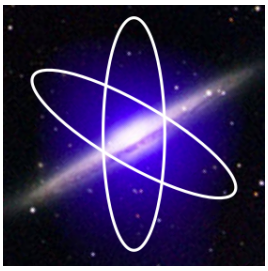


ATSAL and Git

by Thomas H. Kent

ATSAL rests atop a great deal of independently developed software. Git, a distributed version control system (DVCS), provides a mechanism both for tracking development of ATSAL and for controlled merging of externally developed software updates. Git is well-suited to this task but fairly challenging to master. This document describes how to use Git in the context of the ATSAL project.

1 Contents

1 CONTENTS **2**
2 INTRODUCTION **4**
3 GIT AND VIRTUAL MACHINES **5**
4 QUICK GIT GLOSSARY **5**
5 GIT TOPOLOGY **7**
6 ADMINISTERING A SERVER **9**
6.1 INSTALLING GIT ON A SERVER 9
6.1.1 Gitolite Setup *9*
6.1.2 Git Administration *11*
6.1.3 Managing Git Users with Gitolite *13*
6.1.4 Cloning a Repository From a Gitolite-managed System *14*
6.1.5 Backing up the Repository *14*
6.1.6 Changing the Origin for Everyone *16*
6.2 SETTING UP THE ARCHIVE FILE TREE 16
6.3 ARCHIVES VS. BARE ARCHIVES 17
7 ARCHIVE STRUCTURE **20**
7.1 DIRECTORY STRUCTURE 20
8 SETTING UP A DEVELOPER SYSTEM **23**
8.1 USING A PRECONFIGURED VIRTUAL MACHINE 23
8.2 BASIC SETUP 24
8.2.1 Installing Git *24*
8.2.2 Establishing an Identity *24*
8.2.3 Keeping it Simple *24*
8.2.4 Related Tools *24*
8.2.5 Git Administration ..... ***Error! Bookmark not defined.***
8.3 THE .GITIGNORE FILE 24
8.4 XCODE'S GIT SUPPORT 25
8.5 WORKFLOW 26
8.6 CLONING THE ATSAL TREE 26
8.6.1 Cloning the Entire Tree *27*
8.6.2 Cloning Individual Submodules *27*
8.7 LEARNING GIT WITHOUT SHOOTING YOURSELF IN THE FOOT 28
8.8 DOING A BUILD 29
8.8.1 XMLRPC *29*
8.8.2 XSpec Server *29*
8.8.3 XClient (test bed) *29*
8.8.4 ATSAL Python Library *29*
8.8.5 Running the Library *29*
9 MORE ON GIT **30**
9.1 GIT TAGS 30
9.2 GIT BRANCHES 30
9.2.1 HEADs, and the Detaching Thereof *33*
9.2.2 ATSAL Branches *34*
9.3 GIT SUBMODULES 37

9.3.1 Branching in Submodules 38

10 GIT CHEAT SHEET 40

11 APPENDIX 1: .GITIGNORE FOR XCODE PROJECT FILES 42

2 Introduction

Git is an open source Version Control System (VCS). We use it for ATSAL as well as several related projects.

See Scott Chacon's book *Pro Git,* a free download:

http://www.git-scm.com/book

Pros:

- Open source
- Supported on all platforms
- Wide user base
- Keeps a copy of the archive on each local system, resulting in much faster operation, ability to work without access to a server, and built-in redundancy. (Such systems are called Distributed Version Control Systems, or DVCSs.)
- Offers a fast, lightweight branching mechanism, allowing branching to be used in new ways relative to other systems (key feature)
- Allows users to clean up local work before pushing changes to other server(s), reducing server clutter (key feature)
- Supports methods of use for projects of widely varying degrees of complexity, including those with large numbers of collaborators
- Tracks files of any type
- Uses hash values that, among other things, reliably detect file corruption before it propagates to other copies of the repository (`git fsck --help`)
- Makes it fairly difficult for developers to shoot themselves in the foot
- Neutral about use of related tools such as diff tools or editors
- Reasonably easy to configure to allow developers read/write access and other users readonly access
- Git can import from Subversion

Cons:

- Keeps a copy of the entire archive (all history) on each local system, which can impose a high space impact for casual contributors
- Maintains all versions as discrete files (vs. deltas as in most other VCSs), increasing space consumption
- Using submodules and branches effectively is confusing

This document describes how we use Git for ATSAL and several related components, and the setup necessary to support it. The usage pattern is designed initially to support an anticipated model as follows:

- About 4-8 active developers (possibly working on unrelated subprojects), with read/write access.
- Eventual open source archive with readonly access.
- Organization as independent subprojects (e.g. AtomDB, XSPEC server, AtomDB for iPad, ATSAL).
- Periodic packaging of self-contained releases, in source form for Linux as well as source and binary format for Mac OS X.

3 Git and Virtual Machines

Git allows you to go back in time and restore the source file tree at some previous point. But there is much that Git does not capture about the past:

- The host operating system release may have changed.
- Third party tools and libraries that are not modified as part of our own development process are stored in Git as tarballs, not installed source trees. It takes a couple of days to rebuild and install these tools, and reverting to older versions of tools may be difficult.
- Git does not capture applications that assist with development, such as XCode or BBEdit.

To perform a bug fix for an older release, it may become necessary to regress some or all of these components. **Hence for each major release, we snapshot the virtual machine in order to capture all prior development state**. Virtual machines are not just a convenience when getting a new developer started, or a quicker way to transfer development from one physical machine to another; they are also a key part of the archiving strategy.

4 Quick Git Glossary

add	Adds a new file to the staging area in preparation for adding to the repository, or adds a modified file to the staging area. If the file was previously staged, the new add overwrites the previous file.
bare archive	A Git archive is stored in an invisible file, `.git`, in the root directory of a project. A "bare archive," created with `git clone --bare <non-bare archive>` or with `git init --bare`, contains the archive itself, without any checked out files. Bare archives serve as shared repositories or as backups.
branch	A separate chain of commits that branches from another branch. Switch branches with `git checkout`; list branches with `git branch`.
checkout	Selects a new branch in the local archive as the current default, and *replaces* files in the project's directory tree with those from the specified branch.
clone	Copies a Git archive. The copy may be a bare archive or a full archive. It includes only the specified module by default, but `git clone` offers a switch for recursive cloning.

commit	Places all staged files (and only staged files by default) into the current or specified branch in the local archive.
detached head	Normally the HEAD points to the tip of the current branch, advancing automatically as commits are performed. Some operations leave the HEAD disconnected from the tip, pointing to a specific version but with no defined place to commit to. You are pointing to a node in the tree, but haven't defined a branch.
fetch	Updates the local archive to contain files from the origin. Does not change your current work environment.
.gitignore file	A file listing all files that Git should not include in the repository. Files that are regenerated as part of the build process should not be included in archives. Important: the `.gitignore` file is different for each submodule.
.gitmodules file	Exists only if a project has submodules. Lists the submodules and where to get them. Created by `git submodule add`. URLs can be edited by hand later if needed.
gitolite	Utility for managing fine-grained user access to a set of Git archives.
HEAD	Shorthand for the tip of the current branch. HEAD^ is shorthand for "parent of HEAD": `git checkout HEAD^` means "back up to the files one version prior."
merge	Accepts files from another branch and merges them into the currently selected branch. (See also *rebasing*.)
mirror	Creates an efficiently updated copy of a Git archive as a backup.
origin	The default name of the Git archive from which the local archive was cloned, and which serves as the default for pushes and pulls. In our configuration, the origin is the shared repository.
rebasing	A different type of *merging* that consolidates history from a branch into a single atomic checkin. The end result matches that of a merge, but the commit history is more comprehensible.
pull	A *fetch* followed by a *merge*.
push	Pushes the current branch to the default tracking branch, or to a specified branch. Fails if someone else has pushed in the meantime; you need to pull their changes and integrate them first.
rm	Removes a file from the staging area (`git rm --cached`) or from the repository (`git rm`). The local copy is not disturbed.
SHA-1 hash	A 40-character hexadecimal string, essentially a checksum, that Git calculates as a unique identifier for a file in the repository, or for a particular version in the tree. The hash is used to verify file integrity, or to compare files efficiently. The first few characters of hashes are used as alternate names for versions otherwise unnamed.
staging area	A temporary holding area for copies of files awaiting a commit. The temporary area makes it easier to verify that the correct files are assembled prior to actually committing them. `git add` adds a file to this area; `git rm --cached` removes it. Only staged files are committed by default.

stash	A way to temporarily set aside work in progress without actually checking it in, to allow work on a different branch.
submodule	A completely independent Git archive that is referenced by the owning project only as a particular version and a URL. Submodules are not interlinked, so changes to a submodule made by others are not integrated into your own tree except when you decide to.
tag	A character string, typically a version number, that serves as a name for a particular version.

5 Git Topology

Git permits several topologies appropriate for projects of varying degrees of complexity. Because the early stages of ATSAL development involve a relatively small number of contributors, a simple arrangement is most suitable, though it may become more complex over time.

The master Git server is presently located on simone.cfa.harvard.edu, a HEAD-managed machine on the "DMZ," that is, isolated from other HEAD systems. We use a utility called gitolite to allow multiple project members to share access to a single shared git account, called `atsal_git`.

There are several protocols available for Git access. Developers will access Git via SSH, so that the OS and well-established protocols ensure security, data integrity, and validation. The easiest way to share a Git archive with open source users is to place the master archive in an HTTP file sharing tree and make a simple modification that allows Git to access it. HTTP is inefficient relative to other protocols, but it requires very little administration and is adequate for modest volumes. (Public access will not occur until the source tree is more mature.)

The server machine is backed up incrementally as one protection against data loss. Since each developer also has a copy of the archive, data loss is highly unlikely.

Initially, a developer clones an archive from the server to begin work. The local archives resides within the root directory of the project. So, for example, if `~/atsal` contains the ATSAL project files, `~/atsal/.git` contains the local archive. (Some VCSs have hidden file(s) at each level in a directory hierarchy. Git relies on a single hidden directory at the project root. Discarding this one file discards the entire Git archive.)[1]

The initial clone downloads the entire repository. The developer then works on the local machine, doing commits to the local archive, creating branches, merging branches back into the archive, for as long as necessary. When the code reaches a stable point or a subtask is completed, the developer typically merges any branches into their work branch, then pushes the result to the shared server. Thus it is possible to excise some of the detailed edit and branching history prior to commits to the shared server, so that the server contains less clutter and stabler releases.

[1] Well, one file *per submodule*…

An attempt to push changes to the server fails if any other developer has pushed changes to any files in the same branch in the meantime. In such cases the developer must pull the changes to the local machine, merge them, test the result, and push the merged code to the server. This differs from some VCSs, in which merges happen on the server, and merge errors can disrupt progress for multiple developers at the same time.

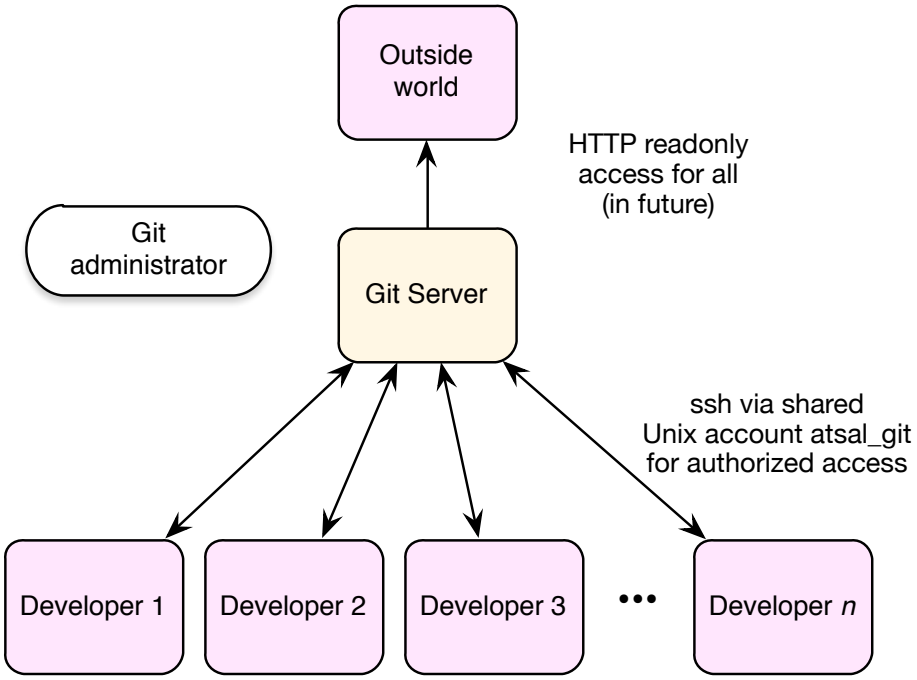


Figure 1. The Git administrator controls fine-grained access via gitolite. The repository is placed in the HTTP server's directory tree, making it available for readonly checkouts via HTTP protocol. The repository is backed up via an incremental backup utility. Server load is modest in typical use.

