**Activity 2:**

**Version Control Systems and Git**

This activity discusses Git concepts, navigations, states and save operations.

## **Content Learning Objectives**

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

* Describe the importance of & issues in sharing & managing source code.
* Describe the version control system (VCS), and its key operations.
* Define Git and describe the different process of working on a program through GitHub/Gitlab

## **Process Skill Goals**

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

* Teamwork
* Information processing

## **Team Roles**

*Record role assignments here.*

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

**Model 1: Saving Backups**

## **Information**

When we write software, we create **many files**, and organize them in a directory or folder.   
Larger projects get spread across more files and directories.

However, in many situations we might need **more or different copies** of our files. For example:

1. Your computer (with all of your files) stops working or is stolen.
2. Your room or building is destroyed by fire, flood, or other disaster.
3. You need the exact version of a file (or set of files) from a day, week, or year ago.

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

* + - 1. To avoid problems with files that are lost, corrupted, or that have changed a lot, we should save a **backup copy** of every file. We could make a backup on different schedules, such as: every **hour** or so, every **day**, or every few **weeks** (or months). Which backup **schedule**:

|  |  |  |
| --- | --- | --- |
|  |  | **Hourly / Daily / Weekly** |
| **a.** | Requires the most time & effort? | **hourly** |
| **b.** | Requires the least time & effort? | **weekly** |
| **c.** | Is least likely to lose important work? | **hourly** |
| **d.** | Is most likely to lose important work? | **weekly** |
| **e.** | Is most likely to be skipped or forgotten when you are busy and under pressure? | **hourly** |

* + - 1. We could store backups in different places, such as: a local **device** (e.g. desktop, laptop, USB drive), **shared storage** (e.g. network drive, cloud storage), or a **physically secure** site (e.g. safe, bank vault). Which backup **location**:

|  |  |  |
| --- | --- | --- |
|  |  | **Device / Shared / Secure** |
| **a.** | Requires the most time and effort? | **secure** |
| **b.** | Is most risky if your device is broken or stolen? | **device** |
| **c.** | Is least risky if your device is broken or stolen? | **secure** |
| **d.** | Is easiest to share with other people? | **shared** |
| **e.** | Is most vulnerable to hackers? | **shared** |

* + - 1. Summarize **key insights** about when and where to store backups. Identify other criteria that might affect this decision.

#### **Backups that are more frequent reduce risk of loss, but are more work and more distracting.**

#### **Backups on local devices are easiest but least reliable.**

#### **Secure backups are safest but least convenient.**

#### **Shared storage is convenient for sharing but vulnerable to hackers, etc.**

#### **Use multiple options for redundancy.**

#### **Preserve file structure, not just file content.**

## **Model 2: Multiple People**

## **Information**

Projects involve teams of **people** (often in different places). If two (or more) people try to edit a file at the same time.

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

Will the chance of conflicts go **up**, **down**, or stay the **same** if we:

|  |  |  |
| --- | --- | --- |
|  |  | **Up / Down / Same** |
| a. | **Decrease** the size of the team. | **down** |
| b. | **Decrease** the total LoC (lines of code) in the project. | **up** |
| c. | **Split** the project into more, smaller files (same LoC). | **down** |
| d. | Save files **more often** (e.g. every 5 minutes, rather than every 30 minutes). | **down** |

It is easy (with software) to track who edits which files. When two (or more) people try to edit a file, there are several general strategies:

