Overview

The organizing system I am describing is distributed source control management more commonly known as version control. Distributed source control management is a process by which companies and individuals manage documents (and versions of documents) and share them among various agents including contributors, human and non-human testing agents, and machines.

This organizing system can be anything from maintaining copies of files and manually naming them “Draft 1”, “Draft 2”, “Draft 3”; having one file maintain all of its own changes through a system like SCCS¹; tracking changes to files by having a system save complete copies at save time; or saving only the changes that are made (down to the bit level) as is the case with git.² This case study will focus specifically on the git source control management system.

Below I introduce some terminology as well as three separate models or workflows for projects with multiple contributors.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repository</td>
<td>An abstraction representing the location of code either on a developer’s computer or on a server.</td>
</tr>
<tr>
<td>Shared Repository</td>
<td>A server based repository where multiple developers can push and pull changes without having to explicitly clear those changes with others.</td>
</tr>
<tr>
<td>Blessed Repository</td>
<td>A server based repository where contributors must have their code explicitly checked and only a few developers can contribute changes without those changes requiring inspection by others.</td>
</tr>
<tr>
<td>Developer</td>
<td>An individual who makes changes to a repository.</td>
</tr>
<tr>
<td>Developer Public</td>
<td>The public repository of an individual developer.</td>
</tr>
<tr>
<td>Developer Private</td>
<td>The private repository of an individual developer.</td>
</tr>
<tr>
<td>Integration Manager</td>
<td>An individual who manages the integration of changes from groups of developers into a blessed repository.</td>
</tr>
<tr>
<td>Lieutenant</td>
<td>An integration manager who manages the integration of changes from bottom-level developers but cannot push changes into a blessed repository prior to review by a dictator.</td>
</tr>
<tr>
<td>Dictator</td>
<td>An integration manager who manages the integration of changes coming from lieutenants.</td>
</tr>
<tr>
<td>Pull Request</td>
<td>A set of changes that is created by a developer and sent to the integration manager for review prior to merging into the blessed repository.</td>
</tr>
</tbody>
</table>

¹ The Development and Management in Unix-Based Projects, Robert J. Glushko.  
² http://git-scm.com/about/info-assurance
Arguably the most simple is the centralized workflow example.

Typically every contributor to a shared repository will have their own locally stored repository to which they can make and commit changes prior to pushing them to the central repository.

Another common workflow is the integration manager or simple hierarchical workflow.

There is one blessed repository and any number of private and public developer repositories. These repositories are updated by developers who then create pull requests for the integration manager to review prior to merging into the blessed repository.

The third repository is the most structured of those outlined and has a nested hierarchical system.
There is one blessed repository controlled by a dictator who receives pull requests from a limited set of lieutenants who in turn receive pull requests from outside developers.

Although there are many meta-organizing systems, or workflows, that can vary greatly across organizations and projects, the above workflows are the most common and the ones I will use as examples throughout this case study.

Image Credit: [http://git-scm.com/about/distributed](http://git-scm.com/about/distributed)

**What is Being Organized?**

The fundamental resources being organized are sets of changes to a document collection as well as the metadata associated with those changes. This metadata includes the individual those changes, the message associated with those changes written by the developer, the date and time the changes are committed, and who approves those changes. A commit is cryptographically secured so that changes cannot be made to a file or a commit message once changes have been committed (outside of a new commit). Each commit also receives a unique identifier, or hash, which can be referenced in the future in order for someone to checkout, or jump back to, the repository as it was at a specific point in time.

It is important to note that the git source control system does not make any structural changes to files or folders and tracks changes in a separate location. It monitors bit level changes to document resources in the collection and records them when an agent finalizes a set of changes through a commit. This is an important detail because changes in documents are not tracked in real time but rather at specified commit times.

Resources in the collection are also organized through branching and forking, processes described in the How much is being organized section of this case study.

**Why is it being organized?**

Writing computer programs is a difficult exercise for both humans and computers. Managing versions across time, location, and contributor can be challenging for organizations and projects. The git version control system resolves these challenges through key features like traceability, maintainability, governance, and easy distribution of the repository. Monitoring bit-level changes allows for simple traceability for changes made down to the original line, author, date and commit message.

---

3 [http://git-scm.com/about/info-assurance](http://git-scm.com/about/info-assurance)
The organizing system is essentially a historical record of everything that has changed in a set of files or folders and explanations of why those things have changed. It is important that the changes are recorded alongside the developer’s commit messages as opposed to the messages being tracked outside of the organizing system. This was shown by Dr. Robert J. Glushko to be nearly impossible to scale effectively.4

By organizing files and messages at commit time, it is simple to store the historical record of the repository and how and when decisions are made because each one is explicitly confirmed by the developer. Another benefit is that of simple governance and control structures; agents can review changes before integrating them into a blessed repository. Another benefit of explicit organizing times is the simplicity by which experimentation and feature development can occur while still maintaining a core, stable codebase.

**How much is it being organized?**

One of the remarkable features of this organizing system is its flexibility in organizing the historical record of individual lines in specific files to hundreds of files of remarkable size. The versions of the files can be nearly limitless as well. One particularly remarkable example is the GNU Compiler collection, a git repository that has, at the time of this writing, nearly 135,000 commits by over 500 people and nearly 7 million lines of code in 34 languages with the first commit in November of 1988.

