**Activity 3:**

**Making Changes and Branches**

This activity discusses making changes to local files, remote copies, and the main project’s repo. The activity also covers the concepts of branches, pull requests, and merging.

Please use this dictionary whenever needed to understand ambiguous terms: <https://www.britannica.com/dictionary>

## **Content Learning Objectives**

*After completing this activity, students should be* able *to:*

* Describe the different stages of making changes in a project.
* Understands the pull request, push and merge terms.
* Describe the concept of branches and commits.
* Describe the concepts of using branches with git.

## **Process Skill Goals**

*During the activity, students should make progress toward:*

* Teamwork
* Information processing

## **Team Roles**

* *Before you start, make sure everyone in your team has a new role (not the same role as the last activity).*
* *If you have three people, combine the Manager and Reflector roles.*

*Record role assignments here.*

|  |  |
| --- | --- |
| Manager |  |
| Presenter |  |
| Recorder |  |
| Reflector |  |

**Model 1: Making Changes**

## **Information**

When a developer is working on the code, they will be making changes on their local repo. They basically open up a code file in their editor, make some changes, save the file. (In the figure below, this is represented by the change of the words from white to green)



The code is in a version-controlled repository. So, making changes isn’t quite as simple as opening the file, editing, saving. Version control maintains a complete history of all changes. So, ensuring that that history is maintained is one of the things we’ll have to deal with.

## **Instructions and Questions (15 min)**

**Step 1: *Push to Origin***

After making the changes you like on the file locally. Maybe you want to upstream them to give them back to the project. To do that you’ll need to get them to cloud so you will need to move those changes to your remote repo – i.e. to your origin and eventually move them to the main repo also – the upstream. The first step is to push the changes to your origin. (In the figure below, the change to green in the origin reflects your local changes being copied there).



1. ***Why do you think we push to the origin?***

* Most importantly, you are not allowed to write to the upstream repo, remember only maintainers are allowed to do that.
* The upstream repo cannot see your local copy because it is on your machine.
* You might erase them, or turn your machine off, or move it to some other location. So pushing puts your changes into the hosting service
* Then the upstream will be able to find them – once you tell the upstream that they exist!

1. ***Where are the changes now? (local copy, remote copy, or main repo)***

**Step 2: *Making a Pull Request***

Now your changes are in the hosting service. So now you need to tell the maintainers at the upstream that you have some changes you’d like to contribute. You do that by making a Pull request. The pull request is essentially a message to the upstream maintainers. The pull request tells them about the changes you want to contribute. The maintainers and the upstream repo can see your changes because they are also on the hosting service.

****

1. ***Why pull request is important?***
2. ***Are the changes in the main repo now?***

**Step 4: *Merge into Upstream***

****

If the project maintainers/leaders like the changes you proposed in your Pull Request. They merge your proposed changes into the upstream. That is, they combine your code with the code in the main repo

1. ***Why merge into Upstream is important?***

## **Model 2: *Branches and Commits***

## **Information**

Time

**main**

Commits

Can also think of a **commit** as

**a copy of all the files**

in the repo at a given point in time.

* **Branch**: A branch is a named sequence of commits.
* **Commit**: Each commit describes a set of changes to the project.
  + The changes made.
  + Who made them.
  + When they were made.
  + A short description of the changes and their purpose.
  + A unique identifier.

To understand how version control (e.g. git) tracks changes and keeps a history, we need to know about branches and commits. The contents of a repository are organized into branches. You can think of this sort of like track changes in Word or Google Docs if you have used that.

**Branches grow over time:**

* Developers add new features, fix bugs, revise code, delete unused things, etc.
* Each change is described by a new commit that becomes part of the branch.
* It is this sequence of commits that documents the history of the project.
* Another way to think of it is that you could start from an empty project, then go along and apply each commit in order. When you have applied all of the commits you have the current state of the project.

## **Instructions and Questions (10 min)**

## **Main and Feature Branches:**

Time

**main**

newColorScheme

newColorScheme2

nullPointerFix

**RULE:** **Only commit to feature branches.** Maintainers will merge accepted changes into the main branch.

## **Main Branch**: the designated starting point for new work.

## **Feature Branch**: a copy of the main branch on which a new feature, bug fix, or other changes are made.

## When you want to do some work (e.g. add a feature, fix a bug, etc), you create a new branch. The new branch is like a full copy of the repository, you then add commits for the work you want to do to the new branch. These branches on which you do work are generically called feature branches. But you will give them specific names that are descriptive of what they do. (e.g. newColorScheme – to develop a new color scheme for the user interface.).

## You may be working on multiple different tasks at the same time. When doing so you can have multiple feature branches simultaneously. e.g. stop working on the new color scheme to fix a null pointer exception that was discovered. You can even make a branch of a branch. e.g. started working on the new color scheme but after a while decided you wanted to try it a different way. Instead of deleting what you’ve done, you can go back and create a new branch from any commit and try something different. That way if you later decide the first way actually was better you can jump right back to it. You can switch back and forth between branches to work on different tasks to try different approaches

1. ***Are there names for Main branch? (you can search the Internet)***

## the master branch – outdated terminology

## the dev branch indicating its where new development is occurring.

## there is nothing special about the name – it is just a convention.

## different projects will do it differently.

## ***What is the maximum number of branches allowed in each project?***

## ***Discuss as a team the advantages of using branches. You may support your answer with examples.***

# **Model 3: *Branches and Git***

## **Information**

***At the end of this model, students are requested to re-watch the video from Activity 1 FOSS and answer questions regarding it. Instructors have the option of:***

* ***Post the video as homework before class starts***
* ***Play the video in the classroom before students start working on the model.***
* ***The video duration is 5:41 sec.***

***Repos and Branches***

Let’s now see how the idea of branches translates into work with GitHub and git.

