Source Code Merging

Intro

This document describes the policy for code merging on the QuikView project at T-Mobile. You can find a "how to" document elsewhere on this Qwiki.

Code development at T-Mobile follows "best practices"

All parts of QuikView development have followed best practices since project inception. The following are rules regarding code merging, done throughout a sprint. These rules address several kinds of merges: from trunk to branch (a "merge up"), a branch to a higher branch (another "merge up") and from a branch to trunk (a "merge down"). Merging up from trunk to branches is a necessary practice during intervals between releases. Merging down from a branch to trunk (with reintegration) is a necessary practice for production releases. Merging from a branch to a branch is optional.

Merge Policies

1. **Merging up from trunk**: Code in branches must updated with code from trunk *periodically.* The merge interval should be set by the teams working in that branch and trunk, and it depends on the type of work being done. If the branch is experimental, merging up from trunk infrequently might be the best course. If a branch is for the next phase of the project, merging up from trunk should occur every few days.
2. **Merging up from another branch**: This merge type is optional. Developers may want to merge code up from a lower branch to pull in changes sooner than they'd see them if they wait for that branch to get merged down to trunk. Note that when a lower numbered branch is merged down to trunk (reintegration), all its changes go in trunk. The next time another branch gets merged up from trunk, all the changes now in trunk get merged to that branch. Nothing is lost. Subversion prevents a branch from being merged down to trunk if it had not previously merged up from trunk with ALL the changes. It is important for the merge teams to communicate what they're merging and when. Branch to branch merging requires more coordination than a normal, periodic merge up from trunk.
3. **Merging down to trunk**: Branches get merged back into trunk when a new version is completed, typically at sprint end or when we have a production release. Because branches have been consistently merged up from trunk, the merge back to trunk should be far easier than merging all the changes made during the sprint back to trunk. Remember, Subversion will not let you merge down to trunk if you haven't brought in all the changes currently in trunk to your branch.
4. **Refactoring**: **If you are refactoring code, it's very important to do frequent merge ups during the process**. Subversion will not help you as much when files get moved around, deleted, or renamed. Similarly, if a directory is renamed, moved, or deleted, the number of conflicts is magnified. If you wait to merge after the refactoring is complete, expect a great deal of extra work. In one case, T-Mobile developers patiently waited more than a week for a branch to be thawed while trunk was being merged up. Code had been merged up before the refactoring, but not during the process. This left the merge team with a nightmare of a merge process. If you're going to move stuff around MERGE UP FIRST! If you're moving or renaming a file, merge up before you do. Frequent merging DURING the refactoring process will prevent a lot of code conflicts.
5. **Set up Jira tasks:** Teams should allocate time in Jira tasks for merging code. Merging code should be easy, but it does require thought and time to execute. Merges from branches down to trunk take longer, and take a bit of planning. The time allocated in Jira should be set properly, with the description and time allocated matching the effort involved.
6. **Scrum teams decide which developers will merge code:** Merges are completed by developers, who are most familiar with the code and functionality of the application. This process is not done by the SCM team, tho they will be happy to provide assistance.
7. **QA testing***:*When a merge is planned, our local QA people should be notified. QA can verify the functionality of the application after the merge is complete, but before the code is checked in.

Merge Methods

Merging up can be done by either "cherry picking" code, or by merging all changes.

- **Merging it all:** This is the easier method. Typically one will run into fewer conflicts with this method. Because we're talking about merging up from trunk, stability should be very high. This type of merge is easier to track, as well. The downside is that the code being merged up could bring along changes that are undesirable. Recent full merges have taken anywhere from 25 minutes to 2 hours.
- **Cherry Pick merging**: Choosing individual files could be done by picking from Jira bugs or stories marked as completed. Find a completed story, find the files with that revision in subversion, and merge those files. This avoids merging code that's not completed and that could affect the functionality of the code in a bad way. The downside of this method is that it has a higher chance of conflicts. Additionally, after merging a Jira bug or six, the code previously checked into your source becomes further outdated. Subversion can have a difficult time merging that code later. This process also takes longer, since the person doing the merge has to track down the files and revisions of those files before starting the merge process.

Cherry Pick merges are more difficult. Subsequent merges are far more likely to have additional conflicts because of the missed revisions from the previous Cherry Pick merges. These will take longer than a typical full merge.

Find the documentation for merging with subversion here . These docs are very good, but they reference command line tools. If you're uncomfortable with the command line, don't worry. The Eclipse svn plugin being used by our developers has almost equivalent functionality. TortoiseSVN (Windows) also has some nice tools that help with svn functions.

