Project Management Tools & Concepts

Arctech Info Private Limited

Tools & Concepts covered in this session

- GIT
- Agile
- SCRUM
- Kanban
- JIRA (Bugs & Support Ticket management)
- Azure Devops
- GitHub

GIT

- Git is a free and open-source distributed version control system
- Designed to handle everything from small to very large projects with speed and efficiency.
- Git is easy to learn and has a tiny footprint with lightning-fast performance
- Features include
  - cheap local branching,
  - convenient staging areas, and
  - multiple workflows.

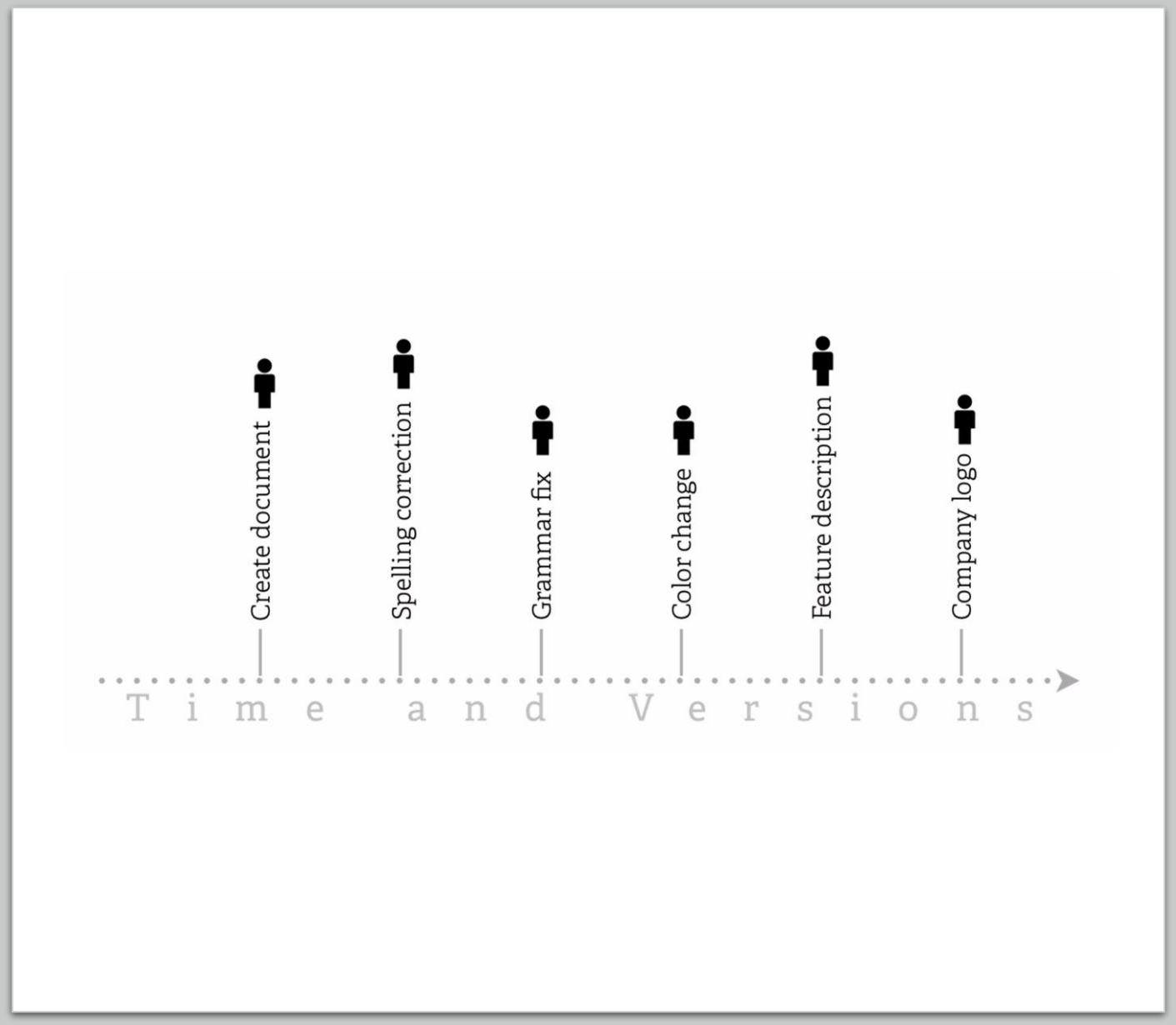
What is Version Control

- Designer, document creator or Software Developer (most common)
