Source Control Practices

## Overview

Like most development activities, source control interactions can be inefficient, error prone, and cumbersome if not practiced by a disciplined team following a process. This document attempts to outline the best practices for source control integration with our development process.

## Interaction Paradigm Summary

The edit-merge-commit paradigm offers a low-friction approach to source control that helps to maximize developer productivity by allowing concurrent development operations and eliminating the need for a persistent connection and ongoing synchronization with the repository. Unlike the exclusive check-out paradigm, where a given file can be worked on by a single developer, the edit-merge-commit paradigm allows for parallel work streams to be performed concurrently against a given file.

There is a certain amount of risk inherent to concurrent development, mostly in the form of multiple changes to a single file requiring a merge to resolve. Most modern version control systems help to mitigate this risk by viewing merge operations as a core activity and providing solid tool support for them. The majority of merge operations can be safely and properly handled by automatic merging, requiring no further action on the part of the developer.

The edit-merge-commit paradigm accepts these risks in exchange for the reward of offering a development experience with fewer collisions between developers and less network chatter, making it a natural choice for a distributed development team.

## Workflow

The basic edit-merge-commit workflow is as follows:

1. Edit the working copy as needed.
2. Merge any recent changes from the repository into the working copy.
3. Commit the changes to the repository.

All files within a working directory are editable with no coordinated check-out required from the repository. A developer may simply open a file and begin to make the changes needed. When a logical unit of work is completed, the developer commits that change set to the repository. Most often, the local changes are accepted without issue. In the event that one or more files of the change set have been altered within the repository since the developer’s working copy was updated, the developer must ensure that the repository changes are incorporated into the change set before the commit can take place.

To incorporate the changes, the developer requests an update from the repository which will trigger a merge operation for each file that has changed in both the repository and the working copy. In the majority of cases, the changes are merged transparently by the source control client and no further actions are required by the developer. In the event that the changes could not be automatically merged, the developer is responsible for performing the merge using a differencing tool or accepting one of the file versions (repository or working copy) to be preserved in its entirety. Once the latest repository changes have been integrated into the working copy, the developer commits the change set.

## Development Practices

When working under the edit-merge-commit paradigm, it is important that developers follow a common set of practices to help facilitate minimal file contention and ensure that merge operations are minimal and simple. The following practices should be observed:

Update and Commit Frequently

Because development is concurrent, it is important for a developer to integrate the repository changes into his or her working copy and the local changes into the repository as frequently as it is reasonable to do so. The guidance is a minimum of once per day for each operation, but developers are encouraged to do so more frequently.

The underlying philosophy is to fail early should there be tasks overlapping or integration issues so that resolutions can be sought early in the development cycle instead of remaining hidden until just before a release. By attempting to continually integrate changes, it becomes much easier to identify and expose areas of contention. As a general rule, any resulting merges ought to be automatic and transparent as developers should be working on different parts of a given file, if their tasks overlap at all. If a file is experiencing high contention it indicates one or both of the following:

1. The class defined by the file does not have a single responsibility to perform or is not well organized. This is a warning sign that the class should be considered for refactoring.
2. Planning of features for the iteration was not ideal. Multiple developers have been assigned tasks which resulted in changes that are too tightly coupled. Either those items should be tasked to a single developer or those features should be spread across multiple iterations.

If no easy resolution is possible then the early discovery of contention will indicate to the developers that they need to coordinate to ensure that they are managing the contention as best as possible, including sharing duties if needed. For example, Developer A may handle all changes to File1 for multiple features while Developer B handles all changes to File2 for the same features.

Work in Logical Units

In support of frequent integration, development tasks should be organized into logical units of work, such that they can be committed to source control on a reasonable schedule without causing issue for other developers. Any change sets committed to the repository should result in a compilable codebase for those updating their working copies.

When working on a feature that requires new project/solution items, it is recommended that those items be created as empty skeletons and committed before they are developed, in order to minimize contention on the project/solution files.

Unless it is very small in scope and can be completed in less than a day, it is not recommended that developers attempt to complete an entire feature before committing to the repository. Instead, units of work should be comprised of a small subset of functionality for a given feature. Ideally the change set should also include the unit tests that support the functionality.

If a developer is unable to decompose a feature into smaller change sets, it may be a sign that the feature is large and complex enough that its development should be performed in a dedicated branch, which can be synchronized against the trunk on a frequent basis.

Write Meaningful Commit Messages

When committing a change set to the repository, always accompany it with a meaningful comment. The purpose of the comment is to describe what the logical unit of work was, and why the change was made. Comments should be brief and focused, with the intent of allowing an individual browsing the repository history to understand the lineage of commits.