Merge scheduling

Merges should be scheduled weekly, or more frequently. Infrequent merges mean more code conflicts when you finally do merge. This is especially true if there has been code refactoring.

Frequent merges are quick and prevent headaches. The longer you wait, the more trouble you're likely to see. To quote the Subversion manual regarding merging, "[frequent merging] is a best practice: frequently keeping your branch in sync with the main development line helps prevent “surprise” conflicts when it comes time for you to fold your changes back into the trunk." Conflicts are a bear.

Who does the merging?

Developers do the merging. Ideally, representative developers from both trunk and the branch in question handle the merge as they're most familiar with the code changes. Some discussion is necessary to determine what will be merged, noting the consequences of code changes. Jira and Greenhopper could be a big help with determining when code will get merged. Noting who made the code changes could help determine who will do the merging.

Here's a suggested process for doing a code merge.

1. Determine who will do the merge
2. Notify interested parties that the merge will commence at some date/time. QA and developers working in the branch, for instance. SCM too
3. Execute the merge. Either Cherry Pick or Full merge.
4. Fix conflicts, if any
5. Build and create the executable.
6. Notify QA and others that you have a new air file to test. Provide the Jira IDs so QA will know what to look for.
7. QA gives it a once over
8. If QA approves, commit the changes to subversion.

Find the specific steps in the "how to" document elsewhere in this Qwiki.

Important Items

Get the **latest version** of Subversion and install it. Anything older than version 1.6 won't work with our repository. Later versions have more advanced merging facilities. I'm currently using version 1.6.15 of subversion.

Be careful what subversion docs you read! Google searches often find old documentation, which won't have all the features documented. In particular, the merging parts of the documentation have been radically updated to match the new versions. Don't reference the 1.5 version of the docs!

Don't do ad-hoc merges. Merges should be real "svn merge" and not ad-hoc. Subversion does a lot of work for us, and we need to make use of that. It saves a heck of a lot of time by automatically making the code changes where it can, and only flagging conflicts. Far more important is that Subversion helps keep track of what's been merged and what hasn't.

Take good notes and put them in the commit log message. It should specifically mention what was merged. It is imperative to add good comments to the checkin message when a merge is checked in.

Additional Notes:

Proper merging avoids a lot of time consuming problems and associated headaches

Merging from trunk to branches avoids duplicated effort and code conflicts

With parallel work going on in branches and trunk, we often need the bug fixes found in trunk copied to the branches. The branch represents the next release. Those working in that branch may run across a bug, and fix it. That same bug was already fixed in trunk. When the code is finally merged, there may be conflicts because the code the same places in the same file will be changed. That bug fix effort will now have been done three times. Without merges up from trunk, the branch will become further behind and less ready for release.

Frequent merging avoids duplicated effort. It saves time by avoiding code conflicts. It's fast. One recent full merge from trunk to a branch took about an hour and a half to complete.

Merging from branches to trunk avoids code loss

Parallel coding efforts in trunk and branches are done for different features and bug fixes, affecting a lot of the same code. Let's say a branch is created June 1, and coding commences. Only occasional, ad hoc merging is done during the sprint. At the end of the sprint, engineers decide they don't have time to do a proper merge with trunk. A decision is made to just clobber trunk with the branch, as the developers are "certain" all changes made in trunk migrated up to the branch. Trunk is clobbered, and replaced with the code from the branch. Trunk builds and runs fine because it's just the branch's code. A week later we discover some bugs that we thought were fixed, are broken again! How did that happen? Some of the code in trunk didn't get merged with the branch. We won't know what code didn't get merged to the branch! The developers will need to spend time walking thru history on the "old" trunk, and reincorporating it into the current trunk. This is time consuming, nerve wracking and tedious.

A proper merge to trunk ensures we won't miss any code. A merge down from the branch to trunk ensures there won't be missed code, since

Subversion prevents a merge down if that branch doesn't have all the latest code from trunk.

We merge up from trunk to keep our code stable, and current.

Merging up from trunk should be very low risk. Trunk should always be stable. Trunk represents the next release.

Subversion lets us know where we stand in the branch code.

Subversion keeps track of what's been merged into the branch code. This prevents you from merging the same changes twice. Merging without Subversion does not provide this assistance.

Subversion merges typically do the "right thing."

Just like doing a subversion checkout, a subversion merge will automatically merge the code for you. No manual diff necessary. Subversion makes the process much easier than using a diff tool to walk thru directories.

Merging a branch back to trunk closes the branch to further checkins.

The code is still there, along with the history, but the branch is blocked from further checkins. This saves confusion.

Reference

Subversion Best Practices

Subversion Docs

Apache Subversion Home