  - Creates things
  - Saves things
  - Edits things
  - Saves the thing again
- Version control helps when you “Save the thing again” and again and again
  - When you did it
  - Why you did it
  - What the contents of the change were
- You can review this information anytime in the future

History Tracking

These changes are simple to manage

- when a single person is doing them.
- Or if a single file is being changed, it's trivial to track it
- Some editing tools themselves maintain change history.



Collaborative History Tracking

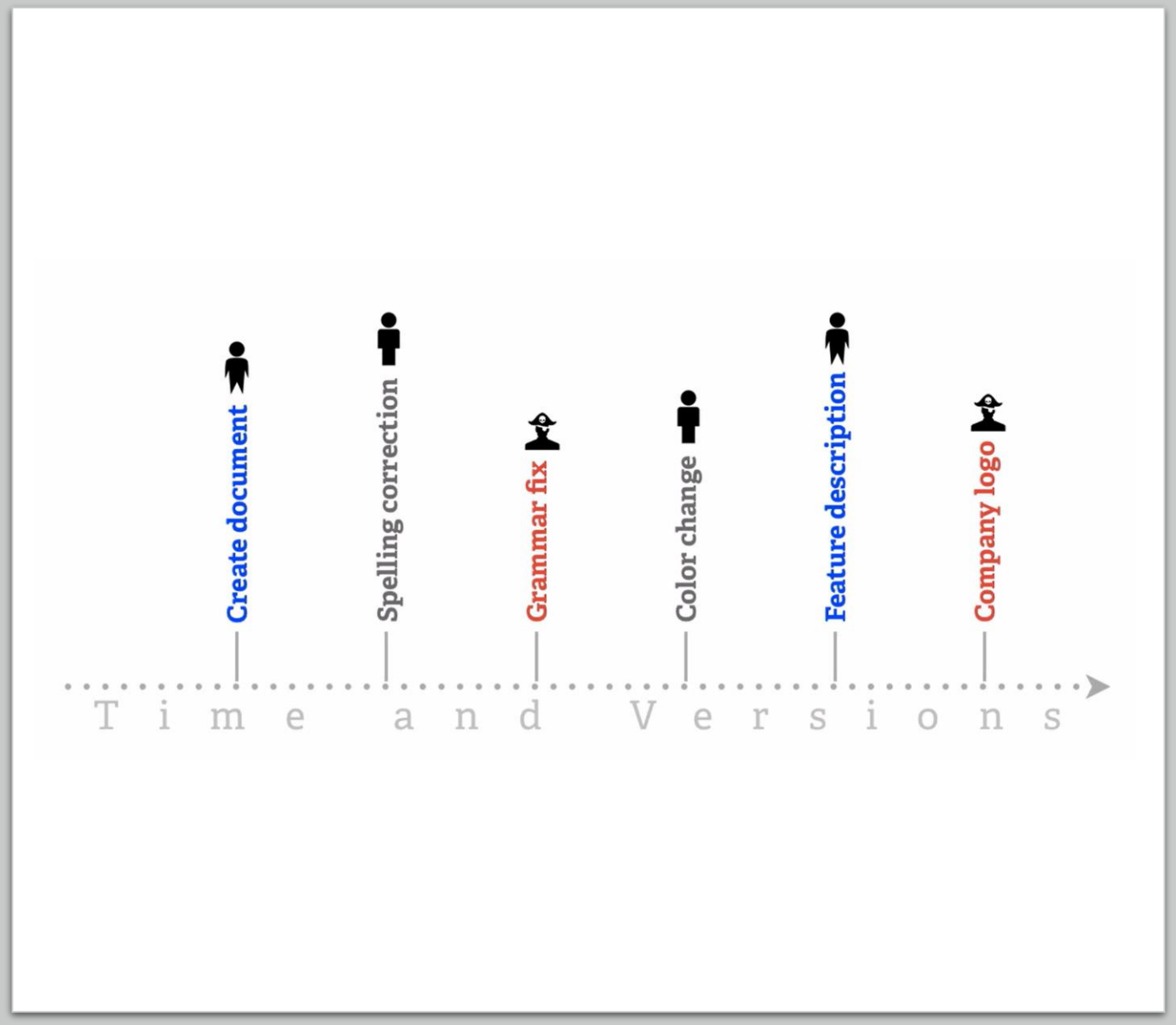
GIT really shines when we are talking about collaborative tracking.