When you fork and clone a repository you get a copy of its main branch. You can get copies of the other branches in the repo also if you request them. The main branch includes all of the commits and all of the history of the changes in that branch. Your origin and local clone repos contain a copy of the main branch and its commits from the upstream.



***Branch/Repo vs Local Files***

So, now you have your local repository. It contains the main branch copied from the upstream but the commits in there are fixed, they are part of the project history. So, you can’t change them. Instead, what happens is that you have a yet another copy called **your local files.**



## **Instructions and Questions (20 min)**

***Step 1: Create Feature Branch***



As a RULE, you will not work on code in the main branch. Look at the figure above and answer the questions.

1. ***Instead of working in the main branch, what you have to do? And why?***

* create a new feature branch in your local repo (Note: there are no commits in the feature branch yet)
* that is because you haven’t made any changes
* and you haven’t committed them to this branch yet.

**Step 2: *Switch to Feature Branch***

Before you can commit any changes to this branch you need to tell git that you want to use it.

This is called switching to a feature branch. Switching to a feature branch makes it the target for future commits.



1. ***From looking at the above figure, what do you notice about the local files? At what stage are they?***

* the Local Files are still green
* They still match the last commit from main here.
* This is because we have not yet changed them.

**Step 3: Local Files: *Edit Local Files***

Now when you edit your local files, they will be become different from the last commit (here they are colored blue to indicate a set of changes.)

****

1. ***Note that the commit in the main branch from which they were copied didn’t change, it is still green. Why?***

* It is part of the project history.

**Step 4: git: *Stage Changes and Commit***

****

When you have completed some work, you need to commit it to the feature branch.

Committing a set of changes to a feature branch is a two-step process:

* First, you add the files that you have changed to the Staging Area using the stage command. The set of blue changes is on the stage.
* Then you commit those changes to the feature branch. The set of blue changes is added to the feature branch.

Example:

* You write some code to do something (one commit)
* Then you write code to test it (another commit)

1. ***Why do you think it’s done in two steps?***

* It may take a number of logical changes to implement a feature
* You could do it all in one big commit.
* But better to stage and commit smaller changes
* That way each change has its own record in the project history.

**Step 5: Edit/Add/Commit Cycle**

****

Can repeat the edit, stage, commit cycle as many times as is necessary.

Try not to think of a commit as a save. When you save, it changes your Local Files, you should do that constantly!

Make a commit when you have completed a nameable unit of work.

* I.e. when you complete some cohesive sub-task. E.g. adding a text field, writing a test, etc. This ensures that you will be able to give every commit a meaningful message. If you can’t think of a short description of what you’ve done, maybe it’s not a commit.

1. ***In this example, there are multiple parts to implementing a feature; adding the text field for name, adding a drop down for class year and adding a button for submit. How many commits should be there? And why?***

three

**Step 6: Push Branch to Origin**

Now in order to upstream. You need to get your changes into the cloud, that way you can ask the upstream maintainers to consider them. To do that, you push your branch to your origin. this makes a copy of your branch and any commits it contains to your GitHub space.

****

1. ***Where are the changes now? (local copy, remote copy, or main repo)***

**Step 7: Make Pull Request**

Once your branch is in the cloud (I.e. on GitHub), you can notify the upstream of your changes to ask them to consider merging them into main branch. This is called a Pull Request.

****

1. ***Watch again “Open Source Basics” video by Sarah Moyle for Intel Software (the same video you watched in Activity 1).*** [***https://www.youtube.com/watch?v=Tyd0FO0tko8***](https://www.youtube.com/watch?v=Tyd0FO0tko8)

***Then, answer the questions:***

* 1. ***Who wanted to make changes to the recipe and what were the changes?***
* Aunt Maria wants to add chocolate chips.
* Uncle Miles wants to add nuts.
  1. ***Describe how the situation in the video is related to the concepts of making changes, branches and pull requests.***
  2. ***From the video, where the merge into Upstream happened?***
* At Aunt May’s main recipe.

**Team’s Reflection (5 min)**

* *Reflector:* lead the discussion to review the team’s performance during the activity. The team must discuss two positive points and two areas for improvement for the next activities.
* *Recorder:* type the discussion’s summary.
* *Reflector:* be ready to present the points when asked.

## **References & Resources**

* <https://gitlab.com/hfossedu/kits/GitKit>
* <http://foss2serve.org/index.php/Git:_Git_Intro_Activity>
* <http://foss2serve.org/index.php/Git:_GitHub_Issues_and_Pull_Requests>
* <http://foss2serve.org/index.php/Git:_GitHub_Workflow_Activity>
* <http://foss2serve.org/index.php/Intro_to_GitHub_(Activity)>
* <http://foss2serve.org/index.php/Version_Control_(Activity)>
* <http://foss2serve.org/index.php/Work_Locally_with_Git_from_the_Command_Line_(Activity)>

|  |  |
| --- | --- |
|  | *© 2010-2023 by Grant Braught,, Ali Al-Faris. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.* |