6 Administering a Server

This section is only for sysadmins and Git administrators. The sysadmin sets up a single shared user account to host an archive and installs needed tools. The Git admin is a project member who authorizes user access to specific archives.

We use a utility called gitolite, http://gitolite.com/gitolite/, to manage access to the Git server. This works by creating a single user account, `atsal_git`, which allows ssh access, but *only to the Git server*. Gitolite controls access by individual users to specific Git archives. The `atsal_git` account does not have special privileges. It uses public key access to connect. The public key is created with a passphrase that is managed by the host OS to avoid the need to enter multiple passwords when using Git.

The only time elevated privilege is required is when transferring an existing Git archive from another location. The transferred archive's files must be wholly owned by, and writable by, the `atsal_git` account. For some reason, this does not happen by default.

A computer stolen from a project participant, if its password is cracked, can only be used to check into Git. It cannot login to CfA computers. Access from the stolen computer is easily blocked using gitolite.

6.1 Installing Git on a Server

Git is pre-installed on Linux systems (as far as I know).

On a Macintosh, install the Apple XCode Development package, including the optional command line tools.

6.1.1 Gitolite Setup

These steps are performed once by a sysadmin, when setting up a new Git archive for use with gitolite. First, review the gitolite docs:

http://gitolite.com/gitolite/

On the **client machine**, generate a key pair for ssh. This user will serve as the initial Git administrator.

```
cd ~/.ssh
ssh-keygen -t rsa
```

This prompts for a key name, id_rsa by default. Override the default and enter your username instead.

Enter a very secure passphrase. You won't have to enter this passphrase often, so there is no excuse not to create a secure passphrase.

The result is two files, `username` (private key) and `username.pub` (public key).

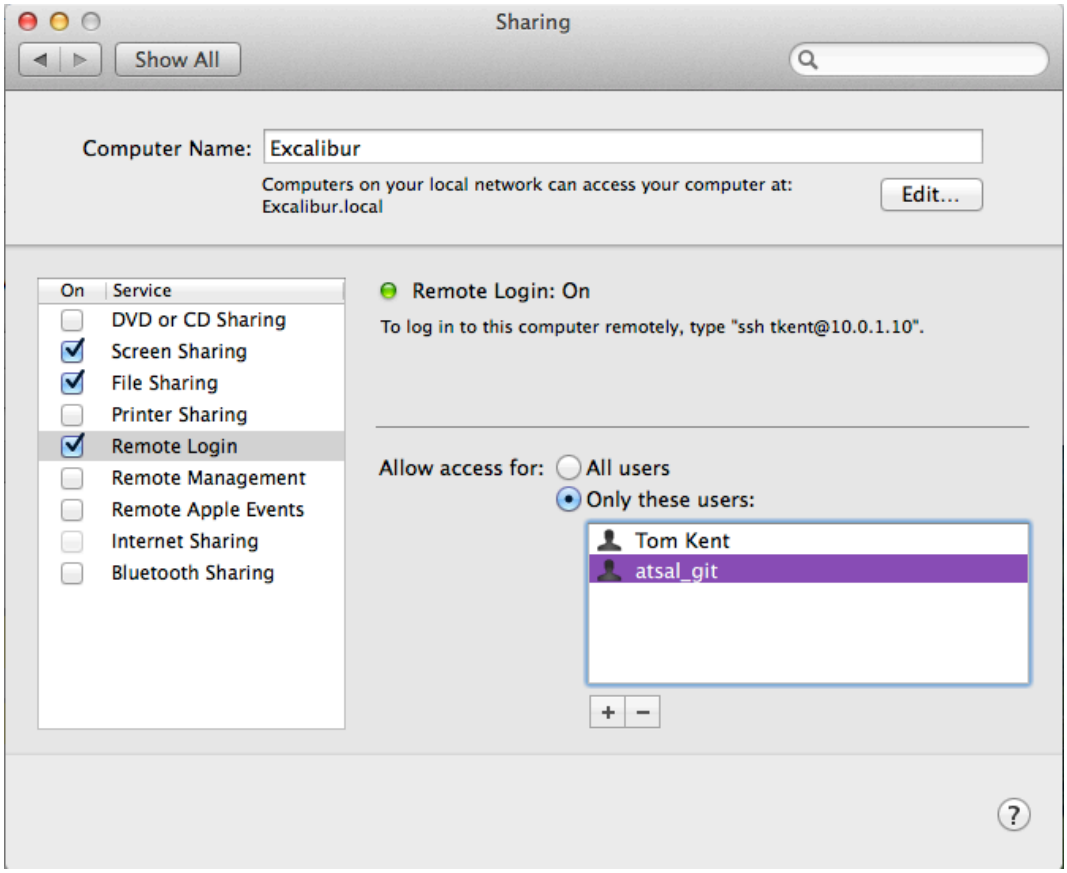
Add it to your keychain on the client machine:

```
ssh-add username
```

This will prompt for the pass phrase.

On the **server machine**, create a new user account called `atsal_git`.

Enable remote logins for the account. If the server is a Mac:



If the server is a Linux machine, the sysadmin may need to enable public key authentication on the server for the `atsal_git` account.

Login as `atsal_git`. Copy the public key somehow from the client to the server. For example:

```
scp username@client.machine:username.pub .
```

Create a `bin` directory. The gitolite command will be installed here.

```
mkdir ~/bin
```

Add this directory to the PATH environment variable in your preferred shell initialization file. On a Mac, this is `.bash_profile` by default. You will have to log in again to pick up the new PATH variable.

Create a repositories directory. Gitolite is hard-coded to look for repositories in this location.

```
mkdir ~/repositories
```

Get the software and install it:

```
git clone git://github.com/sitaramc/gitolite
gitolite/install -ln
gitolite setup -pk ~/username.pub
```

The `git setup` step above copies the Git administrator's public key into the `~/.ssh/authorized_keys` file, so you can delete the public key now.

```
rm ~/username.pub
```

The `authorized_keys` file now looks something like this. Even Git administrators cannot ssh into this account. They are restricted to Git. Only a sysadmin can configure normal interactive ssh access to this account.

```
# gitolite start
command="/Users/atsal_git/gitolite/src/gitolite-shell tkent",no-port-
forwarding,no-X11-forwarding,no-agent-forwarding,no-pty ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQDC4dY1Zj56JtZcgiy3RjE3aFkiH38DabPW8DabPW8Da
.....
.....
.....
.....
.....
P6+/TTVVRcEikQIUBVq1cO1+ekFkO6XT+jm82i98VRbZ5Spg/iFhLbbW+9QwNn9MJKZuifpw7
5iBLxb tkent@ATSAL.local
# gitolite end
```

Ensure that the protection on this file and `~/.ssh` is set properly. If these protections are wrong, `ssh` refuses to allow public key access. (This constraint may be specific to Mac OS X.)

```
cd ~
chmod 700 .ssh
cd .ssh
chmod 600 authorized_keys
```

It is also necessary to block write access to non-owners of the home directory:

```
chmod 755 ~
```

6.1.1.1 Adding Existing Repositories

Transfer any existing repositories to this location:

```
cd ~/repositories
git clone --bare /path/to/old/repository/dev.git dev.git
git clone --bare /path/to/old/repository/private.git private.git
git clone --bare /path/to/old/repository/atsal.git atsal.git
git clone --bare /path/to/old/repository/web.git web.git
git clone --bare /path/to/old/repository/heasoft.git heasoft.git
git clone --bare /path/to/old/repository/mobile.git mobile.git
```

As a root user, correct the permissions on the transferred archives:

```
su root
find . -exec chmod 744 \{\} \;
find . -exec chown atsal_git \{\} \;
```

Run this command:

```
gitolite setup
```

6.1.2 Git Administration

Once the archive is set up, one or more project members are designated as Git administrators. It is not necessary for sysadmins to manage Git user access.

On a **client machine**, a Git administrator checks out the gitolite-admin project. This is the mechanism used to add and remove users and control their access.

```
cd ~
git clone atsal_git@simone.cfa.harvard.edu:gitolite-admin.git
Cloning into 'gitolite-admin'...
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (6/6), done.
Checking connectivity... done.
```

To add a user, say Alice, Alice first creates a public key for herself on her own machine. (There may already be a public key for Alice's machine, but it is preferable to generate a separate keypair for this use.)

```
cd ~
ssh-keygen -t rsa
```

This will respond with:

```
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/#yourusername#/.ssh/id_rsa):
```

Instead of the default name, `id_rsa`, use Alice's username, e.g. `alice`. This name is how Gitolite identifies the user to Git, so it is important to conform to this naming convention.

Next, `ssh-keygen` prompts:

```
Enter passphrase (empty for no passphrase):
```

Enter a secure passphrase. In normal use, you will only have to type it once or twice, so there is no excuse not to use a very secure passphrase.

```
Enter same passphrase again:
```

Hit return again. You will get something that looks like this:

```
Your identification has been saved in /Users/alice/.ssh/alice.
Your public key has been saved in /Users/alice/.ssh/alice.pub.
The key fingerprint is:
48:c5:f5:e5:6f:f4:c1:00:cf:18:d9:ed:90:72:62:f1 alice@wonderland.local
The key's randomart image is:
+--[ RSA 2048]----+
|        o.o .    |
|     . o . + +   |
|      +   = E o  |
|     . . . * + * |
|      . u   o + +|
|           .   + |
|                o|
|                 |
|                 |
+-----------------+
```

You may be wondering what a randomart image is. I know *I* am. You will be relieved to know that you do not need to memorize it and type it in.

Copy Alice's public key (*never* the private key) to your client. Keys are stored in `gitolite-admin/keydir`. Keys are stored in a directory named for each computer,

in order to avoid name conflicts. For example, if Alice uses nodes wonderland and alicelaptop, the directory structure looks like this

```
gitolite-admin/keydir --+--wonderland -- alice.pub
                        |
                        +--alicelaptop -- alice.pub
```

The key name determines the user's name to Git, while the directory that represent nodes above have no special meaning beyond avoiding name conflicts.

Move the key into `~/gitolite-admin/keydir/nodename`. (Normally such a key would be added to the server's `authorized_keys` file, used by ssh. Gitolite manages this part of the process for you—don't edit the `authorized_keys` file by hand.)

```
mkdir ~/gitolite-admin/keydir/nodename
mv alice.pub ~/gitolite-admin/keydir/nodename
```

Next, add it to the archive and push to the server:

```
cd ~/gitolite-admin
git add keydir/nodename/alice.pub
git commit
git push
```

To remove a user:

```
cd ~/gitolite-admin
git rm keydir/nodename/alice.pub.
git commit
git push
```

Gitolite scans the keydir and transfers keys in and out of the `authorized_keys` file based on the keys present in the `keydir` directory. You can disable a user's access without discarding the public key—see the next section.

6.1.3 Managing Git Users with Gitolite

A Git administrator manages user access to specific Git archives by cloning the gitolite-admin archive, as mentioned in the preceding section.

```
git clone atsal_git@simone.cfa.harvard.edu:gitolite-admin
```

The file `~/gitolite-admin/conf/gitolite.conf` looks something like this. In this example, `tkent` and `rsmith` have read/write access to make changes to the `gitolite-admin` files, as well as to all the archives associated with the ATSAL project. Alice can read/write the ATSAL project, but not add and remove users. This file is edited, committed and pushed to the server in order to change user access rights.

```
# Permissions:
# R, for read only
# RW, for push existing ref or create new ref
# RW+, for "push -f" or ref deletion allowed (i.e., destroy information)
# - (the minus sign), to deny access.

# Gitolite administrators
repo gitolite-admin
    RW+     =   tkent
    RW+     =   rsmith

# This is part of gitolite
repo testing
    RW+     =   @all

# ATSAL project archives
repo atsal dev private web mobile heasoft
    RW+     =   tkent
    RW+     =   rsmith
    RW+     =   alice
# For later
#    R       =   @all
```

6.1.4 Cloning a Repository From a Gitolite-managed System

The syntax is slightly different when cloning a git repository via gitolite. Instead of specifying the remote path fully, it is specified relative to `~/repositories`:

```
git clone --recursive atsal_git@simone.cfa.harvard.edu:atsal.git
```

The command above clones to directory ./atsal. To clone to *directory* instead:

```
git clone --recursive atsal_git@simone.cfa.harvard.edu:atsal.git
directory
```

6.1.5 Backing up the Repository

This procedure is for the Mac or other systems that use `launchctl`. In this case we want to do a backup at 3:01 AM every day. The backup will be initiated by the backup system, not the master.

