CM Crossroads

the Configuration Management Community

Tasks and Branching Patterns by Brad Appleton

by Robert Cowham, Brad Appleton, Steve Berczuk

This months article builds on some recent discussions regarding task related branching patterns [1]. The key question seems to be how to develop individual tasks, potentially in parallel, to achieve:

- Appropriate isolation of task implementation
- Minimization of merging
- Tracking task implementation across branches, and in particular to releases (and thus automatically into release notes – an excellent indicator of process maturity[2])

What is a Task?
As described by Austin Hastings, the Task-Level Commit (TLC) pattern [3] is used to group a set of related changes to separate objects into a single task level operation which can be tracked and propagated across releases. Ideally this requires some level of SCM tool support for “change sets” or the equivalent.

The question is how to most successfully manage these tasks?

Task Branch Pattern
The classic “Task Branch” pattern [4] is recommended for development tasks that take a long time to implement and where intermediate steps are potentially disruptive to other users, or other code lines (Mainline or Active Development Line).

Some people get confused by this pattern and what *Streamed Lines*[5] calls “Branch per Task”. The difference between Task Branch and Branch per Task is one of policy. Task Branch simply says: here is a problem and a context in which creating a Task-Branch is a known best practice. Branch per Task says **always** create a Task-Branch for every development task (whereas "Branch per Major Task" says always create them for "major" tasks).

Notice that we do not assume that a large feature/fix/enhancement is somehow merged only once, and all at the end. We can split it up into multiple, smaller development tasks and merge those to the trunk one at a time instead of all at once. This is basically a form of *incremental integration* within a given feature/fix/enhancement.

Task Branch doesn’t assume the granularity is constrained to be as large as the entire feature/fix/enhancement. What it does assume is that **if** the feature/fix/enhancement can be split into independent development subtasks, that they can be integrated incrementally. That means it would have to be both possible and permissible to integrate working partial functionality of a fix/feature/enhancement (see later for when this might not be possible).

The amount of merging that must take place is mostly the same regardless of whether we use branch-per-task, branch-per-major-task, or no task-branches at all but we still use task-level-commit.

Ultimately every file checked-out and updated must be merged back to the integration codeline. The issue is whether it is all done at once, or in smaller increments, and how "out of sync" our changes (or "the baseline") can get in between integration intervals.

Incremental Integration
Here, *incremental integration* is a generally accepted best practice, and integrating in smaller chunks is usually better than integrating in larger chunks provided that:

- everything can and still does work correctly; and ...
- integration of working partial "changes" is not an issue for releasing or propagating (if they are an issue, it will impact you no matter which of the strategies you use).

In some organizations, there is a significant difference in both time and criteria between what development calls "good" and what QA/Test/CM calls "good":

- If the time difference is substantial, then development can quickly get out of sync with other parallel changes and have a lot of merging that could be avoided if "approval" could happen much faster.
- If the "criteria" difference is substantial, bad things can happen. Plus it can lead to even more battles between development and the approving group regarding their differing criteria.

The other problem is that using branches for tasks or fixes/features/enhancements is predominantly done for the purpose of either *isolation* and/or *identification*, where:

- *isolation* is the bundling together of all the deltas for a change so they can easily be propagated to multiple codelines
- *identification* is the bundling together of all the deltas for a change so you can refer to them (and track them) with a single name or identifier

Using a branch for a task or feature is definitely valid (often even necessary) if your purpose is isolation. But if your purpose is solely for identification, then branching is not the best way if isolation isn't necessary and we should instead look for other ways of indicating a change-task besides isolating it on a branch when isolation isn't needed.