* allow the first to edit the file, and **prevent** others from editing at the same time.
* **notify** everyone who tries to edit the file, so they can avoid or resolve problems.
* **ignore** the danger, and let people to find and fix problems on their own Which strategy:

|  |  |  |
| --- | --- | --- |
|  |  | **Prevent / Notify / Ignore** |
| a. | Requires the **most work** to implement. | **notify** |
| b. | Is **least convenient** for teams (people must wait for others to finish work). | **prevent** |
| c. | Is **most convenient** for teams. | **notify or ignore** |
| d. | Is most likely to lead to **problems**. | **ignore** |
| e. | Is usually called pessimistic locking.  *(Not allowing simultaneous updates to records. As soon as one user starts to update a record, a lock is placed on it. Other users must wait until the first user has finished committing their changes, thereby releasing the record lock. Only then can another user make changes based on the previous user's changes.)* | **prevent** |
| f. | Is usually called optimistic locking.  *(Allows multiple users to attempt to update the same record without informing the users that others are also attempting to update the record. The record changes are validated only when the record is committed. If one user successfully updates the record, the other users attempting to commit their concurrent updates are informed that a conflict exists.)* | **notify** |
| g. | Is usually called overly optimistic locking.  *(Assumes that the collision will never occur, therefore the system does not try to About the issue. It is important to understand that this system can be only used in a single-user mode)* | **ignore** |

Summarize **key insights** on how to **reduce file conflicts**.

#### **Smaller teams often have fewer problems - adding people can add problems.**

#### **More, smaller files reduce the chance of conflict within files, but may add other problems.**

* + - 1. Summarize **key insights** about **locking strategies**.

#### **Pessimistic locking seems straightforward but is inconvenient for teams.**

#### **Optimistic locking seems like a reasonable tradeoff.**

#### **Overly optimistic locking seems simple but risky, but can work well with good tools.**

# **Model 3: Version Control Systems**

## **Information**

A **version control system (VCS)** is a tool specifically designed to help manage versions, and attempts to combine the strengths of the various alternatives explored above, and to address additional issues explored below.

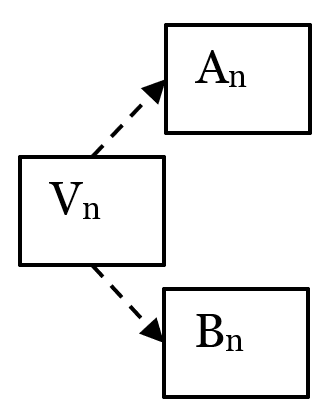
A **VCS** helps to manage multiple versions of files. A VCS is commonly used for software development, but can be used for other types of files. For example, a **wiki** is a VCS for website content, and a **log-structured file system** or Apple’s **Time Machine** is a VCS for an entire drive. In the questions below, **Vn** is the current version of all files in the VCS, and **Amy** and **Bob** are two developers.

Version control systems are software tools for:

* Maintaining project history
  + Review
  + Revert
* Facilitating collaboration
  + main branch / production version
  + Developers working asynchronously on feature branches
  + Merging new features into main branch

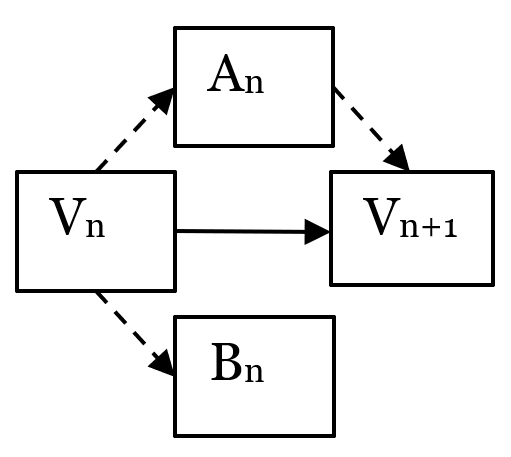
## **Instructions and Questions (15 min)**

To edit files, Amy and Bob first make their own local copies **An** & **Bn** of everything in **Vn**. We say that Amy and Bob **check out (or fork)** their own copies. Why is it better for them to edit their own copies,   
rather than the files in **Vn**?