The backup procedure is a shell file in `/Users/tkent/bin/BackupATSAL.sh`. The location of the backup script is specified by full path because it will be run from the system. Since we want this to run regardless of who is logged in, or even if nobody is logged in, place the plist in `/System/Library/LaunchDaemons`.

First create `tom.atsal.plist` (or whatever you want to call it):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
	<key>Label</key>
	<string>tom.atsal</string>
	<key>ProgramArguments</key>
	<array>
		<string>/Users/tkent/bin/BackupATSAL.sh</string>
		<string>--argsifany</string>
	</array>
	<key>LowPriorityIO</key>
	<true/>
	<key>Nice</key>
	<integer>1</integer>
	<key>StartCalendarInterval</key>
	<dict>
		<key>Hour</key>
		<integer>3</integer>
		<key>Minute</key>
		<integer>1</integer>
	</dict>
</dict>
</plist>
```

Adjust the label string above if desired, now "tom.atsal," to be something unique relative to other plists. Supply an absolute path to the backup script.

This is intended to run BackupATSAL.sh at 3:01 AM each day. It actually runs at 03:01 and 15:01. The documentation does not clarify this.

Once the plist is copied to /System/Library/LaunchDaemons, it will begin running with the next reboot. To load the new launch daemon without having to reboot:

```
launchctl load /System/Library/LaunchDaemons/tom.atsal.plist
launchctl start tom.atsal
```

To stop and unload the daemon:

```
launchctl stop tom.atsal
launchctl unload /System/Library/LaunchDaemons/tom.atsal.plist
```

To list status:

```
launchctl list
```

This executes `/Users/tkent/bin/BackupATSAL.sh`. This script must be kept up to date with respect to the location of the archives and the list of specific archives to be mirrored.

This is an optional interactive program to monitor and configure launch daemons:

http://www.soma-zone.com/LaunchControl/

A sample backup script is in `atsal/dev/bin/BackupATSAL.sh`.

6.1.6 Changing the Origin for Everyone

Probably not all of this is necessary, but I had to jump through all these hoops to redirect the repos to a new origin, and I am not at all sure that there are not branches that are pointed incorrectly.

For all repos, do everything in this section.

```
cd repo
git config remote.origin.url atsal_git@simone.cfa.harvard.edu:repo.git
```

If the repo has submodules:

First, edit the .gitmodules file:

```
[submodule "web"]
   path = web
   url = atsal_git@simone.cfa.harvard.edu:web.git
[submodule "private"]
   path = private
   url = atsal_git@simone.cfa.harvard.edu:private.git
[submodule "dev"]
   path = dev
   url = atsal_git@simone.cfa.harvard.edu:dev.git
```

Also change these:

```
git config submodule.dev.url atsal_git@simone.cfa.harvard.edu:dev.git
git config submodule.private.url
atsal_git@simone.cfa.harvard.edu:private.git
git config submodule.web.url atsal_git@simone.cfa.harvard.edu:web.git
```

```
git remote rm origin
git remote add origin atsal_git@simone.cfa.harvard.edu:repo.git
git config master.remote origin
git config master.merge refs/heads/work
git push
```

If you are in a branch other than master, you need to merge with master too. In this example, the current branch is assumed to be work.

```
git commit
git push
git checkout master
git merge work
git push --set-upstream origin master
git checkout work
```

6.2 Setting up the Archive File Tree

While archives can be placed anywhere in the server file tree, if they will be shared with readonly users via HTTP, they should be somewhere in the file tree that is shared by the Apache server (or whatever server is in use). This is a good idea even if the archive is not yet available for public access. For this example we assume that the root of the shared file tree (called the "document root" by web hosts) is /var/www/htdocs. On a Macintosh, the default server root is /Library/Webserver/Documents.

We assume here that Apache or some server is configured to act as a web server on the default port (80), and that access on this port is enabled, and that the server machine has a static IP and domain name, or a domain name that automatically updates to track changes to a dynamically allocated IP address. If you can access a web page on your server, it can be used with Git.

The instructions below assume that the primary archive itself is not shared, but instead a clone of the archive. This isn't a security issue (since the archive is read-only), it is a matter of open source access management. It may be desirable to put only major releases of the archive online, rather than the daily updates shared by developers and sometimes broken.

First create a git directory in the default root:

```
cd /Library/Webserver/Documents
mkdir archives
```

This directory will contain all the Git projects. Enter this directory and clone the bare Git archive:

```
cd archives
git clone --bare /path/to/atsal_project atsal.git
```

That's it: the archive is available via SSH. When you want the archive to become publicly available via HTTP:

```
cd /var/www/htdocs/git/atsal.git
mv hooks/post-update.sample hooks/post-update
chmod a+x hooks/post-update
```

Repeat this same process for each project you wish to make public.

6.3 Archives vs. Bare Archives

A source file tree contains a set of files, some of which are user created and belong in an archive; some of which are computer generated (like object files), and therefore should be omitted. For example, the `myproject` directory might contain:

```
helloworld.c
README
```

It is easy to create an archive from a source file tree: `git init` creates the hidden `.git` directory—the archive. Then `git add` populates it with files. Now the project contains both the git archive and the original files that make it up. The original files are now the current set of checked out files, while the archive's hidden files are considered the master. This is called a "non-bare" archive because it contains both files and archive.

By contrast, a bare archive is simply the archive itself. You create this from a non-bare archive as follows:

```
git clone --bare myproject myproject.git
```

Now `myproject.git` contains a copy of `myproject/.git`, except that it lacks any information about checked out files because there aren't any. Multiple Git users

can clone this bare archive and make commits to the bare archive without conflict.

But if your archive commits files to a non-bare archive, confusion arises, because your current version does not match the user of the other archive. It is possible to do this by using tricks with branches, but it is a bad idea.

Hence: **a shared archive should always be a bare archive**.

Converting an archive to a shared bare archive is easy, but converting an archive with submodules into a shared bare archive is more of a hassle.

The master ATSAL archive presently consists of an ATSAL archive with three submodules, `dev`, `private`, and `web`. The `dev` archive in turn contains submodules `mobile` and `heasoft`.

To convert this into a bare archive, we must first create a bare archive of each submodule:

```
git clone --bare atsal atsal.git
git clone --bare atsal/dev dev.git
git clone --bare atsal/dev/mobile mobile.git
git clone --bare atsal/dev/heasoft heasoft.git
git clone --bare atsal/private private.git
git clone --bare atsal/web web.git
```

Now move the non-bare archive aside. (You will probably delete it as soon as you trust your bare archive.)

```
mv atsal atsal.nonbare
```

The newly created bare archives do not include their submodules.

```
git clone atsal.git
cd atsal
git checkout master
```

Since the starting archive contains submodules, the .gitmodules files that list these archives refer to other non-bare archives. We need to correct this manually, by redirecting them to point to the bare archives. Here is the original version of `.gitmodules`:

```
[submodule "dev"]
	path = dev
	url = /Users/tkent/atsal/dev
[submodule "private"]
	path = private
	url = /Users/tkent/atsal/private
[submodule "web"]
	path = web
	url = /Users/tkent/atsal/web
```

Edit this to create this instead:

```
[submodule "dev"]
	path = dev
	url = /Users/tkent/archives/dev.git
[submodule "private"]
	path = private
	url = /Users/tkent/archives/private.git
[submodule "web"]
	path = web
```

```
        url = /Users/tkent/archives/web.git
```

```
git submodule init
git submodule update
cd dev
```

By default, the `dev` repository is headless—no branch is selected. You select a branch as follows:

```
git checkout master
```

As above, edit the `.gitmodules` to point to the bare archives.

```
cd mobile
git checkout master
cd ../heasoft
git checkout master
cd ..
git add .gitmodules
git commit
git push