When you and your team members trying to do similar thing to the same file.

You need to track

- Who changed it
- When they changed it
- Why they changed it

And more importantly unify these changes by merging all these changes into a single merge.



GIT – Distributed 1/4

- One of the nicest features of any Distributed SCM, Git included, is that it's distributed.
- This means that instead of doing a "checkout" of the current tip of the source code, you do a "clone" of the entire repository.
- **Multiple Backups**
  - This means that even if you're using a centralized workflow, every user essentially has a full backup of the main server.
  - Each of these copies could be pushed up to replace the main server in the event of a crash or corruption.
  - In effect, there is no single point of failure with Git unless there is only a single copy of the repository.

GIT - Branching and Merging

- The Git feature that really makes it stand apart from nearly every other SCM out there is its branching model.
- Git allows and encourages you to have multiple local branches that can be entirely independent of each other.
- The creation, merging, and deletion of those lines of development takes seconds.
- Notably, when you push to a remote repository, you do not have to push all of your branches.
- You can choose to share just one of your branches, a few of them, or all of them.
- This tends to free people to try new ideas without worrying about having to plan how and when they are going to merge it in or share it with others.

GIT - Small and Fast

Git is fast.

With Git, nearly all operations are performed locally.

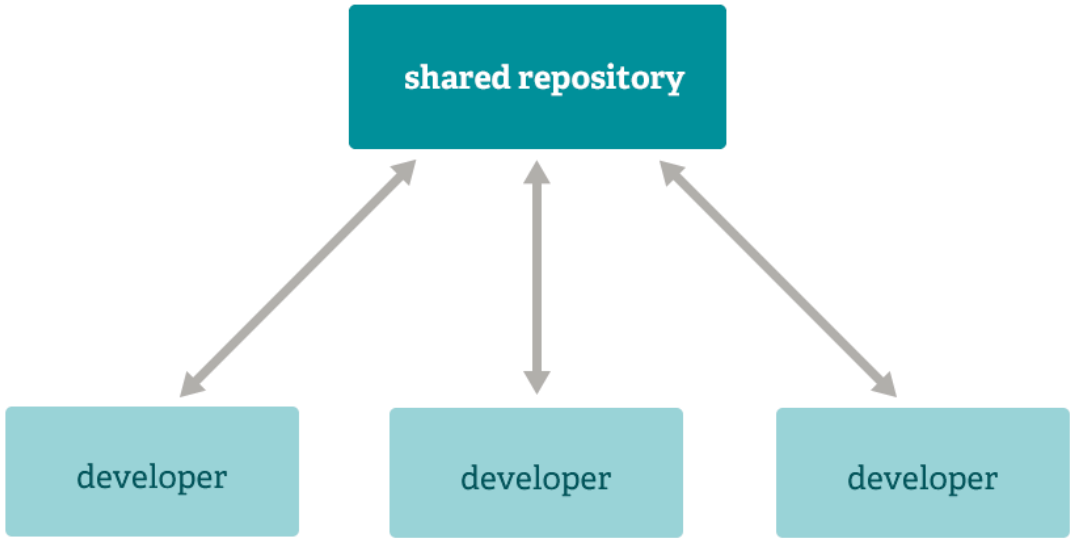
This gives it a huge speed advantage compared to centralized systems that constantly have to communicate with a server somewhere.

