JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2007**

Vol. 6, No. 1, January-February 2007

CM – ~~Configuration~~ Change Management

John D. McGregor, Clemson University and Luminary Software LLC, U.S.A.

Abstract

Configuration management, the traditional CM, has been subsumed by a new CM, change management. The strategic organization looks ahead and this includes planning for change. This is an area of particular interest to software product line organizations. A product line organization must manage multiple options and control implementations of those options. In this issue of Strategic Software Engineering I will highlight some of the issues in old CM that are resolved by the new CM and describe some techniques for addressing the issues.

1 INTRODUCTION

Recently I have had several conversations about configuration management with colleagues who have a variety of responsibilities in the development of software-intensive products. Their perspectives range from developers to business unit managers. Most jump into a discussion of problems they are having, or their people are having, with specific portions of their change process. After several meandering discussions, it is apparent that (1) many people see only the "version control" aspect of configuration management, (2) even those who understand the need for the broader context of configuration management do not have a clear definition of "configuration," and (3) many of the problems have more to do with a strategic view of change and less to do with the mechanics of controlling specific configurations.

A configuration is the set of artifacts that comprise a unit of interest such as a product or an asset [McGregor 03]. I realize this is a very abstract definition but it must cover the different types of configurations that a product line organization manages. The canonical example is the product deployment configuration that includes all of the elements shipped with a product including executables, language files, images, manuals, licenses, and installers. Others include the development configurations for assets or products that includes tools, source, test cases, and models. Even design patterns result in a type of configuration.

A configuration can be thought of as defining a coupling that binds together artifacts that serve some common purpose. That means changes to one of the artifacts in the configuration is likely to trigger changes to other artifacts. From the examples you can see that this is a multi-dimension problem. A software module is bound into configurations for each product in which it is composed. That same module belongs to a development configuration that associates test cases and documentation used when the module is to be modified.

From a tactical perspective, configuration management must provide tools that manage the individual modifications to assets and products and a procedural environment in which distributed, concurrent work is facilitated. (I will refer to configuration management as CM_t in the rest of this article.) An active development project faces many issues related to how content producers will deliver their own work into the production system and how they will be able to request/execute changes to the content of others. The CM_t tool set must include policies and processes, such as defining what constitutes a baseline or how and when to merge their work into the baseline, that guide personnel through their day-to-day actions.

From a strategic perspective, change management is needed to guide the long term health of the organization's assets and products. (I will refer to change management as CM_s in the rest of this article.) An organization's ability to respond quickly to product opportunities depends at least in part on its ability to manage an inventory of assets and to rapidly configure and produce products from that inventory. In fact, I will go so far as to say that many of the problems faced by organizations involving CM_t can be resolved by doing a better job at CM_s.

In a software product line organization a variety of artifacts are managed. Production plans, processes, and software architectures are just a few examples of reusable assets. Each of these will evolve over time. They will evolve at different rates and be introduced at different times as independent team execute their processes. These artifacts must be versioned like any code artifact but there are additional issues specific to the product line environment. For example, how is an artifact tracked through the management system when it moves from being a product-specific asset to a core (product line-wide) asset?

I am going to assume that readers understand the basics of configuration management and only provide a brief overview as part of discussing change management as an introduction to the issues related to software product lines. Then I will focus on product line-related issues.

2 BASIC CHANGE MANAGEMENT

Version control manages the change in artifacts over time. The basic management goal is to support concurrent, distributed work while preventing the destruction of existing work by accident or by whim. Version control systems provide a repository for managed items. The system automatically appends a version identifier that essentially renames the artifact to differentiate it from previous versions.

Version control activities are partially automated using tools such as CVS and Subversion; however, there is usually some portion of the configuration management process that relies on policies and human cooperation. Developers must branch to protect the main stream of development, they must check in code that does not break the build, and the process of declaring a baseline must ensure integrity as future development builds on the baseline.

Once assets or products are baselined, changes to them are controlled. Often a change control board with wide representation approves the content and timing of changes. This coordinates the inevitable ripple effect of changes to dependent assets. The oversight ensures changes do not interfere with near term deliveries or other milestones unless the change is sufficiently high priority.