cd ..
git add .gitmodules
git commit
git push
```

Now the cloned module has updated the `.gitmodules` to use the bare submodules, and the changes have been committed to the local archive and pushed to the bare archive.

7 Archive Structure

The ATSAL project spans several sub-projects, each of which may be extended as part of ATSAL or independently. Some means of coordinating releases across such projects is needed. Git offers submodules for this purpose. The organization below addresses the following requirements:

- Git archives grow rapidly in size, so splitting the ATSAL archive into three main partitions reduces space demand. Most developers and release engineers will need only one or two of these at a time.
- Archives are organized so those working on submodules such as AtomDB can do so without checking out the ATSAL code. Changes made to the submodules are not automatically integrated into ATSAL; ATSAL developers integrate such changes only when they are ready. A Git module keeps track only of the location and version of any submodules; updates from submodules are only done manually.
- Since the ATSAL dev archive will eventually become open source, we try to give careful thought to its organization and content. But active developers also need to share other project-related materials that are not yet well organized or are unsuitable for public release. Hence we need a private archive.

Hence we maintain three completely independent top level ATSAL archives:

web	Contains web pages, tarballs, and documentation necessary for the user-facing portion of the ATSAL web site. This package is managed by release engineers,[2] and contains scripts which update the web site as needed.
dev	Contains code developed as part of ATSAL, including third party libraries that are modified for ATSAL, as well as documentation, installers, etc. Anything essential for ATSAL development is included here, whether for an open source developer or an internal developer. This is checked out by any ATSAL developer.
private	Contains code that might be of value to other developers, or code in early development stages that is not yet properly integrated into the tree, or informal or private documentation. This tree serves as an informal exchange medium. This is also checked out by any ATSAL developer, but is not available to open source users.

The following sections describes the contents of each of these archives.

7.1 Directory Structure

It doesn't matter where the root of this tree is placed, but the rest of the tree must be maintained as shown. ATSAL developers need the dev archive, and the private archive is recommended, as well as all submodules.

[2] Or software developers wearing release engineering hats. Or scientists wearing software developer hats topped by release engineering hats.

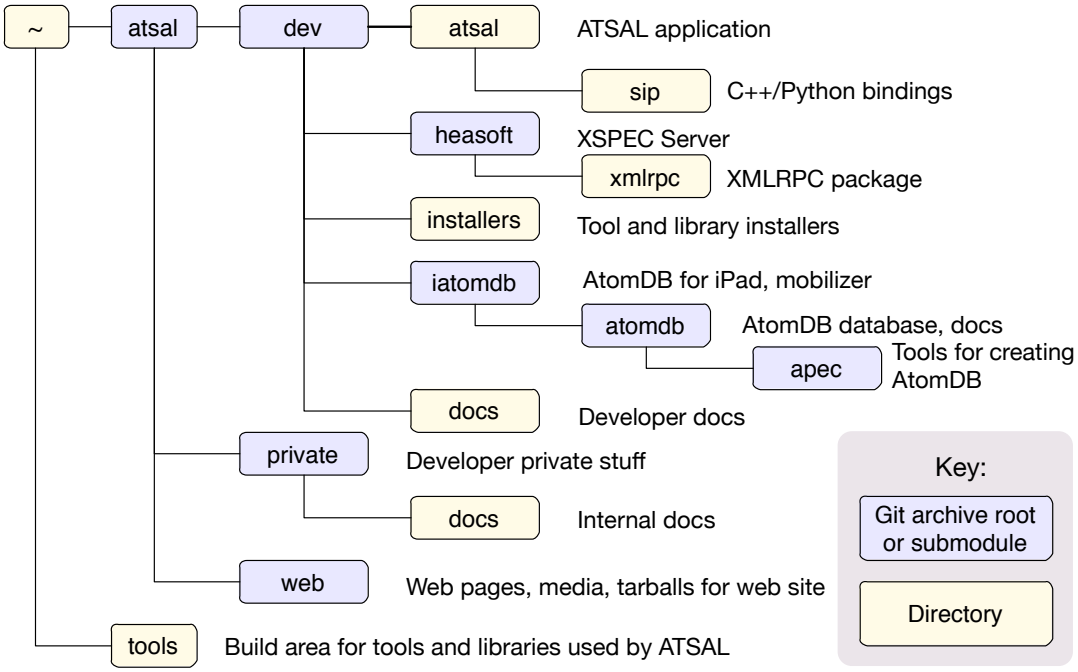


Figure 2. Directory structure, with Git archives shown in blue. Archives may be checked out and updated independently. Updates are not propagated to parent project(s) until a parent project maintainer chooses to do so. However, this means that developers must ensure that all submodules that should be coordinated are committed at the same time.

In `dev`, the atsal directory contains code necessary to create a test client that operates apart from Python, permitting better debugging control. It also builds a dynamic library which is loaded into Python to produce the true ATSAL application. The `sip` subdirectory contains files needed to generate the bindings between C++ and Python.

The `heasoft` module begins with a specific release from HEASARC at GSFC. It contains only the subset of the HEASOFT release that is actually modified to produce the XSPEC server. The archive has a readonly relationship to its HEASARC parent. Eventually we hope to merge the server extensions back into the HEASARC-maintained code base. XMLRPC is a third party library that implements XML-based remote procedure calls. It is integrated in source format instead of built as a free-standing library to simplify debugging and solve some other problems. This module also has a read-only relationship with its parent. It will probably not be necessary to merge later updates to this library into XSPEC server or ATSAL.

The `installers` directory contains tarballs of third party tools and libraries, as a backup in the event that the appropriate version of a software component is no longer available on the web. This directory contains only masters, no installed software or sources. Software installed from these installers is placed in an untracked directory, `~/atsal/tools`. Build products are used directly from the tools directory or sometimes from `/usr/local` or some other shared install location.

The `iatomdb` submodule contains the iPad application for AtomDB, and the associated "mobilizer" tool, which converts the FITS database to a format that is more compact for use in iAtomDB.

The `atomdb` submodule contains the AtomDB database in FITS and MySQL formats, along with associated docs, tools, and web pages. It does not contain the code that generates AtomDB—that is stored in the apec archive.

The `docs` directory contains documentation of use to ATSAL developers. It may also contain end user docs before they are ready for deployment to the web site. (If you move a document from one archive to the other, delete it from the old archive to avoid confusion.)

The `private` archive contains anything of potential use to other developers—source code or tarballs that might be incorporated later, early documentation that may become rearranged later, documents not intended for public access, etc. The organization of this archive is informal. It serves the same basic purpose as e.g. DropBox, with the addition of version control.

The web archive contains materials used to generate the public-facing portion of the web site.

The tools directory is *not* an archive. It is a standard place to put installed tools that are local to ATSAL (vs. being shared from, say, `/usr/local`).

The `apec` archive is TBD.

8 Setting Up a Developer System

These instructions cover Linux and Mac OS X. Install Git according to instructions in a previous chapter.

8.1 Using a Preconfigured Virtual Machine

The virtual machine I created runs under Parallels on a Macintosh. If you are working with a copy of this machine, you have almost everything you need. A generic user account, called adev (ATSAL developer) exists. The password is available from Tom Kent.

- When you start up the copy of the VM, it will ask if you are moving or copying. It is really asking whether to assign a new MAC address for networking. Say you are copying to assign the new address.
- Set the Parallels settings for the virtual machine as you see fit. I suggest that roughly half of the processor cores and memory be devoted to the VM, more or less depending upon intended use. Do not adjust virtual disk sizes.
- It is essential to personalize your virtual machine before using it. First, do section 7.2 below, to tell Git your real name and your preferences, among other things.
- Get an account and VPN information as needed for the machine with the shared Git server.
- Fine tune the `.bash_profile` script in the home directory as needed. I set up this script for the default shell on Macs. If you have a different preferred shell, you will need to transliterate the script for the other shell.
- If you will be using Microsoft Word, Omnigraffle Pro (drawing app), Tower (Git user interface), or BBEdit (text editor) on the VM, you will need to configure your own licenses.
- If you will be performing ongoing development, it is a good idea to configure Time Machine within the virtual machine to back up to a disk on the physical host. Do this by mounting the external disk and setting Time Machine to back up to it.
- Do not use Time Machine on the physical machine to back up the virtual machine. This is too inefficient, and redundant with the preceding step.
- The Mac virtual machine is configured with two virtual disks, “ATSAL” and “Recovery.” ATSAL is the boot disk, and is configured to grow to up to 256 GB in size. Recovery is limited to the default maximum of about 65 GB, and serves no real purpose other that to allow you to run Disk Utility if needed on the ATSAL disk, by first booting the recovery disk. This arrangement was the only way I could find to create a virtual disk that could grow beyond 65 GB in size.

8.2 Basic Setup

These global settings configure your Git work environment. They are not checked in with your project, because they vary from user to user.

8.2.1 Installing Git

On a Macintosh, install the XCode package from the App store, and the optional XCode developer tools. The latter requires a developer account with Apple. Or see *Pro Git,* starting on page 8.

On Linux systems, you can install Git from source or from binary according to instructions published in *Pro Git,* p. 8.

8.2.2 Establishing an Identity

This is critical, because it determines how Git identifies you when you push changes. Your username is **not used**:

```
git config --global user.name "John Doe"
git config --global user.email johndoe@example.com
```

8.2.3 Keeping it Simple

This configures Git for safer default operation. It will shortly become the new default. Make sure it is set though.

```
git config --global push.default simple
```

8.2.4 Related Tools

Some Git commands invoke a difference tool or a text editor. You can select your preferred tools with these commands, issued once per new user. In this example, I selected BBEdit's difference tool (part of the BBEdit command line tools package), and emacs for editing. (Note: BBEdit cannot be used to edit commit messages because it doesn't work correctly if BBEdit is already running.)

```
git config --global merge.tool bbdiff
git config --global core.editor emacs
```

To view your settings:

```
$ git config --list
core.editor=emacs
merge.tool=bbdiff
user.name=John Doe
user.email=johndoe@example.com
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
core.ignorecase=true
core.precomposeunicode=true
```

8.3 The .gitignore File

Note: `.gitignore` files are checked in like any others, and are already present in a project you clone from a Git server.

Like other VCSs, Git does not add files to a project automatically, since it has no way to know which ones are regenerated automatically vs. user-created. Only user-created files belong in the archive. Git warns users about files that are present but not yet added to the archive, to make it easier to remember to add new files to an archive. It depends on a file called `.gitignore`, in each archive's root directory, to exempt certain types of files from these warnings. A sample file is shown below. It blocks generated files such as objects and libraries, build trees, "old" directories, etc.

Some files, such as Mac OS X XCode project files (*.xcodebuild files, really a directory tree), may be created by hand *or* generated by a utility such as Qt's `qmake`. Hence these files are not ignored by default, but you can add specific projects to the ignore list if they are regenerated. A properly configured `.gitignore` should ignore all generated files.

```
# You can end patterns with a forward slash (/) to specify a directory.
# You can negate a pattern by starting it with an exclamation point (!).

*.a                 # no .a files
*.dylib             # no .dylibs
*.o
*.so
*.log
*.pha
*.build/      # build directories
Old/          # no Old directories
old/
*copy/        # Duplicated directories
tmp/
bin/
moc_*.cpp     # Files generated by moc

# Other examples:
# !lib.a    # but do track lib.a, even though you're ignoring .a files
# /TODO     # only ignore the root TODO file, not subdir/TODO
# build/    # ignore all files in the build/ directory
# doc/*.txt # ignore doc/notes.txt, but not doc/server/arch.txt
```

Important: each submodule has its own `.gitignore`, so make sure you modify the correct one.

The XMLRPC subdirectory, though it is not an independent submodule, also has its own `.gitignore` file at this time.

See Appendix 1: .gitignore for XCode Project Files.

8.4 XCode's Git Support

XCode has built-in support for Git, but because the ATSAL project spans multiple XCode projects, you should not use this built-in support. Use the command-line Git commands instead.

Maybe this prohibition is not necessary, but it is recommended for now.

Git supports the concept of a staging area, conceptually a temporary storage area for files prior to performing a commit.

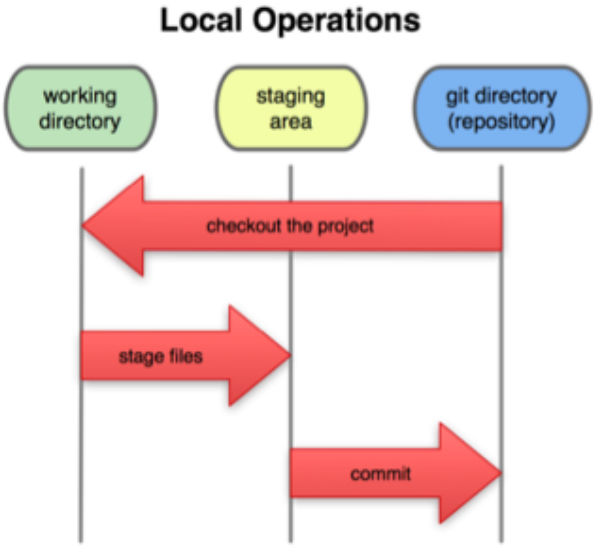


Figure 3. This drawing, figure 1.6 from Pro Git, shows that Git supports an intermediate location for files awaiting checkin, called the staging area. Developers can add files explicitly to this area, then commit all the staged files to the repository.

The staging area is a helpful concept for a relatively complex project like this one, because it prevents "overcommitting." That is, it is easy to accidentally add files to an archive that don't belong there because they are build products. They can be deleted afterwards, but they continue to lurk in the archive's history. So by first moving files to a staging area, you can cancel any superfluous additions before they become a permanent part of the archive, using

```
git rm --cached filename
```

This command removes the file from the staging area but does not delete it from your system.

Staging is exceptionally helpful in the early stages of setting up a project or familiarizing yourself with a new, complex source tree. It can be bypassed later if desired.

8.5 Workflow

Git permits several styles of workflow as needed in projects of varying levels of complexity. For ATSAL, we use "centralized workflow," a common and simple model. In this workflow we designate a single repository as the shared master resource, and each developer pushes and pulls from this central resource. For n developers there are n+1 repositories, each assumed to be backed up by some incremental backup utility, for 2n+2 complete copies. The centralized repository appears as the origin for each developer.

8.6 Cloning the ATSAL Tree

The ATSAL tree consists of three major submodules (`dev` for development, `private` for developer-private material, and `web` for web site materials).

> Note: Cloning the ATSAL tree does NOT clone all the development tools and libraries on which ATSAL depends. Those are usually

stored in `~/tools` (which you build from tarballs that are included in `atsal/dev/installers`) or in `/usr/local`.

If you are starting from a previously prepared virtual machine, all this stuff is pre-installed.

8.6.1 Cloning the Entire Tree

If you have ample disk space, simply clone the whole tree. Cd to the directory in which you wish to place the tree (usually your home directory) and clone it:

```
cd ~
git clone --recursive atsal_git@simone.cfa.harvard.edu:atsal.git
```

Unfortunately this will prompt you for your password for each submodule in the project, unless you have configured `ssh` to avoid the need for a password, as described earlier.

The checks out the ATSAL repository, as well as all the submodules upon which ATSAL depends. This may take awhile. When it completes, a new directory, called `atsal`, is created. If you want to place it somewhere else, add a directory argument:

```
git clone --recursive atsal_git@simone.cfa.harvard.edu:atsal.git atsal2
```

After cloning, you must explicitly select the branch on which you will work for each submodule. Something like this:

```
cd ~/atsal
git checkout work
cd dev
git checkout work
cd heasoft
git checkout work
cd ../mobile
git checkout work
cd ~/atsal
cd private
git checkout work
cd ../web
git checkout work
```

These steps ensure that you have the most recent contents of each of these archives as a starting point, since the default master branch is updated only for major releases.

Although there are six archives, only dev and heasoft are typically pushed, pulled, and branched in unison, since dev contains a client and heasoft contains a complementary server. The others are logically independent, once set up.

If you are only interested in part of the tree, see the following section.

8.6.2 Cloning Individual Submodules

There are three top level ATSAL submodules, so create a directory for all three. I use atsal in the home directory, but you can put the root directory anywhere. The clone process is a one-time event (unless you are creating a new clone of the entire archive on a new machine).