Git is written in C, reducing the overhead of runtimes associated with higher-level languages.

Speed and performance has been a primary design goal of Git from the start.

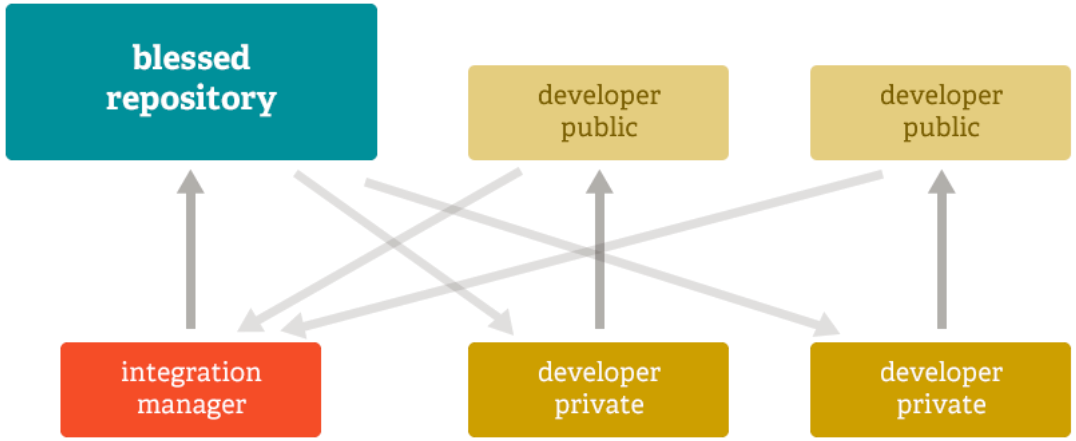
GIT – Distributed 2/4

- **Centralized workflow**
- Git will not allow you to push if someone has pushed since the last time you fetched.
- So, a centralized model where all developers push to the same server works just fine.