Every development effort should have a comprehensive, overarching CM_t plan but the philosophy of CM_t must be woven into the individual role descriptions in the development processes. The plan should coordinate the efforts of those developing assets and products, those who deploy the products, and those who maintain them. The plan coordinates between levels of control, perhaps at the developer, team and project levels. The plan describes such procedures as merge processes that define how new work is incorporated into the main stream of development and patching processes that control the amount of variation between separate branches of a basic module.

The plan describes a governance structure that may include multiple change control boards. Horizontal and vertical lines of authority among the boards correspond to the management structure. Individual teams have a first level change board. These boards interact with other boards on the same level when there is a dependency between the teams. A high level project, business unit, or organization board will hear appeals from the lower levels of control.

Berczuk provides a set of configuration management patterns[Berczuk 01]. I will list a few of them and give my spin on them. I suggest interested readers check out the full list.

- **Private versioning** – I want everything I do to be retrievable but I often branch off to try things in increments that are not appropriate for checking into the project-level repository. I want a private “sandbox” where I can play without disturbing others. This can be implemented simply by providing independent repositories for each developer.
- **Incremental integration** – The “Big Bang” may be an interesting theory in astronomy but not in software development. Combining small pieces involves far fewer dependencies and far less potential for conflict than waiting until all the pieces are ready and integrating all at once. Incremental integration is much more likely to happen if integration is automated. Often tools such as ***make*** and ***ant*** are used for automating builds.
- **Independent builds** – I want to be able to build my code, using a recent baseline, but I do not want to be distracted from the development of my logic to handle the changing dependencies brought on by new builds. Each developer needs a private area, see the first pattern, where they can build their code with a recent, but not ncessarily current, build until they are ready to check the code into the project-level repository. Often this is implemented by allowing each developer to create a branch of their own.

These patterns suggest a number of features of the version tree being managed, more about this later.

3 TIME AND SPACE

In a software product line organization artifacts vary not only in time but also in the product variation space. Existing version control systems typically are linear in time. That is, each revision of an artifact is considered the "next" version of the artifact with the previous version now being out of date.

Periodically a branch is created in this linear progression. This allows a developer to explore an idea without disturbing the linear progression until the idea has proven valuable. Work proceeds linearly along this branch until the new idea is verified or abandoned. Then the work on the branch is either merged into the main line of development or is pruned from the version tree.

CM_t builds on version control by maintaining a description of the set of resources that comprise an artifact. Just as artifacts evolve, configurations also evolve. New versions of various resources are accepted into the configuration and it must be versioned to manage the changes. Each new version of a configuration should be regression tested the same as any other asset.

A software product line organization also varies across its multi-dimensional variation space. Elements may belong to multiple configurations at the same time. Individual assets may evolve at different rates from the other assets. By extending the CM_t patterns discussed above: there should be separate version trees for each asset and each product. This is where CM_t is no longer sufficient and CM_s is required because now there should be some degree of coordination among the version trees.

4 EVOLUTION

Change management, $CM_{s,}$ adds a strategic view of the evolution of the assets and products to the tactics of CM_t. CM_s encompasses planning, policies, and processes for managing evolution of assets and products. In a product line organization, CM_s must be multi-dimension to accommodate the wide range of dependencies that exist among assets and products.

Evolution is long-term change, where long-term is relative to the domain. Six months is long-term in the cellular telephone domain but three years is more realistic in satellite control systems. Applying patches to repair defects and extending an assets's behavior to meet new requirements are just some of the forces that drive the evolution of an asset.

Figure 1 represents the Evolve Each Asset product line pattern [Clements 02]. This pattern describes the various practices that are needed to accommodate the evolution of an asset [SEI 06]. Configuration management is just one of several practices that are needed.