Many developers are careless about commenting their changes, and some may feel that commit messages are not needed. Either they consider the changes trivial, or they argue that you can just inspect the revision history to see what was changed. However, the revision history only shows what was actually changed, not what the developer intended to do; nor does it explain why the change was made. This can be even more problematic when developers are not performing fine-grained commits. With a fine-grained revision history, comments can be useful to distinguish trivial from non-trivial changes in the repository.

Do Not Include Build Output in Source Control

Avoid adding build output and other items that are frequently regenerated; doing so adds a hotspot for contention. In many cases, such as with build output, the generated items are binary and cannot be merged. It is highly recommended that binary references for a project be tracked in such a way that the build output folders can be excluded from source control as a whole.

Certain other types of support files/directory should be omitted from source control as well. Please see the addendum on Excluded File Masks for details.

## Repository Structure and Workflow

Each project within the repository is segmented into branches, tags, and the trunk. Dividing a project in this way allows a well-known structure for managing application versioning and concurrent release cycles.

Trunk

The trunk represents the main body of development, containing the full history of the project from its origin to the current release, encompassing all changes in-between. The trunk’s main purpose is to serve as the primary workspace for the project. Development for the next major release is performed here, and all support bug fixes for existing releases are merged back to the trunk as completed.

Branches

A branch represents a discrete unit of work, derived from a snapshot of the code at a given time. Branches are typically used to provide a workspace either for the support of an existing release or feature development for a future release, and always created as a complete copy of the trunk, not a subset. Branches are temporary in nature, existing for the duration of the work being performed against them, ending with the branch either merged back to the trunk or deleted. A branch is created under the “branches” folder and is given a meaningful name at the time of creation. Branches typically fall into one of the two following categories:

A support branch is created when the current release hits a target milestone, most often deployment for quality assurance. The branch is created from the trunk revision that was built for the milestone. All bug code modifications, whether part of the QA cycle or bug fixes for production, are performed against the branch. Once the modifications have been tested and proven stable, the changes from the branch are merged back into the trunk. All builds and deployments for the release are performed against the support branch. A support branch will exist for as long as the revision that it was created for is actively being used and should be deleted when its revision is no longer in use.

A feature branch is created when development is needed that will not be included in the current release. The branch is created from the current revision of the trunk, with the understanding that code in the trunk will continue to evolve. Because the branch is dependent upon the trunk, changes from the trunk should be merged into the branch as often as possible. When the changes from the feature branch are ready to be included in the current revision, the feature branch should be merged into the trunk and deleted.

Tags

A tag represents the frozen snapshot of a production release. Tags are intended to be read-only and should never be used as a working revision. Each production deployment, including all support hot-fixes should result in the creation of a new tag. Unlike branches, tags are permanent and should never be deleted. Tags should be named after the version of the application that they represent.

Standard Workflow Example

Assume a new project named “ProjectOne” is needed by the business. The new project is added to source control, creating the following structure in the repository:

* $ (root)
* svn
  + Main
  + **Project One**
    - trunk
    - branches
    - tags

Development for the first release is performed, using the trunk as the workspace. Once development on the first release is completed and it is released to QA, a branch is created for the QA cycle, allowing development for the second release to be started using the trunk as its workspace. At this point, the repository looks like:

* $ (root)
* svn
  + Main
  + **Project One**
    - trunk
    - branches
      * + qa-1.0
    - tags

All QA fixes for the first release are performed on the QA branch. The changes from the QA branch should be merged back to the trunk as often as reasonable. This ensures that all fixes and tweaks made for the first release are integrated into the second release. By merging often, integration issues are exposed early in the development cycle, allowing the development of the second release to proceed with minimal risk of hidden contentions.

Once the QA cycle is complete, the first release is built from the QA branch and deployed to production. After the deployment has been verified, a tag is created for the first release. The QA branch is renamed to denote that it is now the current production support branch. The repository looks like:

* $ (root)
* svn
  + Main
  + **Project One**
    - trunk
    - branches
      * + production-support
    - tags
      * + v1.0

Development progresses for the second release, following the same workflow. Once development on the second release is completed and it is released to QA, a branch is created for the QA cycle, allowing development for the third release to be started using the trunk as its workspace. At this point, the repository looks like:

* $ (root)
* svn
  + Main
  + **Project One**
    - trunk
    - branches
      * + production-support
        + qa-2.0
    - tags
      * + v1.0