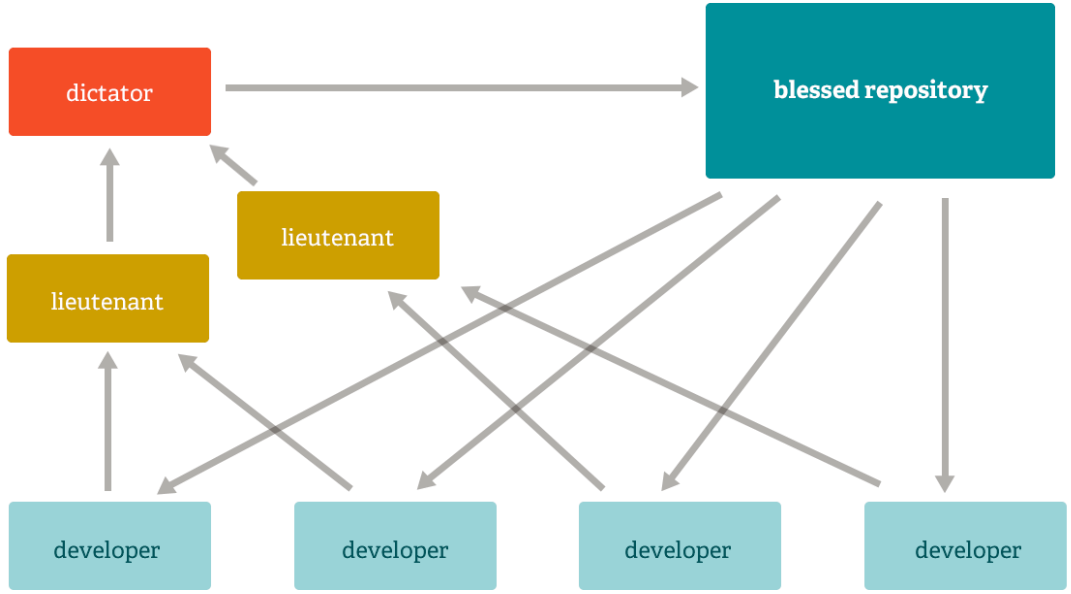
GIT – Distributed 3/4

- **Integration Manager Workflow**
- Another common Git workflow involves an integration manager — a single person who commits to the 'blessed' repository.
- A number of developers then
  - clone from that repository,
  - push to their own independent repositories, and
  - ask the integrator to pull in their changes.
- This is the type of development model often seen with open source or GitHub repositories.



GIT – Distributed 4/4

- **Dictator and Lieutenants Workflow**
- For more massive projects, this development workflow is often effective.
- In this model, some people ('lieutenants') are in charge of a specific subsystem of the project
- Lieutenants merge in all changes related to that subsystem.
- Another integrator (the 'dictator') can pull changes from only his/her lieutenants
- The dictator then pushes to the 'blessed' repository that everyone then clones from again.

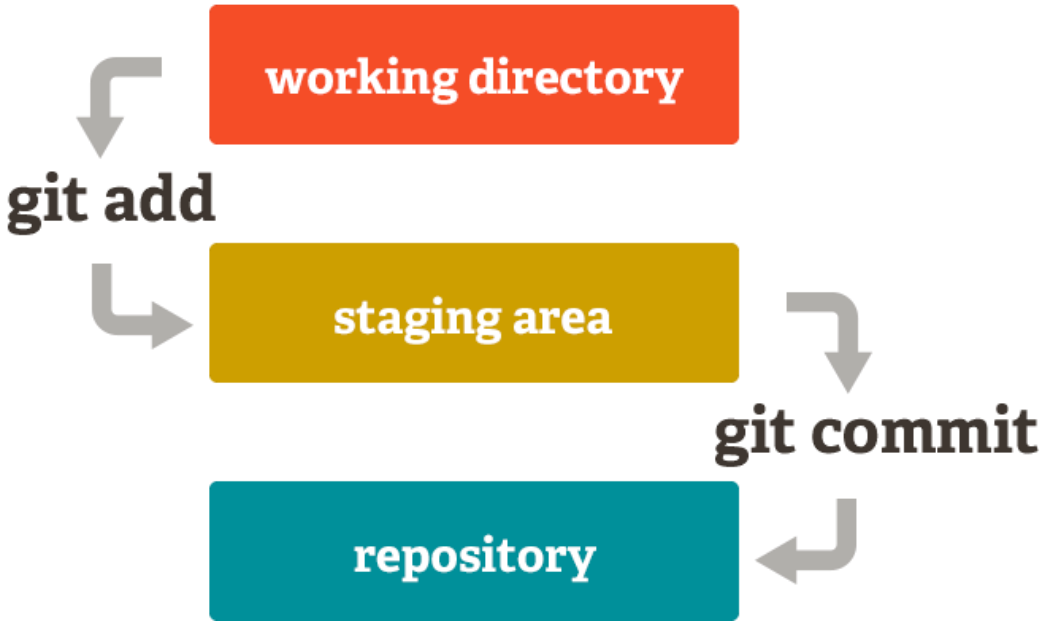


GIT – Data Assurance

- The data model that Git uses ensures the cryptographic integrity of every bit of your project.
- Every file and commit is check-summed and retrieved by its checksum when checked back out.
- It's impossible to get anything out of Git other than the exact bits you put in.
- It is also impossible to change any file, date, commit message, or any other data in a Git repository without changing the IDs of everything after it.
- This means that if you have a commit ID, you can be assured not only that your project is exactly the same as when it was committed, but that nothing in its history was changed.