- Technical Planning – The owner of an asset plans enhancements to the asset. Changes are coordinated so that the impact on other assets is accommodated and so that periodic releases of an asset can be accompanied by releases of required dependencies.
- Tool Support – The tools used to produce an asset, whether a word processor or an integrated development environment, should support automatation of as much of the asset development as possible. Templates and high level models support this automation. These tools should interface with the configuration management tool to manage the changes to the asset.
- Process Definition – The process by which an asset is used to produce products must be updated as the asset evolves. As an integral part of the asset this process is attached to the asset and is part of any configuration that includes the asset.
- Testing – An automated regression test suite is used to check the correctness of modified assets. As an asset evolves, its regression test suite must evolve as well. (The test suite for a non-code asset is a list of scenarios used to review the asset.) The test suites, text plan, and test data are part of the development configuration for the asset.
- Configuration Management – Beyond configuration management tools, policies and procedures guide the asset developer in how to control and manage the evolution of the asset. This practice is woven nto the development process so that asset developers will operate the process as a integral part of their everyday work.
- Data Collection, Metrics, and Tracking – Program assets may be measured for qualities such as performance and security while non-program assets may be measured by their completeness or clarity. Complexity measures and measures of "goodness" should be tracked to keep the evolving asset within manageable limits.

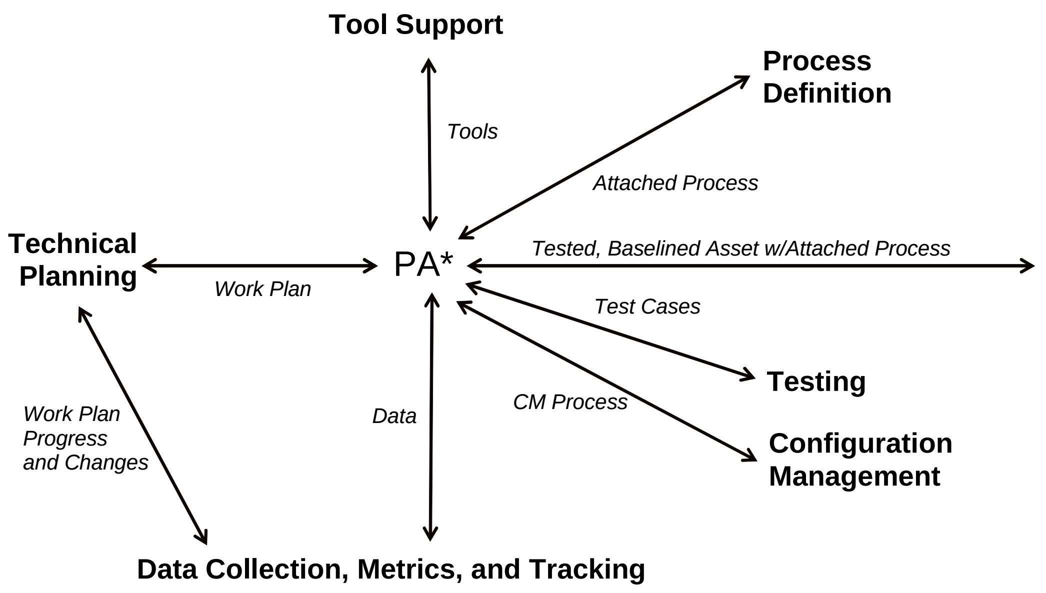

Figure 1 Evolve Each Asset

5 PRODUCT LINE EVOLUTION

In a software product line organization, a number of types of changes must be anticipated, planned for, and managed.

- A new variant value might be added to an existing variation point to accommodate evolution in the product domain.
- Adding products to the product line may lead to the identification of a new variation point.
- Splitting a variation point into two variation points may result in an expanded scope and the need to add or modify assets.
- Requests for new products will drive changes to the membership in a product line and hence its scope.
- Eventually sufficient new variation may be identified that it becomes useful to consider splitting into multiple product lines.

Mohan and Ramesh present three recommendations for change management in a software product line [Mohan 06]:

1. Modularize changes and variation points – The intent is to encapsulate a variation point to prevent changes at that point from rippling into other parts of the products.
2. Track the scope and life of variations – The strategic part of these recommendations is to establish traceability of variations over time and products. Maintenance of variations is a necessary part of the product line life cycle and traceability is critical to the efficient modification of the core assets.

3. Facilitate reuse based on knowledge sharing – The original variation architecture must be communicated to all the designers and then changes to that architecture must be propagated as well. The feature model and the software architecture are vehicles for this notification.

Product line practice is to create a business case for a product to determine that it is economically sound to include the product in the product line. The change management process should ensure that appropriate practices are in place to control the evolution of the product line definition, represented in the scope definition. Product line patterns such as “What to Build” identify dependencies between the scope and business case assets. These patterns define a type of configuration in which the assets produced by one practice have dependencies on assets produced by other practices.