```
cd ~
```

```
mkdir atsal
cd atsal
```

First, clone the developer directory for ATSAL:

```
git clone --recursive atsal_git@simone.cfa.harvard.edu:dev.git
```

This clones all the files (and history) for the dev project, and its submodules. The originating archive is known as the origin. The clone command sets your local master branch to follow the origin's master branch. You can change this later.

> If you forget the `--recursive` switch, it isn't too late. Cd into the newly created directory and issue a submodule update command:
>
> ```
> cd dev
> git submodule update --init --recursive
> ```

Return to the atsal directory if necessary. ATSAL developers also need the ATSAL private directory. (This archive has no submodules, so the `--recursive` switch is superfluous, but it doesn't hurt to provide it in case this changes in the future.)

```
cd ~/atsal
git clone --recursive atsal_git@simone.cfa.harvard.edu:private.git
```

If you are working on the ATSAL web site or release packages, clone the ATSAL web archive:

```
cd ~/atsal
git clone --recursive atsal_git@simone.cfa.harvard.edu:web.git
```

All these are independent. Pushing one does not push the others.

As stated in the previous section, checkout the work branch to obtain the most current code.

8.7 Learning Git Without Shooting Yourself in the Foot

Once you have cloned a repository, you can do all the commits, branches, deletes, experiments, etc., that you like, with the worst-case outcome that you bollix up your **local** repository. If that happens you can simply delete everything and start fresh.

Just don't push any changes until you are confident. Pushing is the action that transfers your changes to the shared repository. You may want to clone the shared repositories and do pushes and pulls to the clone while you are learning. Then you can simply discard the cloned repository when you are done.

The single most important chapter in the Git book is chapter 2. It is also a very good idea to have a clear understanding of staging, branches and submodules. A **very** good idea.

If you are accustomed to informal methods of version control, such as renaming documents with appended dates or version numbers to avoid overwriting earlier versions, abandon this habit with Git and edit the document in place, allowing Git to track the older versions automatically.

8.8 Doing a Build

At the time of this writing, several parts of the build process are kicked off independently, in order to get improved build times. Here is the process for a new build:

8.8.1 XMLRPC

First build the XMLRPC libraries:

```
cd ~/atsal/dev/heasoft/Xspec/src/xmlrpc-c
sh macbuild.sh
```

8.8.2 XSpec Server

Next, build the XSpec server:

```
cd ~/atsal/dev/heasoft/BUILD_DIR
./configure
make
make install
```

8.8.3 XClient (test bed)

Next, build the XClient standalone. Right now this is done interactively, using XCode; later it will also be possible to do scripted builds.

```
cd ~/atsal/dev/atsal
qmake XClient.pro
open XClient.xcodeproj
```

Then build the XCode project and run it.

8.8.4 ATSAL Python Library

Finally, build the Python library:

```
qmake ATSAL.pro
open ATSAL.xcodeproj
```

Build this component.

8.8.5 Running the Library

TBD

9 More on Git

9.1 Git Tags

Tags assigned in the local archive **do not automatically propagate** to the origin. This isn't such a bad thing though, since tag assignment should generally be managed by the release engineer. The RE can add the `--tags` to the Git push command to cause all local tags to propagate to the remote archive. Tags do come along when you do a pull though.

Tags are assigned independently in each archive. However, since a module tracks specific versions of its submodules, pulling a module always gets the correct version of its submodules.

`git tag`	List existing tags
`git tag -a v1.4 -m 'my version 1.4'`	Applies a tag and adds a short descriptive message.
`git log --pretty=oneline` `git tag -a v1.4 9fceb02`	Applies a tag to an older commit by specifying the first few characters of the commit SHA-1 hash.
`git show <tag-name>`	Shows info on specified tag.

The HEASOFT portion of the project, used for XSpec Server, uses a special tag numbering scheme. It begins with the HEASOFT-assigned tag 6.15.1. Later tags are of the form 6.15.1.*major.minor*, that is, extensions to the HEASARC release. When HEASARC updates are merged into our source tree, we start with the HEASARC-assigned version number and number forward from there. For example, if HEASARC releases 6.16, our numbering begins at 6.16.0.1.

9.2 Git Branches

Git branches work differently from those in most other VCSs, allowing for different methods of use. This is a mixed blessing. If you are an occasional user, branches can leave you confounded.[3] You won't lose anything you have committed, but you may lose easy track of it. If you are a regular user of the archive, you can achieve more flexible local development, as well as a more streamlined origin.

A reminder: since Git archives store a full copy of each version of each file, they grow quickly in size. It is effectively impossible to prune archives after they are created. But it *is* possible to manage pushes in a way that reduces clutter. For a bulky project like ATSAL, this is essential.

Start by reading chapter 3 of Scott Chacon's book *Pro Git*.

http://www.git-scm.com/book

Summarizing the book:

[3] I hope to get out of therapy myself before long.

- Branches are "lightweight," easily and quickly created, destroyed, and switched among.
- "HEAD" is shorthand for your current location in the version tree. It may be any branch, and any point along the branch (not just the tip).
- Although it is easy to switch branches, you must commit changes to the current branch prior to switching. Even if the changes are partial and not yet usable, the commit is only to the local archive, and you can pick up later and finish the work. (Alternatively, see `git stash` to put aside work in progress temporarily.)
- **There is no formal association between a branch in the local repository and a branch in the shared repository**. You can push any local branch into any origin branch. A local branch does not exist in the origin repository unless someone creates it. This is necessary to the branching model, but it is easy to forget which branch to push to. Tracking branches make this easier by setting up a default origin branch for subsequent pushes.

  "Tracking branches" are simply local branches that track a specific branch in the remote tree. Pushes and pulls in a tracking branch apply to the tracked branch. For example, if you usually work on version 2 code, but occasionally make bug fixes to version 1, you can track a version 1 branch to make these fixes. The following command pushes the current branch to origin's `work` branch, making the origin's `work` branch the tracking branch, or new default for later pushes.

  ```
  git push --set-upstream origin work
  ```

  After the command above,

  ```
  git push
  ```

  Also pushes the current branch to origin's `work` branch.
- **Since submodules are logically independent, so are branches and tracking branches within each submodule.** If you create a branch called `bugfix82` in the `atsal` repository, no such corresponding branch exists in the `heasoft` repository. It is not harmful to work in different branches in different submodules, but it can get confusing.
- When you switch branches with a checkout command, Git modifies all necessary files in your current project directory to match those in the selected branch. **You are changing the current project to match the branch, *not* creating a second directory tree with the branch.** This is faster, less space-intensive, and, of course, the reason why you need to commit your previous branch first. (You could also clone the archive and create one or more independent archives, but there is no need to do this.)
- The git merge command merges code from a specified branch into HEAD, the currently selected branch. After such a merge, both branches contain the same code, so the temporary branch is superfluous. You can delete the superfluous branch with `git branch -d <temporary-branch>`.

- You can maintain multiple local branches, switching among them almost instantly with `git checkout`. This makes it easy to fix bugs in an older version, then return to current work, or to create and share experimental branches that might not belong in the master.
- Merging is subject to the same problems the apply to other VCSs. Merges proceed automatically, and usually work fine, when there are no conflicts (that is, multiple developers have not made changes to the same section of code). Such conflicts are reported, and must be resolved manually. And files such as Microsoft Word files are treated as binary blobs, requiring manual merging using Word's built-in merge tools. Many other proprietary formats offer no assistance for merging.

  If you wish to modify a "binary" file (one for which Git does not provide automated merging), and you are not the default owner/maintainer of the file, notify other project members by e-mail before making modifications. This helps to circumvent manual merge problems.
- "Remote branches" are snapshots of the branch state of remote repositories at the last time of contact. They are immutable "bookmarks," for reference. They are updated automatically whenever you contact a repository. They take the name "<remote-name>/<branch-name>," for example, `origin/master`.
- A remote branch can be deleted with the counterintuitive syntax:

  ```
  git push <remote> :<branch-name>
  ```
- "Rebasing" is another way, in addition to merges, to bring two divergent branches into sync. See section 3.6.1 of *Pro Git*. The effect of rebasing is the same as that of merging, but it produces a more comprehensible history by batching all the changes made in a sub-branch together.

  Don't overlook this part of the description:

  > 3.6.3 The Perils of Rebasing
  >
  > Ahh, but the bliss of rebasing isn't without its drawbacks, which can be summed up in a single line:
  >
  > **Do not rebase commits that you have pushed to a public repository.**
  >
  > If you follow that guideline, you'll be fine. If you don't, people will hate you, and you'll be scorned by friends and family.
- If you need to put aside work briefly and return to it later, consider stashing it. See `git stash --help`.

`git branch <branch-name>`	Create a new branch from the current branch. **Does not *switch* to that branch though!**
`git checkout <branch-name>`	Switches HEAD to point to the current version of <branch-name>.
`git checkout -b <branch-name>`	Creates a new branch and switches to it, same as both the preceding commands.

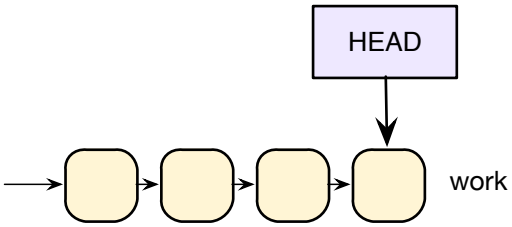
`git checkout --track <remote>/<branch-name>`	Create a tracking branch of the specified remote and branch. Pushes and pulls will apply to this branch rather than master.
`git commit`	Commits changes in your local directories to the HEAD branch
`git merge <branch-name>`	In effect, pulls code from HEAD into your local tree, and merges code from <branch-name> into it.
`git branch -d <temporary-branch>`	Deletes a temporary branch from the local archive. This is done after merging the branch into some other branch, such as the master.
`git branch`	Lists current branches, marking the selected one with "*".
`git branch --no-merged`	Lists unmerged branches. These are "orphans."
`git branch --merged`	Lists merged branches. Since they are merged, they are redundant and can be deleted if desired (except of course for master).

9.2.1 HEADs, and the Detaching Thereof

This section is adapted from a GitHub article:

https://github.com/sitaramc/git-notes/blob/master/articles/detached-head.mkd

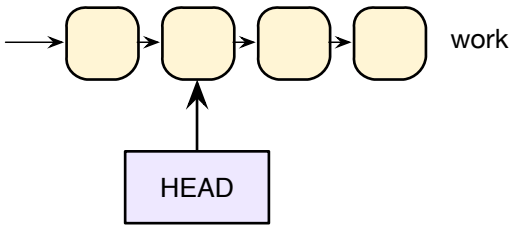
HEAD is shorthand for the tip of the current branch. It designates where your next commit will go. The checkout command is the most common way to set HEAD to point to a new branch.



It advances automatically to remain at the tip as commits are made. A detached head occurs when some operation points HEAD at some other node in the version tree.

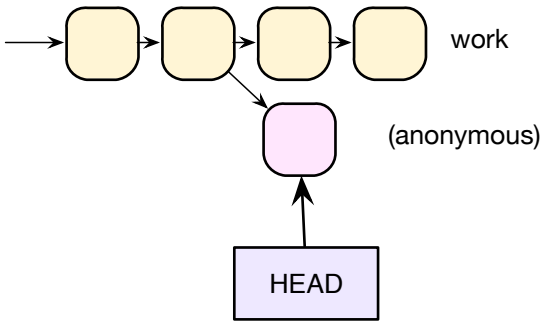
```
git checkout HEAD~2
```

(that's a tilde!) moves HEAD back two generations as shown.



A detached head has no branching strategy, because it does not point to the *tip* of any branch. A commit to a detached head creates an anonymous branch:

```
git commit
```



Here are some commands that result in a detached head:

```
git checkout master^        # parent of master
git checkout HEAD~2         # grandparent of current HEAD
git checkout origin/master  # a non-local branch
git checkout tagname        # since you cant commit to a tag!
```

Once in this state it is easy to lose work. It isn't actually deleted, but it can be hard to recover. If you realize you are in this state, and want to retain any commits you made while HEAD is detached,

```
git checkout -b <newbranch>
```

names the anonymous branch. You can then merge it back into another branch if desired.

If you don't care about the anonymous branch, or you haven't committed anything yet, just check out a different branch:

```
get checkout work
```

I can't think of a reason why Git lets you get into this state in the first place.

9.2.2 ATSAL Branches

Since each developer pushes only the changes they wish to share with others to the shared repository, **each developer's repository is different**. Think of the shared repository as the cleaned up version of your local repository, containing for the most part only versions that work. Each developer repository also contains local cruft. Each local repository is a superset of the shared repository, so if a corrupt shared repository is replaced with a developer's, the extra cruft is present. Git supports mirroring of archives specifically to address the problem of keeping copies of the shared archive.

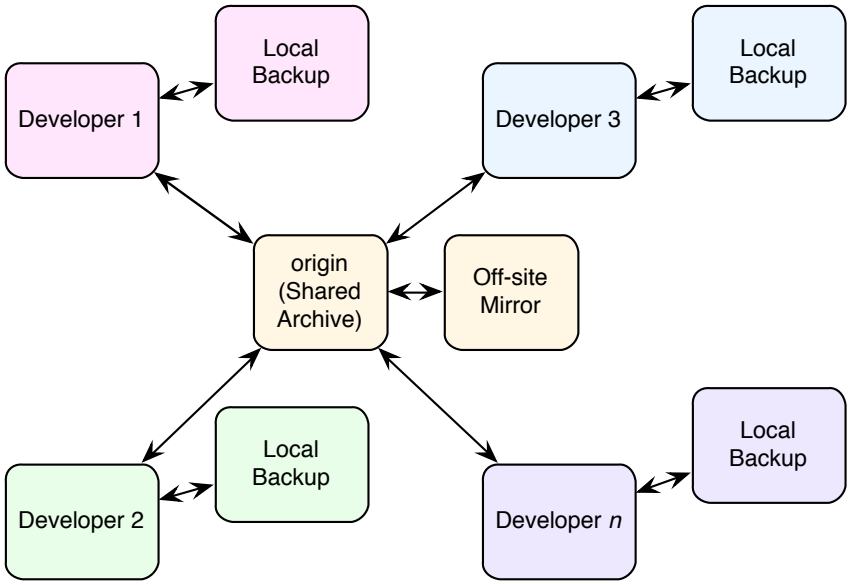


Figure 4. Each developer's repository contains local changes that are not pushed, in addition to the subset of changes that are shared, so repositories are not fully interchangeable. Each repository should be backed up local (with e.g. Time Machine or rsync). The shared archive is mirrored (via `git mirror`) to a different site.

After reading several articles on branching approaches, I found this discussion by Vincent Driessen:

http://nvie.com/posts/a-successful-git-branching-model

The diagram following, reproduced from this article, shows two major, permanent branches: `master`, for releases; and `develop` (we call this `work`) for work in progress. Another major branch is "release branches," created when a work branch is about to be released. Until the release branch is created, development on subsequent releases cannot be started.

Note that the `master` and `work` branches are always present in the origin. Release branches may be as well. Other branches are often created and destroyed by individual developers, and need not be pushed to the origin.

Driessen advocates sharing of branches between collaborators, apart from the origin. I think this is unnecessarily complicated, and don't recommend it.

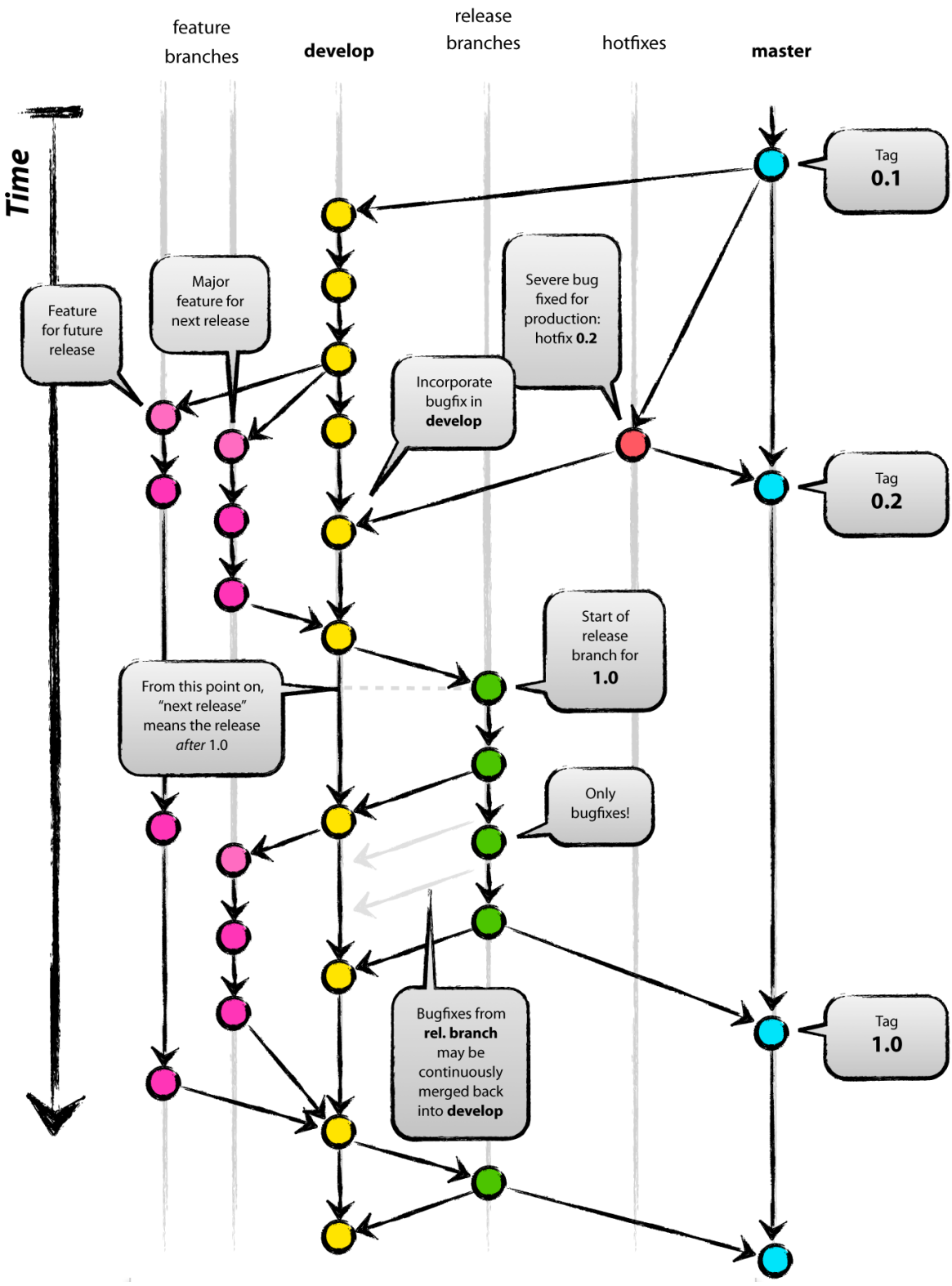


Figure 5. A branching strategy like this is appropriate for the dev and heasoft repositories. We call the "develop" branch "work." Diagram by Vincent Driessen.

Branches are committed to the local archive regularly, perhaps many times per day. Sub-branches are merged into the work branch prior to pushes to the origin.

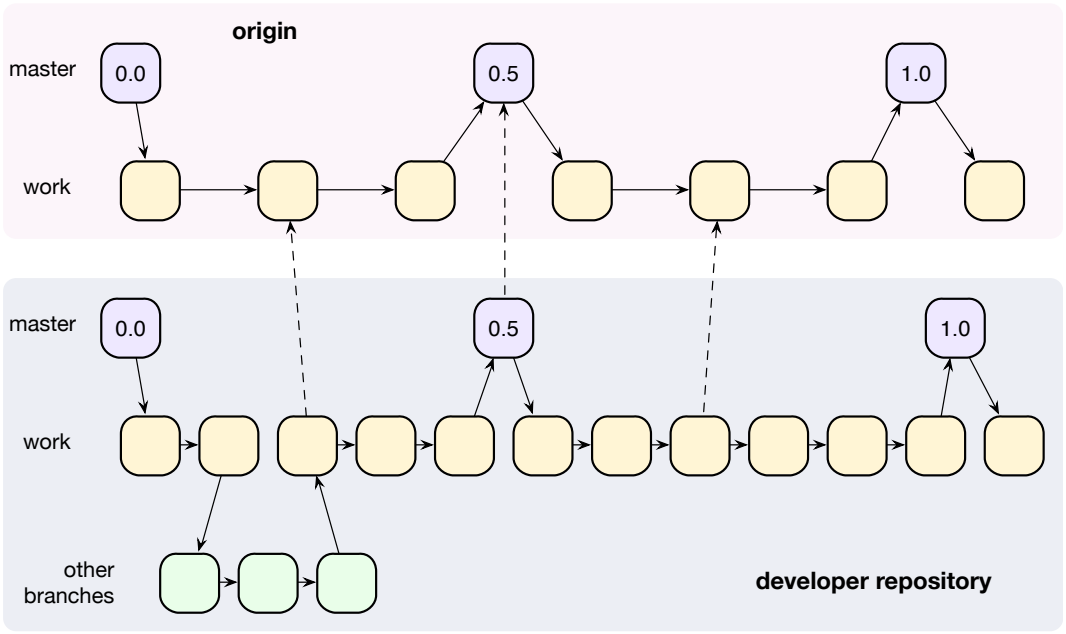


Figure 6. Developer repositories may have many sub-branches, but these are not typically pushed to the origin. The origin consists of relatively stable work versions and, in master, stable releases or major milestones. Typically each master version is tagged.

At this time, I suggest the following branching strategy for the `atsal` archive and the `heasoft` archive:

- `master`, the main branch, contains only stable, tagged versions. It may contain unreleased versions, but it typically contains only significant milestones
- a `work` branch contains normal work in progress, with commits made as frequently as desired. Work in this branch is normally merged with `master` only when stable or for sharing
- If a tagged release is later branched in order to apply bug fixes, the branch is named for the tag, e.g. `work4.15.1`
- When the decision is made to issue a release, a release branch is created, e.g. `release1.5` for upcoming release 1.5. Final release preparations take place on this branch, while the `work` branch continues with development of the next release. When the release is ready, it is merged into `master` and tagged.

9.3 Git Submodules

Submodules are completely independent—you update them manually when you want to sync up with changes made by others. A module contains only a reference to a specific commit for each submodule.

Submodule **gotchas**:

- If you make changes to both a submodule and a supermodule, and commit the supermodule but not the submodule, the supermodule will contain a reference to a version of the submodule that **does not exist** in the shared repository. Other people will try to pull this broken commit and stop inviting you to parties. To avoid this, **always commit changes in submodules first, *then* changes in parent modules**.
- `git submodule update` happily overwrites your current changes if you do not commit first. Don't do this.
- `git clone <archive-URL>` does not clone submodules by default. Use `git clone --recursive <archive-URL>`.
- When adding changes to a submodule to a supermodule's repository, use `git add submodule`, **not** `git add submodule/` (trailing slash). The latter deletes and replaces the submodule.

9.3.1 Branching in Submodules

All origin modules contain a `master` branch, which is tended only the release engineer (RE): developers never push to the master branch. They also contain a `work` branch, to which developers push code believed to be stable. The origin may also contain a release branch when a release is pending, so that developers can put the finishing touches on a new release is parallel with creation of a new release. So the basic rule is that most developer pushes are to the `work` branch.

Each submodule has slightly different branching strategy:

atsal	This exists only to coordinate commits of related ATSAL components. It isn't necessary to create branches other than work and master.
dev & heasoft	This is the root of ATSAL development, and has the most complex branching structure, one similar to that shown in the figure by Driessen. Typically, `dev` and `heasoft` branching should remain synchronized, since these components need to remain coordinated. If you create a branch such as `bugfix82` for `dev`, and there are any server implications, you should create the same branch for `heasoft`. This will create a more comprehensible history later.
mobile	Changes here are unusual, so elaborate branching isn't very important. Master and work are probably all that is needed.
web	Typically one person works on this at a time, so master and work are all that is needed.
private	Master and work branches are fine for this as well.

Here is a first approximation of a utility designed to assist with branch tracking in a multiple submodule environment:

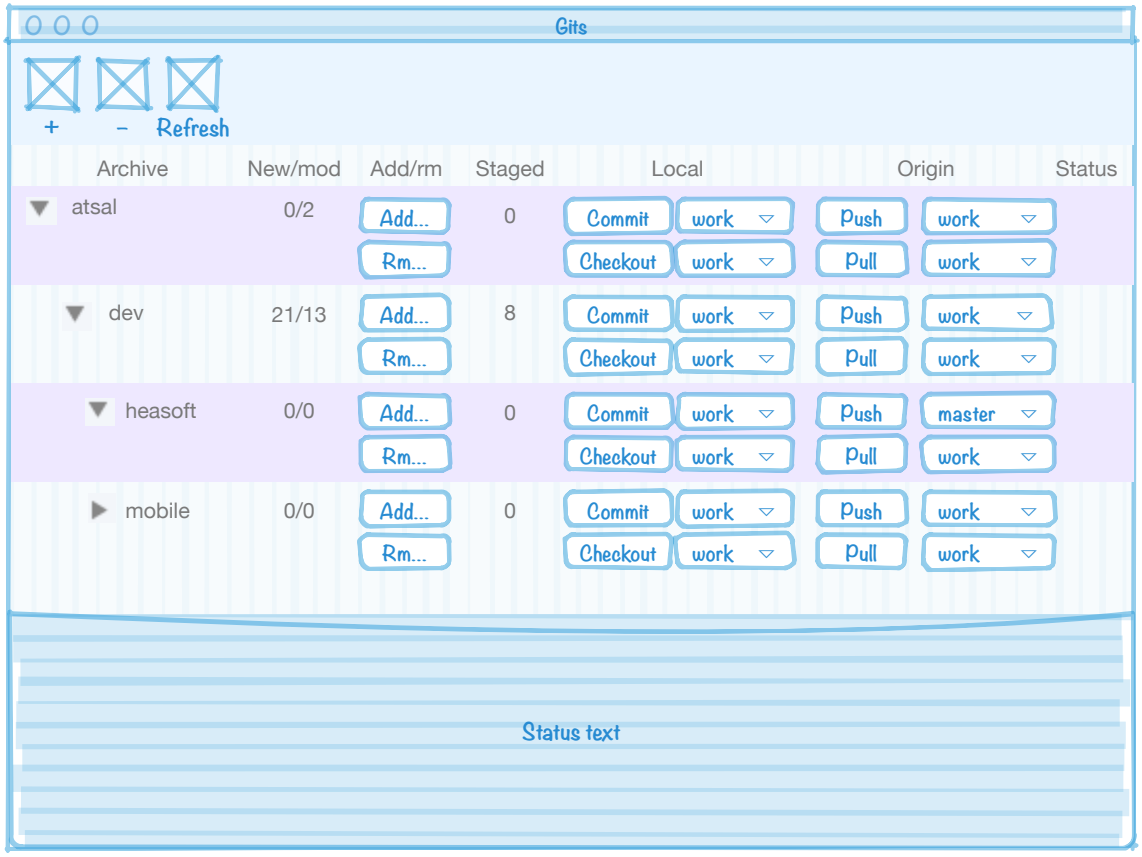


Figure 7. A first cut at a utility to aid development of projects with multiple submodules. For each module, the top row moves work forward, and the bottom row moves it back into the local repository. This is an aid, not a replacement, for the command line and/or other user interface tools.

10 Git Cheat Sheet

This is straight out of the Git book.

Commands	*Description*
`git help <verb>`	Request help on Git function
`git init`	Create a new git archive in the current directory. Initializes an empty archive.
`git add *.c`	Adds files to the staging area in preparation for committing to the archive. (If a file is already staged and you make further changes, just add it again.)
`git rm --cached dontadd.c`	Removes file from the staging area, but does not delete it from the directory.
`rm oldfile.c` `git rm oldfile.c`	Removes file from directory, then from archive. The file is retained in older checkouts, but no longer present in the current version.
`git commit`	Adds all staged files to the archive. Puts up an editor in which you can specify the reason(s) for the commit.
`git commit -m "Commit messages should be more informative"`	Supplies commit message as part of command line instead of in a text editor as above.
`git commit -a`	Like git commit, but commits all modified files, not just those that have been staged.
`git status`	Lists all staged files, modified files, and all files that look like they should be checked in (that is, files that are not exempted in the .gitignore file and are not current part of the archive).
`git mv <oldname> <newname>`	Renames a file. (Important: if you forget to tell Git and simply rename the file directly, Git will notice that the old file is removed and a new one added later, so you can fix this by telling Git to remove the old one and add the new one.
`git log`	Show commit history (many options available)
`git checkout -- <file>`	**Deletes** the current version of the file, replacing it with the most recently committed version from your local archive.
`git fetch origin`	Updates your local archive to mirror the one you cloned from (this is the meaning of "origin.") This changes the archived files, not the files you are presently working on.
`git merge <branch>`	Merges commits from the named branch into the current branch. (See also rebasing.)
`get pull <remote-name>`	Equivalent to a fetch followed by a merge: it makes your local archive match the remote's, then merges those changes into your own current files. The pull command pulls from your currently designated branch. If you omit <remote-name>, it defaults to origin.

`git push <remote-name> <branch>`	Pushes your checked in files to the specified remote's specified branch. If the remote has changed, this command fails; you must pull and merge first.
`git push origin master`	A common case: pushes to the origin archive's master branch. As with the above, this works only if the origin's branch hasn't changed.
`git push --set-upstream origin <branch>`	Pushes current branch to origin's branch, making <branch> the new default for later pushes.
`git fsck`	Check module for corruption.

`git clone --mirror <server>:<archive>.git`
`cd <archive>.git`
`git fetch -q`
	Mirror a git archive for backup. The clone is done only the first time. Subsequently, cd into the archive and do git fetch to update it. Note that mirrors get all the data, but not your git global settings.

11 Appendix 1: .gitignore for XCode Project Files

The following appendix is reproduced from here:

http://www.stackprinter.com/export?service=stackoverflow&question=49478

Git ignore file for Xcode projects

[+474] [15] Hagelin
[2008-09-08 11:07:49]
[xcode git osx version-control gitignore]
[http://stackoverflow.com/questions/49478/git-ignore-file-for-xcode-projects]
Which files should I include in .gitignore when using Git in conjunction with Xcode?

[+334] [2012-08-18 19:22:10] Adam [ACCEPTED]
I was previously using the top-voted answer, but it needs a bit of cleanup, so here it is re-done for Xcode 4, with some improvements.

I've researched *every* file in this list, but several of them do not exist in Apple's official xcode docs, so I had to go on Apple mailing lists.

Apple continues to add undocumented files, potentially corrupting our live projects. This IMHO is unacceptable, and I've now started logging bugs against it each time they do so. I know they don't care, but maybe it'll shame one of them into treating developers more fairly.

If you need to customize, here's a gist you can fork: https://gist.github.com/3786883