GIT – Staging Area 1/2

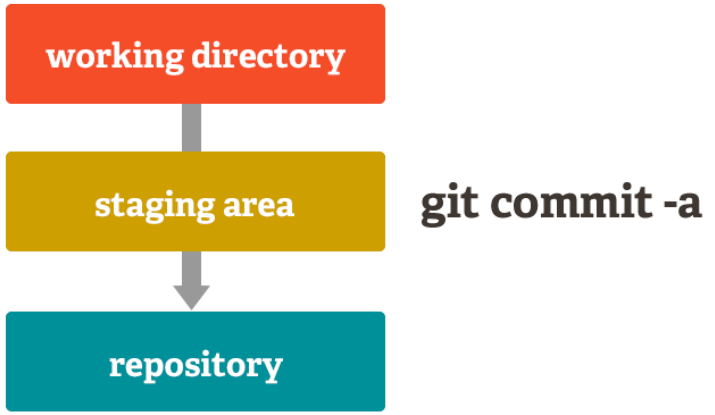
- Unlike the other systems, Git has something called the "staging area" or "index".
- This is an intermediate area where commits can be formatted and reviewed before completing the commit.



GIT – Staging Area 2/2

- This allows you to stage only portions of a modified file.
- Gone are the days of making two logically unrelated modifications to a file before you realized that you forgot to commit one of them.
- Now you can just stage the change you need for the current commit and stage the other change for the next commit.
- This feature scales up to as many different changes to your file as needed.

- Of course, Git also makes it easy to ignore this feature if you don't want that kind of control — just add a '-a' to your commit command in order to add all changes to all files to the staging area.



GIT enable our projects

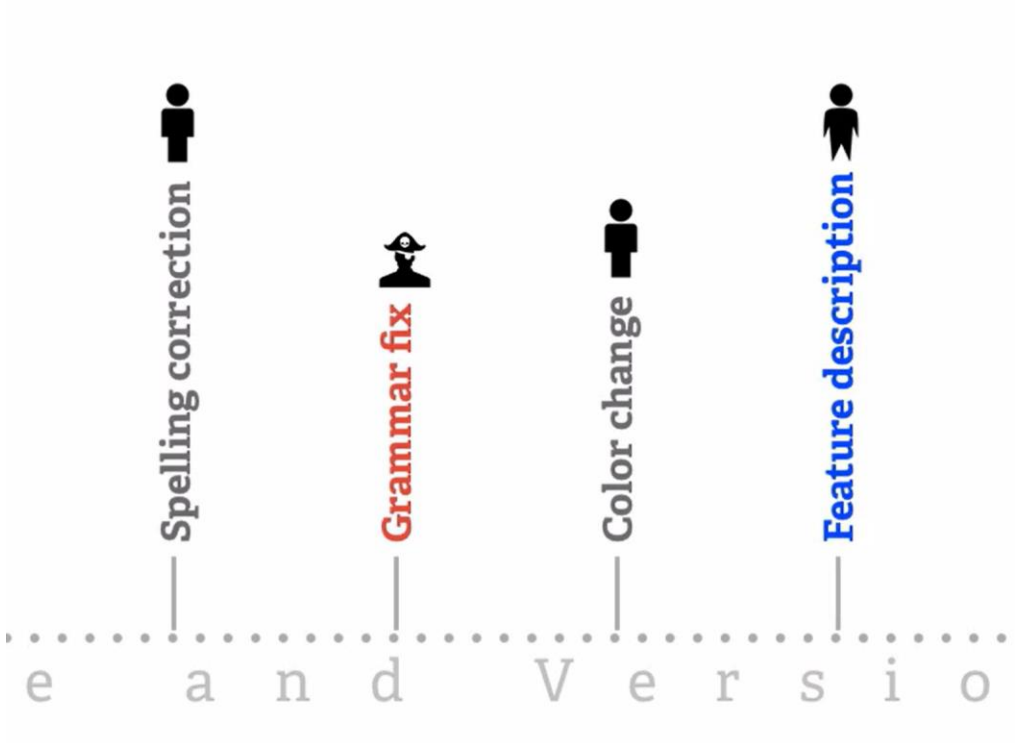
- There are many implementations of Version Control.
- GIT is the most popular and successful one
  - Many Version Control Systems require a complicated server setup
  - GIT is locally enabled. You can version control on your desktop just by using a command line program
  - You don’t need any complicated server software
  - However, if you want to use a server component to collaborate with your colleagues, that too is available.
  - The commands are easy to learn and can happen progressively