In a software product line organization many products may be under development at the same time. Each product will need some product unique artifacts but some of these will be promoted to core assets eventually. The CM process can interact with the promotion process to ensure that the artifact that becomes a product line-wide asset still supports the original use that brought it into existence. One product line evolution guideline is that promoted assets should remain backward compatible with their previous use.

6 IMPLICATIONS OF CHANGE

The change management system must help preserve three characteristics of the assets and products:

- **Correctness -** Changes in assets and products must be tracked and propagated so that the change results in the original target asset being correct and all affected assets and products still bveing correct.
- **Completeness** – Changes in assets and products that modify a specification must be reviewed to determine that the new specification is complete with respect to the commitments made to other assets and products. Changes in assets and products that affect the membership of the product line must be accompanied by a validation of the scope definition.
- **Consistency** – Changes in assets and products must maintain “appropriate consistency” in the asset base and the product portfolio. By “appropriate” I mean that a product line asset base may contain contradictions among variant choices for a specific variation point but a particular configuration of assets should be internally consistent.

Each of these characteristics require traceability mechanisms to ensure that the evolutionary ripples do not adversely affect related assets. Traceability is enhanced by a development approach that emphasizes modeling. The Unified Modeling Language (UML) provides a means of capturing relationships among conceptual entities[OMG 06]. Models of algorithms provide connections among the pieces of data. Beyond the software, the Software Process Engineering Meta-model (SPEM) provides the basis for models that capture relationships among process elements such as the various workproducts being controlled [OMG 05]. Models built using the Eclipse Process Framework MethodComposer are examples of process models[Eclipse 06] that provide sufficient information to support predicting the impact of changes made to one asset on the other assets. These models focus attention on the set of assets that should be inspected or tested to ensure they are still correct, complete, and consistent.

Periodic reviews, triggered by some milestone or event such as a change request being cleared, are needed to verify that assets still possess the appropriate properties. The change management process must have wide participation. All stakeholders need to participate in the reviews and be represented on change control boards to ensure that all perspectives are represented as changes are considered.

7 SUMMARY

Change is inevitable. Anticipating and managing change can have a strategic impact on the organization and its ability to produce products. Tools, policies, and processes are needed to manage both the short-term changes and the long term evolution. Version control, configuration management, and change management come together in a comprehensive strategy to support the production capability of the organization. These techniques provide an infrastructure that supports the controlled modification of completed work and that manages the dependencies among the assets and products to ensure that changes are appropriately propagated. By structuring the infrastructure to be compatible with the relationships among assets, change management becomes more tightly integrated and more supportive. This type of infrastructure is strategically important in software product line organizations whose success is predicated on long-lived, multi-use assets.

REFERENCES

[Berczuk 01] Steve Berczuk. Configuration Management Patterns, http://www.bell-labs.com/cgi-user/OrgPatterns? ConfigurationManagementPatterns, 2001.

[Clements 02] Paul Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. Boston, MA: Addison-Wesley, 2002.

[Eclipse 06] www.eclipse.org. Eclipse Process Framework v. 1.0, 2006.

[McGregor 03] John D. McGregor. The Evolution of Product Line Assets, Software Engineering Institute, CMU/SEI-2003-TR-005, 2003.

[Mohan 06] Kannan Mohan and Balasubramaniam Ramesh. Change Management Patterns in Software Product Lines, Communications of the ACM, v. 49, n. 12, 2006.

[OMG 05] Object Management Group. Software Process Engineering Meta-model v. 1.1, 2005.

[OMG 06] Object Management Group. *Unified Modeling Language v 2.1*, 2006.

[SEI 06] Software Engineering Institute, "Framework for Product Line Practice," http://www.sei.cmu.edu/productlines, 2006.

About the author

Dr. John D. McGregor is an associate professor of computer science at Clemson University and a partner in Luminary Software, a software engineering consulting firm. His research interests include software product lines and component-base software engineering. His latest book is *A Practical Guide to Testing Object-Oriented Software* (Addison-Wesley 2001). Contact him at johnmc@lumsoft.com.