CM Crossroads
the Configuration Management Community

Learning from Concurrent, Parallel, and Distributed Systems Design by Brad Appleton

by **Brad Appleton, Steve Konieczka, Steve Berczuk – June 2004**

This month we do a bit of a "context switch", from the world of parallel development, to the world of concurrent, parallel, and distributed systems design (and back again). The purpose is to see if any of the same patterns of concurrent, parallel, and distributed processing apply to the case of concurrent, parallel, and distributed development.

The Multiple Dimensions of Parallel Development

Multiple projects, multiple variants, multiple products, multiple teams, multiple sites, multiple customer/install bases, … multi-**everything!** Why does it all have to be so complicated! Each "multi"-something introduces a new dimension of complexity and scale for software development. The more "multi's" we have, the more diverse and complex the task of managing, organizing, integrating, coordinating and tracking all of the work.

Agile development methods tend to focus on simplicity and keeping things simple. When it comes to process, they also believe that it's usually better to start small and scale-up by adding incrementally instead of starting with a large all inclusive menu and trying to pare down. As a result, most agile methods don't have too many explicit practices to handle the cases of multiple projects, multiple variants, multiple products, multiple teams, multiple sites, and multiple customer/install bases. Instead the preference is to first try to find ways to eliminate these scenarios before trying to find practices to handle them.

This is probably sound advice as many of us are often too quick to jump to a solution before asking if the problem is one that really should be solved rather than avoided. Still, for many of us, it is nonetheless the case that these scenarios are business realities that aren't going to go away anytime soon. So we nevertheless must seek solutions for these problems.

Agile methods and the agile community sprang from software patterns and the patterns community. So patterns are often a good first place to look for finding such solutions. However, the existing body of patterns literature devoted to CM is substantially smaller than for software design patterns. When we cant find what we're looking for in the CM patterns literature, we may need to look elsewhere.

Core Concepts for Concurrency

If we look at the world of concurrent/parallel and distributed systems design, there are many common concepts and solutions that may also apply to the domain of parallel development. We just need to comprehend them, and how to map them from their "native" domain into our own, so we can see if they are applicable to us!

The literature on concurrent/parallel and distributed computing is fraught with technical jargon about processors, processes, and threads (among other things). Here is an oversimplified primer of some basic concurrency concepts:

- A process is a "task" for a processor to execute. It includes the logical flow of operations to execute, and an area in memory to store the results of computations. Processes may be heavyweight or lightweight; Lightweight processes are called threads.
- *A heavyweight process* has its own separate flow of control/execution, and its own storage area (address space).
- *A lightweight process* (or thread) has its own separate flow of control but executes in a shared address space with other threads

With that in mind, we can now describe concurrent processing, parallel processing, distributed processing, and multi-threading.

- Concurrent processing amounts to doing more than one thing (executing more than one process) at the same time with the same processor.
- Throw in another processor, and we have parallel processing: two processors executing at the same time to perform separate (but possibly related) tasks
- Put them on different machines in a network, and we have distributed processing
- Multi-threading (a.k.a. multi-threaded processing) is literally multi-tasking! It is when multiple threads are active at the same time for a single process.

For a better explanation of these and other concurrency concepts, I heartily recommend chapter 5 of **Patterns of Enterprise Application Architecture**, which explicitly uses examples from the area of source-code version control "because it's relatively easy to understand as well as familiar. After all, if you aren't familiar with source code control systems, you really shouldn't be developing enterprise applications." [2]

Mapping to the Parallel Development Domain

How does it all translate into parallel development? I would suggest the following is one valid translation (and there are probably others):

- A process is like a development task (or set of related tasks), and a place to capture the evolving contents of artifacts that were created/updated as a result of the work done.
- A heavyweight process is like a project for a particular release/variant of a product or component. It can also correspond to a subproject for a significant feature.
- A lightweight process (thread) corresponds to a change-task to develop all or part of a particular fix, feature or enhancement.
- A storage area (address space) corresponds to either:
    - A workspace in which files may be modified and temporary versions created. Or it may instead correspond to …
    - A version-space, in which versions may be created (e.g., a version repository, or a branch or codeline).
- Concurrent processing corresponds to simultaneous update of a file, component, or codeline
- Parallel processing corresponds to working on multiple versions (e.g., releases, variants) of a product or component at the same time (often called multi-project development)
- Distributed processing corresponds to working on the same project across multiple networked sites at the same time (often called multi-sited development).

- Multi-threading is like having multiple change-tasks active at the same either using the same codeline, or within the same workspace.

Big Deal! So what can I do with that?