Thus the important things are:

- **Task-Based development is a good thing** that should be encouraged. See Austin Hasting's *Building on 'Task-Level Commit'* [3]. If you can readily identify and track change-tasks without forcing them to be on a separate branch, then don't use branches for that purpose unless isolation of the changes is also necessary for your overall release strategy (e.g. across multiple branches/projects/releases/variations of your product).
- **Incremental Integration is a good thing** that should be encouraged, and the smaller you can

afford to make your integration intervals and tasks, the sooner you can get valuable and important feedback on codeline quality and stability (which is a good thing)

- **Having changes based on "stale" configurations is a bad thing** and should be discouraged. Here "stale" means w.r.t. any other development that has taken place. Using a stale base configuration increases the likelihood and complexity of conflicts with other changes developed in parallel since the "approval" of the "stale" configuration. This in turn increases the likelihood and complexity and effort of merges and their associated risks. See *Codeline Merging and Locking: Continuous Updates and Two-Phased Commits* in the November 2003 CM Crossroads Journal [6] for various merging strategies to minimize merge contention.

- **Get development and CM/Test/QA to agree** on each other's idea of "good", and have development strive for task-level-commit "criteria" that is as close as feasible to what CM/Test/QA want while still allowing development to proceed at their own pace.

When branches are used for the true purpose of separating legitimately parallel development efforts that would otherwise violate codeline consistency/correctness policies, they can be tremendously effective. When they are used as a mechanism to merely identify (rather than isolate) or to isolate people/communication (rather than the work they are doing), it can do more harm than good, and can work against sound principles of incremental integration and frequent team coordination.

What about Change Propagation?

How does this work when I have to propagate changes across multiple releases, projects, or variants? The added difficulty here isn't so much the additional merging as it is the *tracking* of what needs to be propagated and what has/hasnt been propagated. We now have a tradeoff of wanting to do "good" things like frequent incremental integration, versus needing to be able to track and propagate across multiple releases and/or variants.

- If you batch them up and integrate them all at once, you lose the frequent feedback and coordination benefits of incremental integration (which admittedly may seem like it benefits development more than it benefits CM).

- If you do lots of small integrations in short intervals, development gets those benefits of frequent feedback and team coordination but you have an increased tracking and status accounting problem for propagation to multiple projects/variants (which admittedly may seem like it gives more of a headache to CM than to development)

The typical solution to this decomposition + recomposition problem is either "add another level of indirection", or else dont decompose in the first place.

So you could simply choose not to use separate task-branches at all except in rare circumstances (e.g. "major tasks"). Then you'll need some other way to identify all the changes for a given task (which a branch isnt necessarily the best way to do, and which some tools give a very convenient way of doing).

Auto-Propagation of Changes for an entire Codeline
When propagating changes, you either use the mechanism that identifies all the changes for a given fix/feature, OR you dont propagate at the level of the individual fix/feature, you instead propagate an

entire codeline.

This assumes that every change for a "previous" release will necessarily have to get merged to its successor release being developed in parallel. Ive seen this done successfully as part of a Nightly/Daily Build mechanism (call it Nightly/Daily "propagation") where the ReleaseN-1 codeline is merged to the ReleaseN codeline and then some automated sanity tests are run.

The problem with this is if manual merge decisions need to be made to resolve a merge conflict, then you need to be able to resolve it very quickly. So some folks simply have it attempt the merge and see if it would successfully build, and if it does they commit the result (possibly before automated testing, or possibly after, or both) and if it does not, then notifications gets sent and problems must be resolved, just like for a typical Daily/Nightly Build set-up.

The Change "Docking" Line
Another solution is to use what the *Streamed Lines* calls a "Change Docking Line" (which is kind of like having a Release-Line separate from an Active Development Line, using the terms from the **SCM Patterns** book.

- Development has its codeline and they can create task branches of off it, and "push" commit their changes to the active-development line;

- but CM gets to choose how and when to integrate the result to the release-line: for any candidate merge, they may choose to merge from the development-line to the release line, or they may choose to merge a given task-branch to the release line.