The QA cycle progresses as it did for the first release, with QA fixes are performed on the QA branch and the changes from the QA branch be merged back to the trunk as often as reasonable. When the QA cycle for the second release is complete, the second release is built from the QA branch and deployed to production. After the deployment has been verified, a tag is created. The existing production support branch is deleted and the QA branch is renamed to denote that it is now the current production support branch. Development for the third release continues, following the same workflow. The repository looks like:

* $ (root)
* svn
  + Main
  + **Project One**
    - trunk
    - branches
      * + production-support
      * tags
        + v1.0
        + v2.0

At any point after the initial release, any problems that arise in production, bug fixes should be performed on the production support branch. All changes made to the production support branch should be merged back to the trunk to integrate the fixes. If there is an active QA branch for a pending release, the changes from the production support branch should also be merged to the active QA branch, so that the fixes are integrated with the pending release.

The build for any production hot-fixes should be performed against the production support branch and then deployed. After the deployment has been verified, a tag is created. Assuming one production hot-fix after the second release, the repository would look like:

* $ (root)
* svn
  + Main
  + **Project One**
    - trunk
    - branches
      * + production-support
      * tags
        + v1.0
        + v2.0
        + v2.1

Multiple Concurrent Releases Workflow Example

Assume a new project named “ProjectTwo” is needed by the business. The new project is added to source control, and undergoes its first release using the standard workflow. The current state of the repository looks like:

* $ (root)
* svn
  + Main
  + **Project Two**
    - trunk
    - branches
      * + production-support
    - tags
      * + v1.0

Two large features need to be implemented, title them “LargeOne” and “LargeTwo”. The first feature is going to take two release cycles to complete. The second will take three. While the large features are under construction, another release with a set of smaller features will be developed in parallel and released after one release cycle.

Because the three releases are being developed concurrently, the earliest scheduled release will be developed against the trunk, and the latter two will be performed in release branches. The name of a release branch should be a meaningful identifier which describes the release and is prefixed by an ordinal that corresponds to the order that it will be released relative to other releases in active development. It is assumed that the trunk is always the current pending release. After creating the branches, the repository will look like:

* $ (root)
* svn
  + Main
  + **Project Two**
    - trunk
    - branches
      * + production-support
        + 2-large-one
        + 3-large-two
    - tags
      * + v1.0

Each branch in this model represents a scheduled future release. The order of those releases is pre-determined. It is important to understand the order that the releases will be performed, because it will allow the dependencies to be tracked. For example, when the large feature release is deployed to production, it will be expected that it also incorporates the changes made by the smaller features release.

In order to pay integration costs early in the cycle and to keep the merge difficulty and cost as low as possible, a release branch should be merged with the code for the release that precedes it as often as reasonable. In the context of this example, the branch for release “LargeOne” will merge forward changes from the trunk. The branch for release “LargeTwo” will merge forward changes from the branch “2-large-one.”

Once the current release enters its QA cycle and is moved to a QA-focused branch, the branch that represents the next release is merged into the trunk, becoming the current release. Any outstanding release branches are renamed to reflect their current release order.

After the first release cycle is complete, and the set of small changes has been deployed to production, the repository will look like:

* $ (root)
* svn
  + Main
  + **Project Two**
    - trunk
    - branches
      * + production-support
        + 2-large-two
    - tags
      * + v1.0
        + v2.0

The next release cycle continues following the same pattern. When the current release (“LargeOne”) begins its QA cycle, the release branch “2-large-two” is merged back into the trunk and becomes the current pending release. Once this occurs, the workflow mirrors the simple workflow example once again. At this point, the repository looks like:

* $ (root)
* svn
  + Main
  + **Project Two**
    - trunk
    - branches
      * + production-support
        + qa-3.0
    - tags
      * + v1.0
        + v2.0

Workflow Summary

This workflow model compliments the outlined development practices well, placing an emphasis on paying integration costs incrementally to ensure that issues are exposed and can be addressed early. Doing so helps to ensure that complicated merges late in the development cycle can be avoided.

This model is ideal for an environment where the development focus is centered on one or two releases at a given time. While the model does scale to support multiple concurrent releases, the emphasis on merging and renaming after each release does result in higher costs as the number of concurrent work streams increase.

It is also worth mentioning that this workflow model will break down in the event that there are many large features being developed concurrently which are not, or cannot, be scheduled up-front as releases. Should this prove to be the case, then alternative source control workflows should be considered.