This means the field of concurrent, parallel and distributed systems is ripe for pattern mining. By studying existing patterns and solutions in that field, I can identify possible candidate patterns and solutions in the other. The trick is to always keep in mind what the corresponding problem and solution means in the new context, and if it translates into something valid or well-formed that still makes sense and is still practical. (And of course remembering that people can behave and respond quite differently than source code  !)

With that in mind, let's take a look at a sampling of related patterns in this field. I chose a number of sources that are already documented in pattern form, and for which substantial material is available both online and in book form. By taking a quick tour of these patterns, we can make a quick initial guess at the potential applicability of the pattern to parallel development. I won't pretend that these initial guesses are completely accurate, nor definitive. And I'm very much interested in hearing others' ideas of how these (and other) patterns do or do not "translate" into our CM-related context.

If we are right, then in many cases these "translations" will correspond to some existing tried and true CM patterns for parallel development. In other cases, they may correspond to new or uncommon solutions - and we need to question their applicability, feasibility, or practicality unless we know more about their successfully recurring use on "real world" projects.

Fowler's Patterns of Enterprise Application Architecture [2]

Since we already mentioned this book, let's start by looking thru some of its concurrency-related patterns. (A catalog of these EAA patterns is available online at http://www.martinfowler.com/eaaCatalog/)

> **Transaction Script** organizes a set of related interactions between a client and server as a single procedure. For us, this might correspond to a transaction-based model of CM where a set of related changes to a related set of files is treated as a single change "transaction" in the version repository [3].
> Another interpretation might be that the transaction script is really the automation of a Task-Level Commit and/or Workspace Update that automatically determines the necessary locking or merging and facilitates the resolution of conflicts and atomic-checkin of the resulting changes.
>
> **Unit of Work** seems related to **Transaction Script**, in that a Unit of Work "maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems." So if a Transaction Script corresponds to an Update/Commit operation for a change-task, perhaps the Unit of Work corresponds to the change-task itself so that we may do Task-based Development.
>
> **Remote Façade** provides a wrapper around a set of fine-grained objects that allows them to be manipulated as a collective coarse-grained object. This is allegedly done for performance

efficiency over a network.

This might correspond to fine-grained management of artifacts where the units of checkin/checkout would be individual class methods, module subroutines, or requirements, but the interactive editing environment allows them to be treated as part of a larger overall structure (a document, design model, or directory hierarchy) which permits structure-wide manipulations like search/replace and/or various refactoring operations.

Interested individuals should take a look at commercial requirements tracking tools like *Telelogic DOORS®* or *Rational Requisite Pro®*, and also the *Stellation subproject of Eclipse* for an example implementation at http://www.eclipse.org/stellation and its applicability to Model-Driven Development Environments.

Data Transfer Object collects and carries data between processes "in bulk" to reduce the number of inter-process procedure calls. This might correspond to a kind of "Patch"-set or Update-Packet for remote update of a repository from a networked client at a different site.

Optimistic Offline Lock "prevents conflicts between concurrent business transactions by detecting a conflict and rolling back the transaction." This seems similar to the part of a two-phased atomic commit operation allows either all of the changes to be committed to a codeline, or none of the changes to be committed.

Pessimistic Offline Lock prevents concurrent transactions altogether by allowing only transaction at a time to access data. This seems like classic file-locking to prevent concurrent checkouts of the same file, or like *Full-Codeline Locking* [4] to prevent any other changes from being committed to the codeline at the same time (even if they are accessing completely different sets of files).

Coarse Grained Lock allows locking of a group of related objects together with a single lock. Like Remote Façade, this pattern also seems related to fine-grained artifact management. It could also correspond to locking all the files in an associated directory or component with a single (possibly scripted) operation.

Implicit Lock enforces locking policies/protocols using custom implementations that automatically perform the necessary checking and acquiring/releasing of locks in accordance with their protocol. In parallel development, we do this every time we write a script or program to automate tasks like *Workspace Update* and/or *Task-Level Commit* as part of a *Two-Phased Codeline Commit* [4].

There are other candidate patterns in this book, and even more in its companion book Enterprise Integration Patterns [5] which may be applicable to parallel development. I encourage readers to take a look and submit their feedback to the appropriate CMCrossroads.com forum and/or the scm-patterns YahooGroup!