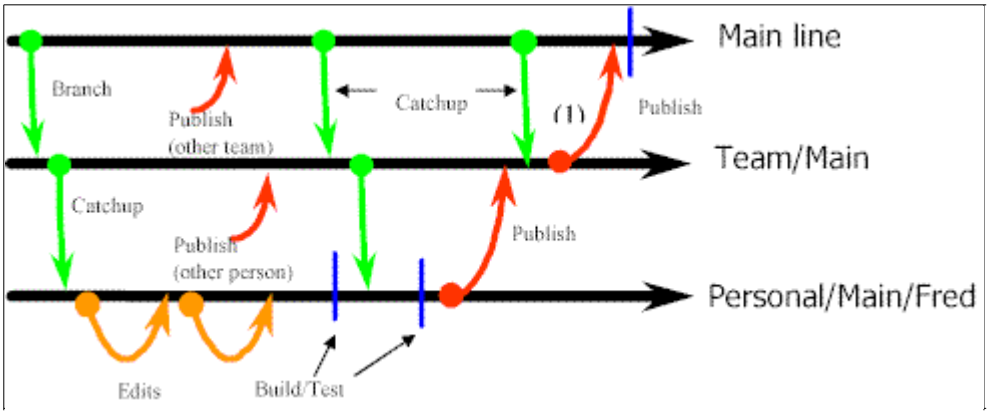
This adds an extra level of indirection by adding an extra "line" of integration for an additional level of control over the granularity and frequency of what gets merged (and when) to the release line, while still giving development freedom over the development line. And the developer for a given task branch gets to choose whether or not to base their changes off the latest configuration on the release-line or the development-line.

The ability to deliver small changes quickly can be (and often is) of tremendous benefit, especially to development (though perhaps not as "wonderful" for CM). Batching them up and releasing all at once can work really well, or can work really badly if its misinterpreted as an excuse to do big-bang integration.

Both the *Daily Automated Propagation* approach and the *Change Docking Line* approach attempt to give the best of both worlds here by enabling you to deliver small changes quickly and to batch them up and test all at once. Each has different tradeoffs associated with it (and you are going to have to make tradeoffs no matter how you solve this problem). If you allow more choices you may increase the likelihood of an inappropriate choice being made.

Case Study – Experience at Symbian [7]
Symbian produces an operating system for Wireless Information Devices such as Communicators and Smartphones. They have nearly 800 developers and the main codeline contains about 200,000 files.

They use the “Mainline” pattern, with almost no changes being made directly on the mainline. Changes are made on Task Branches which are typically Integration Lines for a team, with individual developers potentially working on their own “sub” Task Branches.



The terminology used is “Catchup” for the propagating of changes into a Task (team) Branch or sub-Task (personal) Branch. “Publish” is used for propagating of changes towards the Mainline. If we take publish to the Mainline from the team line (1) above, the key step is that immediately before this being done, a Catchup is done (propagate in the other direction). This Catchup brings any changes made by other teams or individuals into the sub branch, and means that potentially more risky merges are done not on the mainline, but on the team line. Thus the subsequent Publish should be a straight copy of code.

Indeed, it is possible to automate the Publish. A Publish will only succeed if:

- Nothing has changed on the Mainline since the last Catchup into the codeline
- After the automated merge process, but before the actual check-in of changes into the mainline both automated build and Smoke Tests [4] have been successfully performed within that workspace.

There are obvious requirements if the automation of the publish is to be possible:

- The build/compile/smoke test cycle needs to be completed in a suitable amount of time (an 8 hour smoke test means only 3 changes per day will make it in!). This may require different levels of build (e.g. incremental) and smoke test (20 minute vs. 4 hour) to be used as part of this process, thus risking the occasional subsequent break (e.g. during overnight build from scratch and complete smoke test).
- The above cycle can be replaced/enhanced by a level of QA where by the developer attempting to publish is asked to provide evidence of appropriate build/smoke tests having been performed in the Task Branch prior to the publish being done.