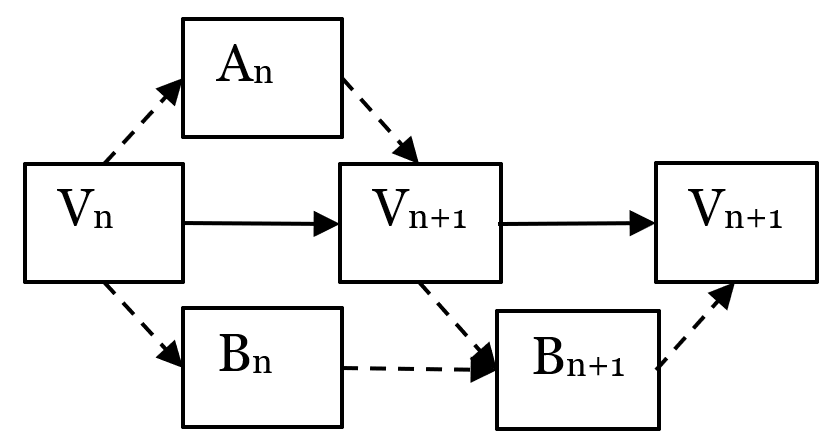
#### **Local copies avoid conflicts when 2 people try to edit the same file at the same time.**

After editing, Amy adds her changes to the VCS, to create new version **Vn+1**. We say that Amy **commits** changes to the VCS. Why should Amy test her code before she **commits**?



#### **To ensure that changes don’t cause problems for Bob or other developers.**

Before Bob can commit changes, he gets all of Amy’s changes from **Vn+1**. We say that Bob **updates** **(or pull)** his local copy from the VCS. Why should Bob **update** and testbefore he **commits**?



#### **To ensure that Amy’s changes (in Vn+1) and Bob’s change don’t combine to cause new problems.**

#### Explain which operation (**checkout**, **commit**, **update**), transfers: a. the **most** data b. the **least** data c. in between

#### **Checkout transfers the most (everything), commit the least (one set of changes), and update is in between (several sets of changes, possibly).**

* + - 1. To visualize versions and connections, it helps to draw **timeline diagrams** (like those above) with a **node** for each version, and a **link** (arrow) for each connection.

Draw a timeline diagram to show the following:

1. Start with version 1 of the project files.
2. Amy and Bob each checkout their own copies.
3. Amy edits & commits (creating a new version).
4. Bob updates.
5. Bob edits & commits (creating a new version).
6. Bob edits & commits again (creating a new version).

#### 

#### **sample answer:**

V10

A10

V11

B10

B11

V12

B12

V13

#### **NOTE: maybe have several teams draw their answers on board for others to review**

# **Model 4: Git**

## 

## **Information**

## Git and GitHub/GitLab are currently the dominant tools for supporting version control and collaboration in software development. These two tools work together.

* Individual developers use git to manage their repositories and track code changes.
* Communities use GitHub to bring together changes from developers and to track and communicate needs (e.g. issue tracker).

## Git:

## CLI version control tool that runs on your computer

## Maintains repositories (repos)

## Complete copy of the code / docs

## Complete history of changes

## Can also be integrated into IDE or GUI tool.

## gitHub/GitLab:

## A cloud service for hosting repositories

## Facilitates collaboration

## Forking/Upstreaming support

## Issue tracker

## Git and GitHub/GitLab may be dominant now, but lots of other options exist and are being used.

## Such as: Version control tools (CVS, Subversion (SVN), and Mercurial) and Repository Hosting (BitBucket, and SourceForge)

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

**Step 1: Local vs Remote**

* **Local:** things that are on a local machine, saved on a hard disk and can be used without access to a network.
* **Remote:** things that are in the cloud, stored on servers in data centers somewhere out on the internet. E.g. at GitHub or GitLab, etc…



1. What tools you can use to interact with things stored locally? Give some examples.

* **editor, browser, shell, git, etc...**

1. What tools you can use to interact with things stored remotely? Give some examples.

* **They can be interacted with through your browser.**

**Step 2: The Upstream**

When you go to join a FOSS community and use or contribute to its project:

* The project code will typically live on one of the repository hosting services (e.g. GitHub)
* The repository that lives on GitHub is the upstream repository
* It is like the reference copy of Aunt May’s cookie recipe.