Paul McKenney's "selecting locking designs for parallel programs" [6]

This work is available online at http://www.rdrop.com/users/paulmck/ and describes approximately ten locking design patterns. First it discusses the various factors that force or toward or away from a

particular style of locking solution. It discusses factors like: speedup, contention, overhead, read-to-write ratio, economics, and solution complexity. I think we can all appreciate how the majority of those issues relate to parallel development. Next the paper describes the various locking design patterns, including the following:

> **Code Locking** is "guarding" access to execution of procedural code and represents the standard critical-section locking in operating systems with semaphores for mutually exclusive access. Offhand, I'm not immediately sure what this corresponds to in the SCM world. Maybe it would be like locking a development-task (or portion a thereof). Maybe one of you reading this has an alternate idea about this that that you could share.
>
> **Data Locking** is akin to locking-out access to all or part of an artifact. I might lock an entire file, or maybe just a version of a file (against checkout).
>
> **Data Ownership** involves assigning data to reside on a particular processor. If we think of data as being source-code and a processor as a developer, this would seem to be the well known *Code-Ownership* pattern. There is a big difference though: If the processor owns the data, its not possible for others to use it without going through that processor, no added restrictions are needed to enforce this.
>
> So this form of "Code Ownership" is not the same as exclusive access. It really corresponds to exclusive access control: The owner is the primary "accessor", but has the authority to permit others to checkout/checkin of the artifacts owned. Some might call this *Code Stewardship* rather than strict ownership.
>
> **Parallel Fastpath** uses aggressive locking for most of the workload of a program, and conservative locking for the rest of the work. This might be similar using strict file-locking for exclusive access only for files that can't really be merged (like some graphics files, models, or documents), or which suffer from very high checkout contention (perhaps that part should be restructured or refactored to reduce contention). Or perhaps the *Phased Codeline Locking* pattern [4] is a more accurate example of Parallel Fastpath.
>
> **Hierarchical Locking** is like locking all the elements in a hierarchical data-structure, then traversing to the desired element an obtaining a lock only on that element while releasing the lock on the whole tree. This seems reminiscent to the **Remote Façade** and **Coarse-Grained Locking** patterns from [2]. Another possibility is the *Double-Checked Codeline Locking* pattern [4], which first does a *Full Codeline Lock*, followed by a *Partial Codeline Lock*, and then releases the codeline-wide lock.
>
> **Allocator Cache** is a scheme for allocating data structures. Normally there would be a single "data-pool" from which to allocate space for such structures and the data-pool would probably be on a particular processor. Using this pattern, you create several data-pools (one per processor). Possible "mappings" to parallel development might be a *Version Cache*, an *Object Cache*, or a *Workspace Pool*.
>
> > *A Version Cache* contains read-only copies of the most commonly accessed versions of the most commonly access files. Rather than taking the time to compute and extract the contents for these versions in every workspace that references them, the contents are already in the cache (eliminating the version content computation overhead), and possibly

linked rather than physically copied (saving file copying and storage at the possible expense of additional network file-access overhead).

- Such a version cache is commonly used in conjunction with a *View-Path* (e.g. VPATH variable in GNU **make**): the development workspace contains only the files it needs to checkout and modify, and all other files are referenced from the version cache via the view-pathing mechanism.

An Object Cache is similar to a version cache, except that it holds copies of already built (derived) objects and libraries for the sake of build-performance improvement of incremental builds. If none of the built objects dependencies and source files have changed (possibly including build/compile options), then the built object is reused (copied or referenced) instead of being recompiled/relinked during an incremental build of the software.

- Sometimes an object cache is used to do *Workspace Seeding*, in which a private workspace is first configured (possibly from a version-cache) and then "pre-seeded" with previously built objects that match its current configuration.

A Workspace Pool is another possibility (although it might correspond more closely to the J2EE *Connection Pool* pattern [7]). It is a pool of pre-allocated (maybe even pre-configured) workspaces for developers to make changes. A *Workspace Pool* can reduce the overhead associated with creating and configuring a new workspace for each task/developer. I have seen this as part of a more sophisticated overall strategy that also employed a *Version/Object-Cache, View-Pathing, and Workspace-Seeding*.

Critical Section Fusing and **Critical Section Partitioning** are the "inverse" of one another. The former combines critical sections of code together (if they are commonly accessed at the same time, or perhaps in strictly sequential order every time); the latter splits up a critical section with high contention into multiple sections with fewer contention (since its assumed not all the demand is for the same part of code at the same time). These correspond to what are probably commonplace refactoring patterns [8] (see http://www.refactoring.com/catalog/).

File Splitting is a common type of *extraction* refactoring pattern that takes a high-checkout-contention source file and splits it up into multiple files. This not only reduces checkout-contention, it also reduces the likelihood of merging and merge conflicts, and possibly also the amount of recompilation when rebuilding.