```
#########################
# .gitignore file for Xcode4 and Xcode5 Source projects
#
# Apple bugs, waiting for Apple to fix/respond:
#
#       15564624 - what does the xccheckout file in Xcode5 do? Where's the
documentation?
#
# Version 2.1
# For latest version, see: http://stackoverflow.com/questions/49478/git-ignore-
file-for-xcode-projects
#
# 2013 updates:
# - fixed the broken "save personal Schemes"
# - added line-by-line explanations for EVERYTHING (some were missing)
#
# NB: if you are storing "built" products, this WILL NOT WORK,
# and you should use a different .gitignore (or none at all)
# This file is for SOURCE projects, where there are many extra
# files that we want to exclude
#
#########################

#####
# OS X temporary files that should never be committed
#
# c.f. http://www.westwind.com/reference/os-x/invisibles.html
```

```
.DS_Store

# c.f. http://www.westwind.com/reference/os-x/invisibles.html

.Trashes

# c.f. http://www.westwind.com/reference/os-x/invisibles.html

*.swp

# *.lock - this is used and abused by many editors for many different things.
#    For the main ones I use (e.g. Eclipse), it should be excluded
#    from source-control, but YMMV

*.lock

#
# profile - REMOVED temporarily (on double-checking, this seems incorrect; I
can't find it in OS X docs?)
#profile


####
# Xcode temporary files that should never be committed
#
# NB: NIB/XIB files still exist even on Storyboard projects, so we want this...

*~.nib


####
# Xcode build files -
#
# NB: slash on the end, so we only remove the FOLDER, not any files that were
badly named "DerivedData"

DerivedData/

# NB: slash on the end, so we only remove the FOLDER, not any files that were
badly named "build"

build/


#####
#  Xcode private settings (window sizes, bookmarks, breakpoints, custom
executables, smart groups)
#
# This is complicated:
#
# SOMETIMES you need to put this file in version control.
# Apple designed it poorly - if you use "custom executables", they are
#  saved in this file.
# 99% of projects do NOT use those, so they do NOT want to version control this
file.
#  ..but if you're in the 1%, comment out the line "*.pbxuser"

# .pbxuser: http://lists.apple.com/archives/xcode-users/2004/Jan/msg00193.html

*.pbxuser

# .mode1v3: http://lists.apple.com/archives/xcode-users/2007/Oct/msg00465.html

*.mode1v3
```

```
# .mode2v3: http://lists.apple.com/archives/xcode-users/2007/Oct/msg00465.html

*.mode2v3

#   .perspectivev3:   http://stackoverflow.com/questions/5223297/xcode-projects-
what-is-a-perspectivev3-file

*.perspectivev3

#    NB: also, whitelist the default ones, some projects need to use these
!default.pbxuser
!default.mode1v3
!default.mode2v3
!default.perspectivev3


####
# Xcode 4 - semi-personal settings
#
#
# OPTION 1: ----------------------------------
#     throw away ALL personal settings (including custom schemes!
#     - unless they are "shared")
#
# NB: this is exclusive with OPTION 2 below
xcuserdata

# OPTION 2: ----------------------------------
#     get rid of ALL personal settings, but KEEP SOME OF THEM
#     - NB: you must manually uncomment the bits you want to keep
#
# NB: this *requires* git v1.8.2 or above; you may need to upgrade to latest OS
X,
#    or manually install git over the top of the OS X version
# NB: this is exclusive with OPTION 1 above
#
#xcuserdata/**/*

#     (requires option 2 above): Personal Schemes
#
#!xcuserdata/**/xcschemes/*

####
# XCode 4 workspaces - more detailed
#
# Workspaces are important! They are a core feature of Xcode - don't exclude
them :)
#
# Workspace layout is quite spammy. For reference:
#
# /(root)/
#   /(project-name).xcodeproj/
#     project.pbxproj
#     /project.xcworkspace/
#       contents.xcworkspacedata
#       /xcuserdata/
#         /(your name)/xcuserdatad/
#           UserInterfaceState.xcuserstate
#     /xcsshareddata/
#       /xcschemes/
#         (shared scheme name).xcscheme
#     /xcuserdata/
#       /(your name)/xcuserdatad/
#         (private scheme).xcscheme
#         xcschememanagement.plist
```

```
#
#

####
# Xcode 4 - Deprecated classes
#
# Allegedly, if you manually "deprecate" your classes, they get moved here.
#
# We're using source-control, so this is a "feature" that we do not want!

