Version 0.1

Organizing FTC Programmers with GitHub and OOP Libraries

Benjamin Broce

Watauga Robotics

https://wataugarobotics.wixsite.com/home

https://github.com/wataugarobotics

Introduction

As the Watauga Robotics programming teams have exploded in numbers over the past few years and our FTC teams continue to acclimate to the possibilities of the Android control system, we have identified a need for systematic code collaboration, abstraction (especially for beginners), publishing, and SDK (Software Development Kit) management. We decided to address these needs with some key expansions on the established solution of migrating our codebase(s) to a networked VCS (Version Control System) - namely Git and GitHub - and by building a set of polymorphic Java libraries to promote abstract thinking in our Op Mode design. This document serves as an overview of our approach.

As an organization, we believe in the capacity for collaborative learning that FOSS (Free and Open Source Software) can facilitate among FIRST participants, and would encourage every team to consider not only publishing their code, but documenting and openly licensing[1] it as well.

[1] GitHub resource for learning about open licenses: https://choosealicense.com/

Git Basics

Note: This section is not intended as a Git/GitHub tutorial, but rather an introduction to what each is, how they can be useful, and some basic methodologies behind their workflow. To begin learning how to use them, see the references[2,3] below.

Git is an extremely powerful and prominent tool in modern software development which enables scaled collaboration and versioned backup for directories of files called repositories. As alluded to by the XKCD comic right[4], there is a learning curve and sort of novice mystery associated with Git (most obviously due to it being a terminal/command line application), but establishing even a basic workflow within it can quickly accelerate a team's productivity.

GitHub is an online service that hosts Git repositories, allowing for asynchronous collaboration and widespread software publishing (nearly all FTC programmers are somewhat familiar with the site, given that the official FTC SDK is developed and released there).

[2] GitHub's official tutorial: https://guides.github.com/activities/hello-world/

[3] GitHub's interactive Git tutorial: https://try.github.io/

[4] There's an XKCD for that: https://xkcd.com/1597/

Due to the prolific learning resources and technical advantages of the Git/GitHub system, many FTC teams have decided to migrate their codebases to GitHub, most often by creating an Organization for their team on the site and creating a repository with either the TeamCode directory or a copy of the entire ftc_app SDK from FIRST. Their workflow would then ideally consist of:

1. **Pulling** the latest collaborative update from GitHub
2. Creating a **branch** from the **master** repository branch in order to add a set of features/updates (this can be avoided by just using master, but is recommended to avoid master containing a broken or untested state)
3. **Adding** and **Committing** each set of related changes
4. **Pull Requesting** and **Merging** tested branch modifications into master

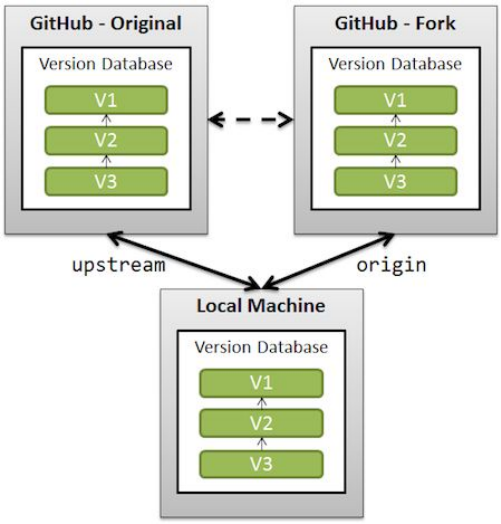
This allows for a highly organized and accessible codebase (and is the preferred method for many styles of development in general), but does offer some limitations:

- Any official updates to the underlying ftc_app must be merged manually, likely requiring that the TeamCode folder be backed up and re-inserted into the new SDK
- It is nearly impossible to maintain a good permissions/management structure for sister teams preferring to use multiple TeamCode folders in the same ftc_app (Team[team #] method)
- Common libraries (for sister teams or open sourcing) are difficult to maintain independent of a single team

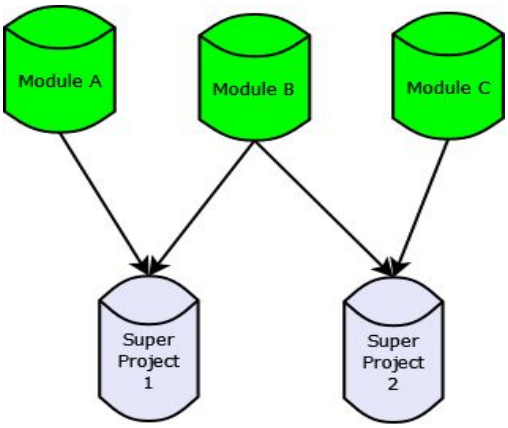
As our teams could see real benefit from solving each of these problems (considering we have two FTC teams for which we would like to combine library development and code management efforts), we decided to research more advanced git workflows to find the right combination here.

Forks and Subtrees

Forking is a GitHub tool that is essentially the remote equivalent of **cloning** a repository locally (copying its contents as well as all of its version control history). Forking the official ftc_app repository generates an updateable copy of the original as a repository within a team's GitHub organization. This means that if the copy is cloned onto a programmer's machine and used as the core of a team's codebase, they can then add a **remote** (a URL that points to a remotely hosted repository) called **upstream** (the original ftc_app in this case) in addition to **origin** (the remote copy which was cloned), to make updating the entire SDK to the latest version without affecting TeamCode as simple as **merging** with the upstream master branch.



Git **subtrees** were essential to how we solved the management issues of multiple TeamCode directories and a common library. These structures allow the entire source of one repository to be nested inside another, with controls in place that allow for pulling



from and pushing to the outside repository in place (all with a much simpler command scheme and work flow than the infamous alternative: submodules).

These two concepts came together to form our teams' overall structure: A fork of the official FIRST ftc_app in our GitHub organization, alongside TeamCode repositories for our sister teams and our libraries repository, which are referenced within the fork as subtrees. This allows us to maintain a unified, futureproof codebase with an easily updatable SDK, safe versions of current TeamCode directories (as only tested code is pulled into the competition-ready master branch of the codebase from the TeamCode subtree repositories), and a separately maintained, shareable libraries directory. Our new workflow now revolves around intra-team development within this ecosystem (with subtree pushing and pulling replacing the normal push and pull commands), minimizing the repository maintainer's duties to occasionally updating the group SDK and pulling in the latest stable branches from the necessary repositories for convenience or competition.

A Case for Bash and the CLI

Git is natively a CLI (Command Line Interface) application, and is what makes all of the above work possible. The particular terminal shell for which Git is designed is called **Bash**, a tool which has become the standard for many of the operating systems in the Unix family (most prominently Linux and macOS). This means that running the bash commands required to accomplish many of the above tasks requires either one of these systems or a Bash emulator (I recommend the git-bash tool included with the Windows git installation tool[5] if the others are not available). Though GUI (Graphical User Interface) extensions for Git do exist, they limit another

[5] Download Git for your System: https://git-scm.com/downloads

important factor in designing a complex structure as above: scripting. Though updating the SDK and pulling from or pushing to a subtree are relatively quick processes, they still require in-depth knowledge of Git that not all team programmers may have, prompting us to make some simple applications (“scripts”) to run the necessary 3-6 git commands (which can be found on our GitHub page). Because Bash scripting is so powerful in cutting down on time-consuming actions (and because much of the learning material on Git is directed toward the CLI), I would highly recommend that you avoid such GUI interfaces when learning to navigate the world of Git.

Coming Soon:

- Object Oriented Design
- FTC Library Design