The

**Upstream**

Repository

1. Why does it live on a hosting service?

* **Collaboration, transparency, release early & often**

1. Will you have “write” permissions to the upstream?

* **Nope, at least not until you gain trust and move up to a maintainer role.**
* **This is going to be an important factor is understanding why GitHub/git work the way that they do.**

**Step 3: Forking the Upstream**

To work with the project, you need to Fork the Upstream. I.e. When you Fork the Upstream, you make your own copy of the main project repo into your GitHub space.

The fork is exactly like the upstream (main project) repository and it knows where the upstream from.



The

**Upstream**

Repository

1. Why would you need to make a copy?

* **To make changes you need write permissions.**
* **You have full control over this copy.**
* **Add chocolate chips, add nuts, add coconut, you name it.**

1. How many forks will there be?

* **At least as many as there are developers**

1. Do you have full “write” permission to the contents of the fork?

* **Yes.**

**Step 4: Cloning your Origin**

This step is to Clone your fork. I.e. When you clone your fork, you make another copy of the repository on your local machine. That copy is called your Local repository. It is on your local machine, stored on your disk drive. The fork from which your local repo is cloned is called your origin repository (It is the” origin” of your local repo.)

Note that the local clone knows where the origin from which it was cloned is. I.e. it knows where it came from.



Your

Local

Repository

Your

Origin

Repository

The

**Upstream**

Repository

1. The name *origin* is used to refer to your fork on GitHub. This indicates that your clone knows the URL of your origin. How is the information about the *origin* remote represented in Figure above?
2. Why do you need to clone?

* **Can open files with your editor**
* **Can compile and run the code**
* **Can work without network access (crazy!)**
* **You also have full write access to this copy.**

1. Why should the local clone know where it came from?

* **It is how we will be able to upstream changes!**

**Step 5: Ready to Work**



Your

Origin (Fork)

Repository

The

**Upstream**

Repository

Your

Local (Clone)

Repository

1. Based on the previous steps and the diagram of this step, how many copies and where are they.

* **The state of things when you are ready to work on the project.**
  + **You have forked the main project repo to create your remote copy (i..e the origin)**
  + **You have cloned your origin to create your local repo.**

**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/Intro_to_FOSS_Project_Anatomy_(Activity)>
* K Crowston, J Howison (2005) [The social structure of FOSS development](https://firstmonday.org/article/view/1207/1127). *First Monday* 10(2).
* C Jensen, W Scacchi (2007) [Role migration and advancement processes in OSSD projects: A comparative case study](http://ieeexplore.ieee.org/document/4222598/). In *Proc. of the 29th Int’l Conf. on Software Engineering (ICSE’07)* (pp. 364–374). IEEE Computer Society.
* KR Lakhani, RG Wolf (2003) [Why hackers do what they do: Understanding motivation & effort in FOSS projects](http://papers.ssrn.com/sol3/Papers.cfm?abstract_id=443040). *MIT Sloan Working Paper* 4425-03.
* CG Wu, JH Gerlach, & CE Young (2007) [An empirical analysis of OSS motivations & intentions](https://www.sciencedirect.com/science/article/abs/pii/S0378720607000067). *Information & Management* 44(3):253-262.
* Y Ye, K Kishida (2003) [Toward an understanding of the motivation of open source software developers](https://dl.acm.org/citation.cfm?id=776867). *Int’l Conf. on Software Engineering*, 419-429.
* The Open Source Way 2.0[*https://www.theopensourceway.org/the\_open\_source\_way-guidebook2.0.html*](https://www.theopensourceway.org/the_open_source_way-guidebook2.0.html)

|  |  |
| --- | --- |
|  | *© 2010-2022 by Clif Kussmaul, Greg Hislop, Lori Postner, Heidi Ellis, Ali Al-Faris. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.* |