## Project Taxonomy

The source control tree for each project should provide support for not only the project’s code, but also for the miscellaneous assets that support the project. It is recommended that all projects adopt the same taxonomy and extend it if the project needs dictate doing so. The recommended taxonomy is:

* $ (root)
* svn
  + Main
  + **ProjectName**
    - trunk
  + build
  + component
    - * v1.0
      * v2.0
    - src
      * **ProjectName**
      * sandbox
        + developer-1
        + developer-2
    - documents
    - sql-scripts

**ProjectName ($/svn/MainProjectName/trunk) –** Thecontaining folder for mainline development of the project and its various support assets. All code, documents, scripts, resources, and assets are contained herein.

**build ($/svn/MainProjectName/trunk/build) –** The containing folder for any build-related resources. This folder will contain subfolders for the different environments that the solutions can be deployed to. Examples of content here are build scripts and environment-specific configuration files.

**component ($/svn/MainProjectName/trunk/component) –** Intended for use only by projects that are reusable components, this folder serves as the container for published builds of the component. All published builds should be organized as subfolders named after the version of the build. This is to allow for consumers to selectively upgrade. The general guidance is to keep all public releases going back 2-3 versions at minimum. This allows consumers to adopt newer versions with breaking changes on their schedule. This folder will not be present for projects that are not intended as reusable components for public consumption.

**src ($/svn/MainProjectName/trunk/src) –** The containing folder for code assets. This folder contains subfolders for each independent solution as well as a folder for developer sandboxes.

**ProjectName ($/svn/MainProjectName/trunk/src/ProjectName) –** This folder contains the primary solution for the project, defining the application/component/services and associated support libraries. Should other solutions be needed for the project, they should be added as a sibling, not as a subfolder.

**sandbox ($/svn/MainProjectName/trunk/src/sandbox) –** The containing folder for the developer sandbox area. This folder contains subfolders for each developer. This area allows the developers to house any prototype projects or similar scratch work related to the project that would benefit from source control.

**documents ($/svn/MainProjectName/trunk/documents) –** The containing folder for any project-related documents. This folder may contain subfolders for organization purposes and may also contain documents directly.

**sql-scripts ($/svn/MainProjectName/trunk/sql-scripts) –** This folder contains any SQL scripts used to manage the projects data stores. If more than a single data store is used, this folder should contain subfolders corresponding to each data store, which the scripts in their appropriate containing folder.

## Additional Resources (Subversion Focused)

***Version Control with Subversion***

<http://svnbook.red-bean.com>

A comprehensive book on Subversion use and concepts, both basic and advanced.

***CollabNet's Best Practice Recommendations***

<http://svn.collab.net/repos/svn/trunk/doc/user/svn-best-practices.html>

Recommended best practices by CollabNet, the corporate sponsor of Subversion. This guidance is oft-referred to by the Subversion community.

***Subversion branching quick start***

<http://nedbatchelder.com/text/quicksvnbranch.html>

Some good information on branching and parallel development strategies within Subversion.

***Source Control HOWTO - Chapter 3: File Merge***

<http://www.ericsink.com/scm/scm_file_merge.html>

The third chapter of Eric Sink's source control overview, providing information on merge operations.

***Streamed Lines: Branching Patterns for Parallel Software Development***

<http://www.cmcrossroads.com/bradapp/acme/branching/>

A deeply detailed discussion of branching patterns for concurrent software development. Very dry, very thorough.

## Addendum: Excluded File Masks

Directories and files matching the following patterns should be excluded from source control. If possible, please configure your client with a rule to exclude them.

|  |  |  |
| --- | --- | --- |
| [Bb]in  [Oo]bj  [Tt]mp  [Pp]kg  [Cc]vs  [Dd]ebug  [Rr]elease  [tT]emp  [tT]empPE  [Tt]estResults  temp  \_private  \_vti\_bin  \_vti\_cnf  \_vti\_log  \_vti\_pvt  \_vti\_script  \_vti\_txt | \*.bak  \*.obj  \*.ncb  \*.lib  \*.idb  \*.pdb  \*.ilk  \*.msi  \*.pch  \*.suo  \*.aps  \*.user  \*.suo  \*.webinfo  \*.pdb  \*.\*scc  \*-results.xml  \*.incr | \*.scc  \*.SCC  \*.vspscc  \*.projdata  \*.vbw  ~vssc  \*[tT]humbs.db  \*.log  \*.tmp  \*.temp  ~$\*  ~$\*.\*  \*.\*~  \*.~\*  ~\*.\*  Ankh.\*  [Dd]esktop.ini |