*.moved-aside

####
# UNKNOWN: recommended by others, but I can't discover what these files are
#
# ...none. Everything is now explained.
```

I assumed you meant a gist - seeing as the official github project for .gitignores was unmaintained and refusing submissions last time I looked (IIRC there were 200 ignored pull requests, and a huge number of Issues that were being ignored) - **Adam**
(1) Ok, I just noticed a problem with this. I was doing some git acrobatics, and when I checkout out back to master and applied my stashed changes, I had lost my saved build schemes! fortunately, I had backed them up, just in case... but the solution is to ignore a little more specifically inside the xcuserdata directory. I changed `xcuserdata` to `xcdebugger` and `UserInterfaceState.xcuserstate`, which are really the offensive ones to commit. - **samson**
I also suggest to add .svn for projects that work with both source control systems - **Michael Kessler**
@samson - " I had lost my saved build schemes! " -- argh! That's what I was trying to avoid! Sorry :(. I've added an exception for "xcschemes" which seems to be what my Xcode is using - **Adam**
That's better than my approach. I'll try that, thanks! - **samson**
@MichaelKessler I can see that helping for some projects, but using one project with two SCM's sounds very unusual (dangerous in many ways). I've been on projects where we did it deliberately - but 9 times in 10 when I've seen it, it was an accident. In most cases, I think it's a major bug / mistake to have two SCM's versioning one set of files, so I'd rather leave the .svn folder in there -- for most people, they'll see all the .svn files appear and go "WTF?" and realise their mistake. - **Adam**
@Adam, In general you are right - there is almost no reason to work with 2 SCMs on one project. But I have personally used this approach several times. The most common (for me) case was when a client gives me his existing project with SVN. I work with GIT. I think that there was no project where SVN worked 100% well for me - there are always problems with it. This is why I always create my own git repository for all the ongoing commits and ignore the .svn folders. Eventually I commit all the changes to client's SVN and forget about it. - **Michael Kessler**
I added cocoapods to the mix - **cannyboy**
"the syntax for .gitignore is very hard to understand" you serious ?? its an easy pattern system it took about 2 minutes to understand it.. iv'e seen much worse things... - **martyonair**
@martyonair - try googling, and see how many people discover (belatedly) that gitignore does something very different than what they thought it would do. For a trivially simple file-format, it's surprisingly easy to misunderstand / use incorrectly (since then, I've seen a lot of projects that are misconfigured too, and their authors don't realise it) - **Adam**
(4) You shouldn't be ignoring `*.lock` or `Podfile.lock` (never mind the redundancy). You want the exact same versions installed in all workspaces, you don't want the "latest

version". - **tvon**
I have removed the Podfile part. I didn't add that originally, SO says someone else added it and I carelessly copy/pasted it into the gist. My apologies for any/all confusion and misunderstanding. I really dislike the way StackOverflow lets anyone edit your answers :(. - **Adam**
@Adam Thanks, though the `*.lock` line will still cause problems. If you are using bundler then a Gemfile.lock is important (and I'm not sure what that line is trying to do anyway, I've never had random `.lock` files show up). - **tvon**
(1) There's now an explanation line for EVERYTHING, line by line. This should make it much clearer, and make it easier to customize for your own projects. - **Adam**
@Adam: Thanks for this! Notice that the gist link still points to v2.0 instead of v2.1. - **Ricardo Sánchez-Sáez**
(9) Please update for Xcode 5! Thanks! - **Hyperbole**
(4) Hi! update for Xcode5: Just add `*.xccheckout` to this file. - **skywinder**
(1) @skywinder - do you have a reference to Xcode docs on this? Ideally a URL to a paragraph in Apple's docs that states what xccheckout files contain. I'm being hyper-cautious about adding more files to gitignore - one mistake, and we could cause someone to lose their important work! - **Adam**
(2) @Adam As I can see, this file contains VCS metadata, and should therefore not be checked into the VCS. No, there no mentions on `developer.apple.com` about `xccheckout`. But on official github page, this file included already in the gitignore file. `https://github.com/github/gitignore/blob/master/Objective-C.gitignore` - **skywinder**
(1) @skywinder According to this answer on SO you may be wrong: stackoverflow.com/a/19260712/153422 - that file is important. I will NOT ignore files until you can prove they are irrelevant - if we get it wrong, the damage is great. The "official github page" IS CONSISTENTLY WRONG (you really shouldn't use it), and is definitely NOT an argument for doing it right! - **Adam**
(1) 6 weeks later, and Apple still hasn't replied to my request for docs on "what" xccheckout file contains :(. I guess we won't be getting any docs. - **Adam**
Update: It's been 4 *months* and ... Apple still hasn't responded to the bug report. Recommendation: don't file bug reports with Apple, it's a waste of your time :(- **Adam**
1

[+247] [2008-09-08 11:14:32] Hagelin
Based on this guide for Mercurial [1] my .gitignore includes:

```
.DS_Store
*.swp
*~.nib

build/

*.pbxuser
*.perspective
*.perspectivev3
```

I've also chosen to include:

```
*.mode1v3
*.mode2v3
```

which, according to this Apple mailing list post [2], are "user-specific project settings".

And for Xcode 4:

```
xcuserdata
```

[1] http://boredzo.org/blog/archives/2008-03-20/hgignore-for-mac-os-x-applications
[2] http://lists.apple.com/archives/Xcode-users/2007/Oct/msg00465.html

(50) I don't particularly like the *.pbxuser/*.perspective/*.perspectivev3 patterns. I much prefer the following *.xcodeproj/* !*.xcodeproj/project.pbxproj That ignores everything inside a *.xcodeproj except the project.pbxproj. - **Kevin Ballard**
(5) I do not ignore *.pbxuser, *.perspective and *.perspectivev3 because I like to keep those settings back when I clone my repository. - **lajos**
(3) Use build/ to exclude only directories named build in case you might have a script or something named build that you don't want to ignore. - **nicerobot**
(1) I like to leave in build/Release-iphoneos so I have a copy of every released device app I seed out to people. Patterns to add would be build/Debug-* and build/*-iphonesimulator . - **Ryan McCuaig**
(1) I'm not sure if I'd keep this in my repository. Perhaps a better solution would be to keep your build folder outside of your project in some central location (e.g. /builds/projectname/**) and keep the /builds directory backed up with something like GetDropbox? - **Luke Redpath**
(7) Also you might want to add that you can make a "global" gitignore file like this: git config --global core.excludesfile ~/.gitignore - **Jess Bowers**
(1) I agree with Luke regarding his hesitation to keep the build directory version controlled in the same repository. In any case, if you often integrate external libraries and projects into your xcode projects, you should configure your build directory to be common anyway. I typically keep mine in /Users/Shared/<username>/Products - **Michael G. Emmons**
(46) I'd like to caution everyone who added .gitignore file **after** they have committed the project: those files you ignore are still being tracked. You'll have to remove them from git manually using `git rm --cached <files>` - **pixelfreak**
(2) Using xcuserdata is bad as it prevents your xcschemes directory from being version controlled. - **Erik**
(3) In Xcode 4, it appears that the only files to worry about are xcuserdata directories and .DS_Store, and the rest aren't needed in .gitignore. I have an active project that's never been opened in XCode 3, but only in v4, and none of the other files were present in my file hierarchy. It uses Storyboard instead of .xib's for the views. My config is normal too, i.e. no "weird" customizations like moving the build directory from its default location. - **curtisdf**
(2) According to stackoverflow.com/a/9552687/599884 and github.com/github/gitignore/blob/master/Objective-C.gitignore it seems to be a good idea to also ignore xcworkspace - **Christoph**
(3) The comment @KevinBallard is extremely useful, except that it contains a small oversight.
`*.xcworkspace/* !*.xcworkspace/ !*.xcworkspace/contents.xcworkspacedata` works, since it first blacklists every **file** in the project folder, then whitelists the folder itself and then whitelist the project file. This way, the entire folder will not be blacklisted, which causes git to skip it entirely. - **SpacyRicochet**
(19) @SpacyRicochet: Comment formatting has apparently changed since I wrote the comment. Hence the italics. My pattern is supposed to look like *.xcodeproj/* !*.xcodeproj/project.pbxproj. Of course, these days you do need to adjust it for workspaces. - **Kevin Ballard**

(3) Everyone: please upvote @KevinBallard comment on May 10. It's hidden by default and his original comment is incorrect now. Save another dev some grief. - **Ben Dolman**
2
[+45] [2010-10-13 14:09:21] Vladimir Mitrovic
For Xcode 4 I also add:

```
YourProjectName.xcodeproj/xcuserdata/*
YourProjectName.xcodeproj/project.xcworkspace/xcuserdata/*
```

(68) If you just add `xcuserdata`, then that takes care of both. - **MattDiPasquale**
For some reason just adding xcuserdata without the prefix didn't work for me. I thought it should, though. Odd. - **badcat**
3
[+43] [2008-12-08 10:42:27] Abizern
Regarding the 'build' directory exclusion -

If you place your build files in a different directory from your source, as I do, you don't have the folder in the tree to worry about.

This also makes life simpler for sharing your code, preventing bloated backups, and even when you have dependencies to other Xcode projects (while require the builds to be in the same directory as each other)

You can grab an up-to-date copy from the Github gist https://gist.github.com/708713

My current .gitignore file is

```
# Mac OS X
*.DS_Store

# Xcode
*.pbxuser
*.mode1v3
*.mode2v3
*.perspectivev3
*.xcuserstate
project.xcworkspace/
xcuserdata/

# Generated files
*.o
*.pyc


#Python modules
MANIFEST
dist/
build/

# Backup files
*~.nib
```

(7) I do have the build folder outside of the project folder, but when other users build the project, it by default is recreated in the project- so I found that adding it to the ignore file is a better solution, otherwise it gets readded in their commits. - **lajos**
4

[+21] [2009-06-28 20:04:44] program247365
I included these suggestions in a Gist I created on Github:
http://gist.github.com/137348

Feel free to fork it, and make it better.

(5) Also one of the Github guys has collected some .gitignore files. Here is the Objective-C specific one- github.com/github/gitignore/blob/master/Objective-C.gitignore - **program247365**
Also the Thoughtbot folks came up with this project - github.com/thoughtbot/liftoff which will add a sane default .gitignore files, see their blog post on it: robots.thoughtbot.com/post/33796217972/... - **program247365**
5
[+8] [2010-02-01 21:33:00] tbarbe
Heres a script I made to auto create your .gitignore and .gitattributes files using Xcode... I hacked it together with a few other people's stuff. Have fun!

Xcode-Git-User-Script [1]

No warranties... I suck at most of this - so use at your own peril

[1] http://github.com/tbarbe/Xcode-Git-User-Script

6
[+7] [2013-04-17 13:57:43] Wanbok Choi
I'm using both AppCode and XCode. So `.idea/` should be ignored.

append this to Adam's `.gitignore`

```
####
# AppCode
.idea/
```

7
[+6] [2012-08-04 18:30:45] Eric
The people of GitHub have a pretty exhaustive and efficient .gitignore file for Xcode projects: Objective-C.gitignore [1]

[1] https://github.com/github/gitignore/blob/master/Objective-C.gitignore

(4) This has already been posted to one of the answers above. I found it to be: incorrect, questionably supported (more than 100 outstanding pull requests!), and undocumented. The fact that it's "incorrect" is the worst of all; they have made an ignore that only works for a narrow set of uses and haven't explained what or why! Hence: my answer above, which corrects their bugs AND explains what's being done and why, so you can make educated decisions on a project-by-project basis (on a new project, I sometimes forget why some of the items are in there - the comments help me decide :)) - **Adam**
8
[+5] [2008-09-08 17:51:38] Dave Verwer
Mine is a .bzrignore, but same idea :)

```
.DS_Store
*.mode1v3
*.pbxuser
*.perspectivev3
*.tm_build_errors
```

the tm_build_errors is for when I use TextMate to build my project. Not quite as comprehensive as Hagelin but I thought it was worth posting for the tm_build_errors line.

9
[+4] [2013-10-16 07:00:40] Wanbok Choi
For XCode 5 I add:

```
####
# Xcode 5 - VCS metadata
#
*.xccheckout
```

From Berik's Answer [1]

[1] http://stackoverflow.com/a/18448100/1602311

10
[+3] [2011-11-16 20:52:58] Paul Cezanne
I found that projects that included other project became broken when I include the xcworkspace files in my list of ignores.

11
[+2] [2009-11-12 16:03:23] Steve M
make them Global and not at the directory level so your not pushing them to others..

12
[+2] [2010-01-13 14:12:40] Hoang Pham
http://shanesbrain.net/2008/7/9/using-xcode-with-git

13
[+2] [2012-09-25 21:38:58] user1524957
I've added:

```
xcuserstate
xcsettings
```

and placed my .gitignore file at the root of my project.

After committing and pushing. I then ran:

```
git rm --cached UserInterfaceState.xcuserstate WorkspaceSettings.xcsettings
```

buried with the folder below:

```
<my_project_name>/<my_project_name>.xcodeproj/project.xcworkspace/xcuserdata/<m
y_user_name>.xcuserdatad/
```

I then ran git commit and push again

Did you add it also? Or is this just all you do? - **hakre**
(1) Yes, I added both but xcusersate was the main offending file. Adding that was the only way I could push my code remotely. Otherwise I was stuck in a feedback loop that required commit before push. So you commit, then Xcode 4.5 would ask you to commit again and you are never able to push because the pre req is committing. - **user1524957**
14
[0] [2013-09-04 08:03:12] Basil Abbas
We did find that even if you add the .gitignore and the .gitattribte the *.pbxproj file can get corrupted. So we have a simple plan.

Every person that codes in office simply discards the changes made to this file. In the commit we simple mention the files that are added into the source. And then push to the server. Our integration manager than pulls and sees the commit details and adds the files into the resources.

Once he updates the remote everyone will always have a working copy. In case something is missing then we inform him to add it in and then pull once again.

This has worked out for us without any issues.