This particular model is working reasonably well for Symbian, although it is not unknown for the build on the Mainline to be broken (this is treated as very high priority item to be fixed).

This is what the Streamed Lines paper [5] calls a "Change Docking Line" (which is rather like having a Release-Line separate from an Active Development Line, using the terms from SCM Patterns [3]).

This adds an extra level of indirection by adding an extra "line" of integration for an additional level of

control over the granularity and frequency of what gets merged (and when) to the release line, while still giving development freedom over the development line. And the developer for a given task branch gets to choose whether or not to base their changes on the latest configuration on the release-line or the development-line.

Comments
As noted by Daniel Patterson in the original discussion [1], people often forget that they're branching much more often than they realise:

> *A branch is not necessarily an "in-repository registered 'branch'". A branch is, at its simplest, a copy of some items that is being modified independent of another copy. Whenever you try to reconcile the changes between these two copies, you have to "merge". Any version control tool that supports non-locking modifications has this concept. For example, a CVS working copy is essentially a branch, without a private history. When you "update", you merge from the mainline (the repository), into your private branch. When you commit, you merge the other way.*

Who does the merge? Push -vs- Pull Integration
For those concerned about the amount of merging effort for lots of small task-branches, is it assumed that all that task-branch-merging is being performed by a dedicated integrator (what Brian White's book [8] calls the "*integrator pull*" model of integration), or is their assumption is that each developer merges their own changes from the task-branch to the integration branch (the "*developer push*" model of integration)?

The more task branches there are in a given time-span (even if they are all fairly small), the more of a burden it can seem for a single integrator to do all that merging on a per-task-branch basis. However, if developer-push is used, then the merge effort and burden doesn't seem so bad - provided you have established a good codeline checkin policy and the developers follow it.

Here, people's own experience may vary:

- We know many build-meisters and integrators who feel that allowing developers to do their own merging is like giving a small child a loaded pistol, and they would never seriously consider it.
- We know others who have successfully worked with developers to help establish and automate such a codeline checkin policy and discipline, and who swear by it, and would never go back to an "integrator pull" model again.

Note the potential for automation which the "Catchup" propagation in the Symbian case study allows.

Conclusion
Task Branches are a good thing, but like any pattern need to be used when appropriate. If you are able to develop directly on the Mainline or Active Development Line then please do so – it will minimise potential merges! But if not, then use Task Branches intelligently, and they can achieve the goals of appropriate isolation and parallel development without undue merging overhead.

There is a separate case which we have come across when Tasks are implemented on Task Branches and not merged back to the mainline until shortly before the release because the business was unable to

decide until very late in the day what the actual contents of a release should be (and the business never changes its mind about the contents of a release… - yeah right!). This is left for a future article!

References

[1] "Branch per Change" an Anti-Pattern? - http://www.cmcrossroads.com/article/35109

[2] Release Management – Making it Lean and Agile - http://www.cmcrossroads.com/article/31243

[3] SCM Patterns: Building on 'Task-Level Commit'; by Austin Hastings; CM Crossroads Journal, June 2004 (Vol 3. No. 6)

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

[5] "Streamed Lines: Branching Patterns for Parallel Software Development", http://www.cmcrossroads.com/bradapp/acme/branching/

[6] Codeline Merging and Locking: Continuous Updates and Two-Phased Commits in http://www.cmcrossroads.com/cmjournal/nov03.html

[7] Co-operative development at Symbian - http://www.perforce.com/perforce/conf2001/jackson/jackson.HTML

[8] Software Configuration Management Strategies and Rational ClearCase: A Practical Introduction; by Brian White; Addison-Wesley, August 2000

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 steve@berczuk.com.

Robert Cowham is a Principal Consultant at Vaccaperna Systems Ltd in London, UK. He mainly provides SCM consultancy and training, but still keeps his hand in development wise! You can contact him via http://www.vaccaperna.co.uk/