File Fusing is a common type of *consolidation* refactoring pattern where two (or more) files have code that so frequently must all be modified together that is makes sense to consolidate the related sections of code the same module/class in a single source file that already exists.

Doug Schmidt's TAO/ACE Patterns [9]

Doug Schmidt and colleagues have published a significant collection of network computing design patterns for concurrent, parallel and distributed systems. The patterns are used extensively in the public domain ACE and TAO C++ and Java frameworks (see

http://www.cs.wustl.edu/~schmidt/patterns-ace.html). I won't delve into descriptions of the patterns here; I'll just briefly describe some possible "translations" into the domain of parallel development:

Active Object appears to resemble a combination of Transaction Script and Unit of Work to implement the concept of a *Change-Task* for Task-Based Development.

Scoped Locking appears similar to implementing a *Two-Phased Codeline Commit* [4] mechanism to automatically lock and unlock the codeline at the appropriate times during the two-phased commit process.

Double-Checked Locking Optimization bears some resemblance to the *Double-Checked Codeline Locking* pattern [4]. (In fact, the name of the latter pattern was "inspired" by that of the former pattern.)

Thread per Request and **Thread per Session** seem strikingly similar to the commonly recurring practices of using a private *Branch per Change-Request*, and of using a private branch per developer (a.k.a. Personal Activity Branch) [10]

Reactor, Proactor, and **Acceptor-Connector** patterns seem reminiscent of branching patterns for codelines that are used to add another level of indirection by adding another line of integration for mediating or moderating between development activities with incompatible codeline policies for integration frequency and/or stability (e.g., *Docking Line, Stable Receiving Line & LAG Line, Inside/Outside Lines, Remote Line, and Third Party Codeline* [10,][1])

Asynchronous Completion Token is reminiscent of an *Integration Token* [4], or an Integrator-Pull model of integration (where a developer hands off to an integrator to do the merging back to the codeline), or a *Token-enabled Developer Push* model of integration where a developer is allowed to merge their own changes to the codeline, but must first wait for a *merge-token* from an integration coordinator who tells them which codeline to merge into and/or when it is safe to do so.

The service access and **connfiguration** patterns of *Wrapper-Façade, Component Configurator, Interceptor, and Extension Interface*, seem to require no "translation" whatsoever. They all are effective ways of partitioning and encapsulating source code for variations in platform, functionality, and services using object-oriented design for what otherwise might easily correspond to version branching and/or conditional compilation that would make for significant additional merging effort.

Ouch! My Brain Hurts!

There are several other excellent sources of concurrency, parallelism, and distribution patterns we could look at, particularly [5],[7],[11], and [12]. But we already covered a lot of material and we just careened through at a blindingly rapid pace. And for all that effort, we didn't even sound very certain of some of the "domain mappings" we attempted. Not only that, we only mapped them to patterns/practices we already knew, and not to anything new. So not only did we just overwhelm you with new information, we didn't discover anything new in the process! Or did we?

First off, we hopefully gave you a better grasp of how some basic concepts of concurrency and

distribution translate into parallel development concepts. We also had a taste of what some common concurrency problems are, where to look to see how they are addressed in programming systems, and to find out more about the intricacies of the issues (forces) involved and how to go about resolving them.

A more in depth look at the literature would give us an understanding of terms like: safety (or correctness), liveness, deadlock, synchronization and asynchronous communication, mutual exclusion, semaphores, monitors, guards, resource contention, starvation and sharing, etc., and (more importantly) how these translate into everyday issues in parallel development regarding stability, consistency, productivity, coordination, communication, isolation/insulation of risk, encapsulating and minimizing change impact and variation. Understanding these classic problems and the forces that drive them from the programming domain gives us some more formal conceptual mechanisms for systematically reasoning about parallel development problems, devising solution alternatives, and evaluating the corresponding trade-offs.

Of course we've also presented a large amount of fodder for readers to "mine" the existing problem space for new parallel development pattern candidates and a few forums in which to discuss and refine them. However, not everyone has the time or patience to do all that legwork and would rather see others consolidate and present it for them.

A few of us already attempted this with the existing patterns literature several years back and the result was the pattern collections in [10], [4] and [1]. So please look at those if you wish to see cohesive sets of CM patterns that address parallel development issues. I'm hoping however that this article piqued your curiosity enough to motivate you to take a look for yourself at what's out there (especially some of the more recent works) and begin your own study to learn and discuss (and eventually disseminate) the results of your investigations. For those that do (and you are strongly encouraged to do so), we look forward to hearing from you in the CMCrossroads.com forums and the scm-patterns YahooGroup!