In addition to the sheer scope of the files being organized, the granularity by which they are organized is astonishing. As mentioned above, changes to files are tracked at the bit level ensuring that, as changes are made, the minimum necessary amount is tracked historically so as to save space and speed up reviewing of historical changes. While granular historical code level organization is another aspect of the granular capabilities of the organizing system. Code can be organized into different branches to allow different agents to participate in development concurrently.

These branches are, at creation time, exact copies of the documents in the collection. Branches can be made locally(on the contributors computer) or remotely(on a server or service like Github). A branch can also have any number of commits, any number of sub-branches, and can be merged into other branches. Each one of these features must be performed with a commit, in order to ensure the ability to return to that state prior to that event. Another benefit to branching is that a developer can then make her own changes, whether they will be made permanent or not, to a copy of the repository without changing the blessed or shared repository.

Forking is a similar concept to merging that is primarily used in the integration manager and dictator workflows. Each developer has their own remote developer public repository and pushes changes to their own fork of the original repository. Those changes can then be integrated into the blessed repository through a pull request, which in turn gets a commit message.

The git version control system is well prepared to support any scope and scale of code resources in extremely granular ways both for a historical record and current development. This organizing system coordinates mass amounts of information to be tracked and allows for many contributors to add to large-scale repositories while still maintaining a strict organization hierarchy and historical record.

**When is it being organized?**

The resources are organized at certain points in time in response to actions taken by a contributor. The most common examples of those times are:

- fork time
  - The fork is an exact copy(at fork time) of the original repository.

---

• **branch time**
  ◦ The branch is an exact copy (at branch time) of the original branch.

• **commit time**
  ◦ When changes are committed to a branch a hash is created to uniquely identify these changes.

• **merge time**
  ◦ When a contributor merges their changes into a branch, the historical changes that have been made to the code are saved by the organizing system.

• **push time (centralized workflow)**
  ◦ When a contributor pushes their changes to the remote repository the git system records these changes and other contributors can see that these changes were made.

• **pull time (integration manager or nested hierarchy workflows)**
  ◦ When a contributor requests that a governing agent merges their fork into the primary codebase.

At each of these points in time, the git system organizes the resources into the historical record. In between these specific events (like after a repository is forked but before it is committed to), the code is organized completely by the developer writing the code or the workflow that they are following. However once her changes are finalized with a commit, the git system organizes the information. This is explained in detail in the next section.

**How or by whom is it being organized?**

Fundamentally there are two organizing systems at work, there is the coordination between the agents accessing and modifying the files in the organizing system. Secondly there is the system by which this information, and metadata, is stored in a historical record. The complexity of the historical record is managed by git which provides a much more simple interface for humans or other agents to monitor progress and track changes. However the coordination between agents contributing to a shared repository is a much richer, less mechanical, system and is elaborated in the artifact included with this case study.

In short, an increase in the number of developers contributing to a project introduces large amounts of complexity. Therefore larger projects, or important projects, need to have strict systems of controls concerning contributions and how changes are tracked and validated. The relationships between the developers is a reflection of the project itself as well as the number of users (and types of users) of the system. The git organizing system provides a concrete, scalable organizing system to organize code (through historical commit points with explicit change sets) and contributors (through workflows, forks, and branching).
Artifact
This artifact illustrates implicit and explicit relationships in the git organizing system.

In the above workflow (centralized workflow) all developers have the same explicit authority in the system. This system reflects an implicit trust between contributors that changes will not be made that will damage the system. The relationships in this system are based on implicit trust between developers. This workflow is common for small projects where speed is prioritized over stability.

In this workflow, a contributor will typically organize her branches on her local machine and push her permanent changes to the shared repository without requesting permission to do so through the organizing system.

In the integration manager workflow a dedicated individual (or group of individuals) have authority over what goes into the blessed repository. This system explicitly defines a single level hierarchical relationship structure where certain individuals become responsible for permanent code changes. The implicit trusting relationship between contributors in the previous workflow is now explicit and structured, defined by roles rather than implicit relationships. This system is common for projects with more contributors, that serve many users, or for companies wanting to ensure a level of code quality.
In this workflow, a developer will typically organize his branches on his local machine then push them to his own forked public repository. He will then create a pull request asking for his fork to be merged into the blessed repository by the integration manager.

Finally in the nested hierarchical workflow one individual (or small group of individuals) has complete authority over the blessed repository. This same system cascades down to specific parts of the codebase where individuals have complete authority as to what changes are integrated into their respective sections and eventually considered by the dictator.

This workflow is similar to the integration manager except that the dictator, or top-level contributor, is not sent requests for integration that have not been vetted by a lieutenant. The relationships between contributors are completely explicit and the system adheres to a strictly defined role hierarchy. The implicit trust from the centralized workflow has been replaced by several layers of review (and an implicit lack of trust). This system is common for very large projects or for extremely mission critical systems as code quality and stability are prioritized over speed.

In this workflow, a contributor will typically organize his branches on his local machine then push them to his own fork. He will then create a pull request asking for his fork to be reviewed by a lieutenant who will approve, edit, or deny the request and pass that along to the dictator who will make the final decision as to whether the code will be integrated into the blessed repository.

Image Credits: [http://git-scm.com/about/distributed](http://git-scm.com/about/distributed)