Single User Scenario

- Considering a single Software Developer sitting at a desk at any given point in time, version control is relatively simple
- On the command prompt the GIT commands would look like this
  - Initializing the project / setting up the git control structures
    - `git init myproject`
      - This creates a folder myproject which can hold the project files as well as the control files which store the history snapshots of documents, source code etc.
    - `cd myproject`
      - Change the current folder to the project folder
    - `git add .`
      - This command that notices the files and puts then in a holding zone
    - `git commit –m”Importing all code for this training”`
      - This command permanently records a snapshot of the files as they exist at a point in time.

Collaborative GIT

- GIT is team-centric so collaboration happens naturally
- Conceptually it is very complex, but with GIT it is as still as simple
- Consider two people working on one or more document
- Simple Collaboration



More realistic collaboration

- People working at almost different time but with overlaps
- Each taking a copy of the project in a point in time
- Make their own enhancements to them
- Then bringing back their work to the central copy of the project
  - Merge their changes to the central copy of the documents, source code, etc.
- This overlapping of time and almost parallel work complicates matters
- Lets see how GIT implements it

Example: Jim & Lina 1/3

- Consider two developers Jim & Lina working on the same project
- In GIT, every project has a master branch where the main copy of the entire project is stored
- Lets say Jim works on the **master** branch and adds a logo
  - Creates New logo file
  - Brings this copy to the central server

```
> git checkout master
> git commit –a –m”My new logo”
> git push
```

Example: Jim & Lina 2/3

- Now assume, Lina also participates at the same time but in a different branch - **linafeatures**
  - Adds some programming code
  - Brings this copy to the central server

```
> git checkout –b linafeatures
> git commit –a –m”My new code”
> git push origin linafeatures
```

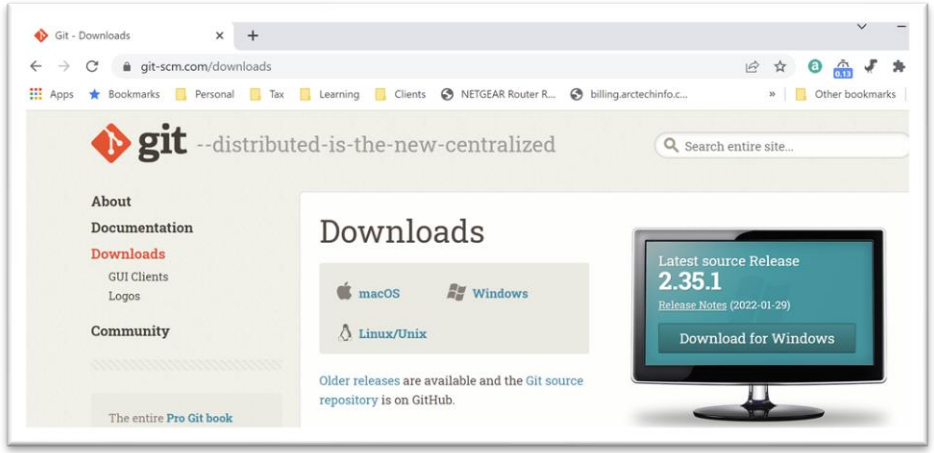
Example: Jim & Lina 3/3

- The last step, merging together Jim & Linas work
  - Retrieving latest copy of centralized files to Jims computer (both **master** & **linafeatures**)
  - Merge all changes from **linafeatures** branch into master branch
  - Push the master branch back to the central server
  - Optionally delete **linafeatures** branch
- During merge git will
  - Provide help if it finds conflicting files
  - Walk you through the steps when deciding whether to keep Jim's or Lina's changes if they collide
  - In case of separate files, the merge will be automatic

```
> git pull
> git merge linafeatures
```