REFERENCES

[1] Software Configuration Management Patterns: Effective Teamwork, Practical Integration; by Stephen P. Berczuk and Brad Appleton; Addison-Wesley, November 2002

[2] **Patterns of Enterprise Application Architecture**; by Martin Fowler et. al., Addison-Wesley, 2002 (also see http://www.martinfowler.com/eaaCatalog/)

[3] *Configuration Management Models in Commercial Environments*; by Peter H. Feiler; **SEI Technical Report CMU/SEI-91-TR-7**, March 1991 (also see http://www.sei.cmu.edu/legacy/scm/abstracts/abscm_models_TR07_91.html)

[4] **Codeline Merging and Locking: Continuous Updates and Two-Phased Commits**, by Brad Appleton, Steve Konieczka and Steve Berczuk; CM Crossroads Journal, November 2003 (Vol. 2, No. 11)

[5] **Enterprise Integration Patterns**; by Gregor Hohpe et. al.; Addison-Wesley, 2003 (also see http://www.enterpriseintegrationpatterns.com)

[6] "Selecting Locking Designs for Parallel Programs", by Paul McKenney in **Pattern**

Languages of Program Design 2, ch. 31, pp.501-531; Addison-Wesley, 1995 (see http://www.rdrop.com/users/paulmck/)

[7] **Core J2EE Patterns: Best Practices and Design Strategies** (2nd ed.); by Deepak Alur, et.al.; Prentice-Hall, 2003 (also see http://java.sun.com/blueprints/corej2eepatterns/)

[8] **Refactoring: Improving the Design of Existing Code**; by Martin Fowler et.al.; Addison-Wesley, 1999 (see also http://www.refactoring.com/)

[9] **Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects**; by Douglas Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann; John Wiley & Sons, 2000 (also see http://www.cs.wustl.edu/~schmidt/patterns-ace.html and http://www.cs.wustl.edu/~schmidt/ACE.html and http://www.cs.wustl.edu/~schmidt/POSA/)

[10] Streamed Lines: Branching Patterns for Parallel Software Development; by Brad Appleton et. al.; (see http://acme.bradapp.net/branching) in **Proceedings of the 5th Annual Conference on Pattern Languages of Program Design**, Allerton Park, IL (PloP'98)

[11] **Concurrent Programming in Java: Design Principles and Patterns** (2nd ed.), by Doug Lea; Addison-Wesley, 1999 (also see http://gee.cs.oswego.edu/dl/cpj/)

[12] **JavaSpaces Principles, Patterns, and Practice**; by Eric Freeman et.al; Addison-Wesley, 1999 (also see http://java.sun.com/docs/books/jini/javaspaces/index.html)

[13] **Patterns for Parallel Programming**; by By Timothy Mattson, Beverly Sanders, Berna Massingill; Addison-Wesley, 2004 http://www.awprofessional.com/title/0321228111

[14] *Architectural Patterns for Parallel Programming*, papers and Ph.D research by Jorge Ortega-Arjona. See http://www.cs.ucl.ac.uk/staff/J.Ortega-Arjona/
[/LIST]

Brad Appleton is co-author of Software Configuration Management Patterns: Effective Teamwork, Practical Integration. He has been a software developer since 1987 and has extensive experience using, developing, and supporting SCM environments for teams of all shapes and sizes. In addition to SCM, Brad is well versed in agile development, and cofounded the Chicago Agile Development and Chicago Patterns Groups. He holds an M.S. in Software Engineering and a B.S. in Computer Science and Mathematics. You can reach Brad by email at brad@bradapp.net

Steve Berczuk is an Independent consultant who has been developing object-oriented software applications since 1989, often as part of geographically

distributed teams. In addition to developing software he helps teams use Software Configuration Management effectively in their development process. Steve is co-author of the book Software Configuration Management Patterns: Effective Teamwork, Practical Integration. He has an M.S. in Operations Research from Stanford University and an S.B. in Electrical Engineering from MIT. You can contact him at [Email]steve@berczuk.com.[/Email] His web site is www.berczuk.com

Steve Konieczka is President and Chief Operating Officer of SCM Labs, a leading Software Configuration Management solutions provider. An IT consultant for 14 years, Steve understands the challenges IT organizations face in change management. He has helped shape companies' methodologies for creating and implementing effective SCM solutions for local and national clients. Steve is a member of Young Entrepreneurs Organization and serves on the board of the Association for Configuration and Data Management (ACDM). He holds a Bachelor of Science in Computer Information Systems from Colorado State University. You can reach Steve at steve@scmlabs.com