick up in order to become effective with the tool.
- Within that set of essentials, identify those concepts and skills which the user must pick up in the first 20 minutes of using the tool.
- Identify those less important pieces of knowledge that a user can pick up at their leisure.
- Avoid mixing the two.
- Provide a learning plan that takes the user from novice to expert.

Again, the driving force behind this is respecting the value of the user's time — if you make it clear what they need to do to become effective and are careful not to mix basics with advanced usage, then users can structure their learning around their work schedule.

Lesson 6: Digestible Chunks of Wisdom

Make any training screencasts short — even if they're well structured, their perceived size can be off-putting (the user sees a 40-minute price tag on the screen and decides to set aside their learning for another day that never comes). The deployment of the collaboration suite at Nokia (including IM tool and online meeting tools) took the approach of breaking everything down into 5-minute lessons. Regular emails would appear in your mailbox inviting you to view the next lesson in the series. The beauty of this approach is that it gives the user great flexibility in when they pick up new skills — again, it respects their time. Small chunks of knowledge presented within a well structured learning plan seems to be the most effective way of enabling users to progress from "white belt" to "black belt."

Conclusion

In deploying ALM, the first and most important part of the project has to be respect for the end user. In deploying Tasktop at Nokia, I've learned that incremental deployment and development and using agile methodology to incorporate the user's perceptions and suggestions into the finished product, as well as providing straightforward and easy-to-digest training resources, will smooth any integration process.

Return to Table of Contents