Installing GIT

- Check if you have GIT installed, else install it from https://git-scm.com/downloads
- Configure GIT if not yet done. This is used by GIT to map each commit with the user




```
> git --version
> git version 2.33.0.windows.2

> git config --list

> Git config --global user.name "Jim Devops"
> Git config --global user.email "jim@comp.com"
```

Creating our 1st repository

- A repository in GIT is a folder containing other folders & files
- Create a GIT repo
  - This will simply create a local folder, to work with GIT
- Create a file in this folder and add it to version control
  - Add is not permanent
  - It is simply signaling to GIT that it wants to participate in version control
- Make a permanent record in the repo by committing the changes in the folder at a given point in time

```
> git init project1
> cd project1
** If the folder already exists only run > git init
> git add file1.txt
> git commit –m”My first commit”
```

GIT Staging

- Files in your Git repository folder can be in one of 2 states:
  - Tracked - files that Git knows about and are added to the repository
  - Untracked - files that are in your working directory, but not added to the repository
- When you first add files to an empty repository, they are all untracked.
- To get Git to track them, you need to stage them, or add them to the staging environment.

```
> git add file1.txt
> git add *.html
> git add “My Folder\*.*”
```

GIT Staging Environment

One of the core functions of Git is the concepts of the Staging Environment, and the Commit.

As you are working, you may be adding, editing and removing files.

But whenever you hit a milestone or finish a part of the work, you should add the files to a Staging Environment.

Staged files are files that are ready to be committed to the repository you are working on.

GIT commit

Since we have finished our work, we are ready move from stage to commit for our repo.

Adding commits keep track of our progress and changes as we work.

- Git considers each commit change point or "save point".
- It is a point in the project you can go back to if you find a bug or want to make a change.

When we commit, we should always include a message.

- By adding clear messages to each commit, it is easy for yourself (and others) to see what has changed and when.

Add another file to repo

```
** Create 2 new files index.html and help.txt in the folder

> git add --all
> git commit -m”Added 2 new files to begin programming”

** Add a new file style.css
** Change index.html to include the css

> git status
> git add --all
> git status

> git commit -m”Some more changes done”
> git log

> git commit --help
> git help --all
** This is a long list. Pg-Up & Pg-Dn. q to exit
```

GIT Branch

In Git, a branch is a new/separate version of the main repository.

Let's say you have a large project, and you need to update the design on it.

How would that work without Git:

GIT Branch

With a new branch called new-design, edit the code directly without impacting the main branch

EMERGENCY! There is an unrelated error somewhere else in the project that needs to be fixed ASAP!

Create a new branch from the main project called small-error-fix

Fix the unrelated error and merge the small-error-fix branch with the main branch

You go back to the new-design branch, and finish the work there

Merge the new-design branch with main (getting alerted to the small error fix that you were missing)

GIT Branches

Branches allow you to work on different parts of a project without impacting the main branch.

When the work is complete, a branch can be merged with the main project.

You can even switch between branches and work on different projects without them interfering with each other.

Branching in Git is very lightweight and fast!

GIT Branches demo

```
** Create a branch
> git branch hello-world-images

** View all branches
> git branch

** Switch to the new branch
> git checkout hello-world-images

** Make some changes/ add new files etc
> git add --all
> git status
> git commit -m "Added image to Hello World"

** switch back to master branch
> git checkout master
```

GIT Branches demo – emergency fix &merge

```
** emergency fix in master
> git checkout -b client-emergemcy-fix

** make changes in files
> git status
> git add index.html logo.img
> git commit
```

GIT summary

What is GIT used for

- Tracking code changes
- Tracking who made changes
- Coding collaboration

What does it do

- Manage projects with Repositories
- Clone a project to work on a local copy
- Control and track changes with Staging and Committing
- Branch and Merge to allow for work on different parts and versions of a project
- Pull the latest version of the project to a local copy
- Push local updates to the main project

Why GIT

Over 70% of developers use Git!

Developers can work together from anywhere in the world.

Developers can see the full history of the project.

Developers can revert to earlier versions of a project.