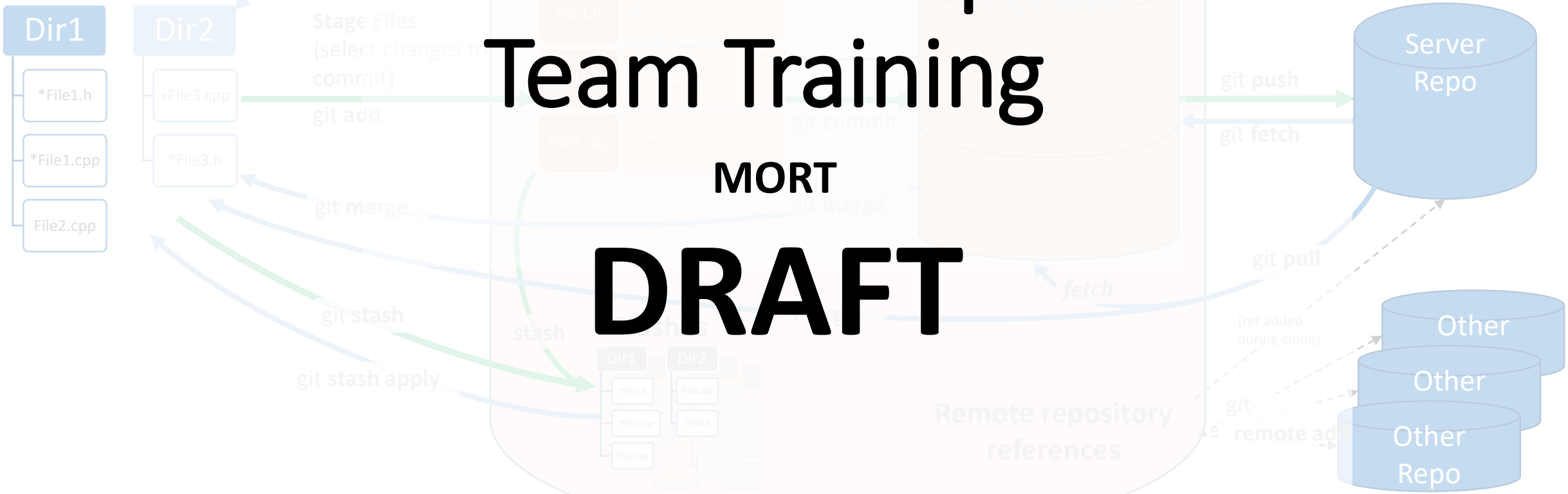
Git for Developers
Team Training

MORT

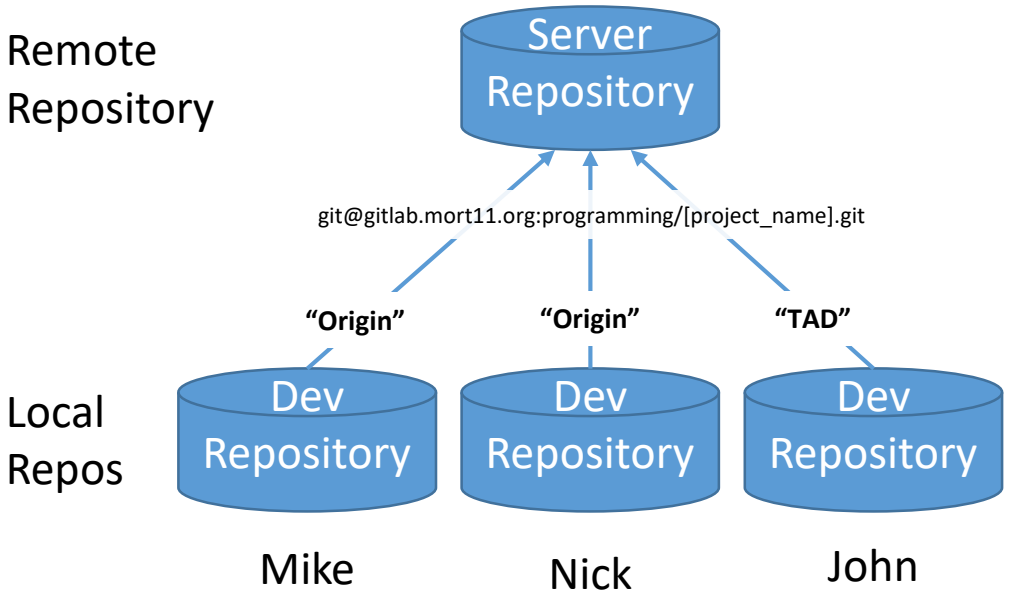
DRAFT



Dan Steinmark

Oct 2019

Git Uses Distributed Repositories



When Repositories are properly synced, they are full duplicates of one another

- New repository created from scratch
- For developers, this will be the ***remote repository***

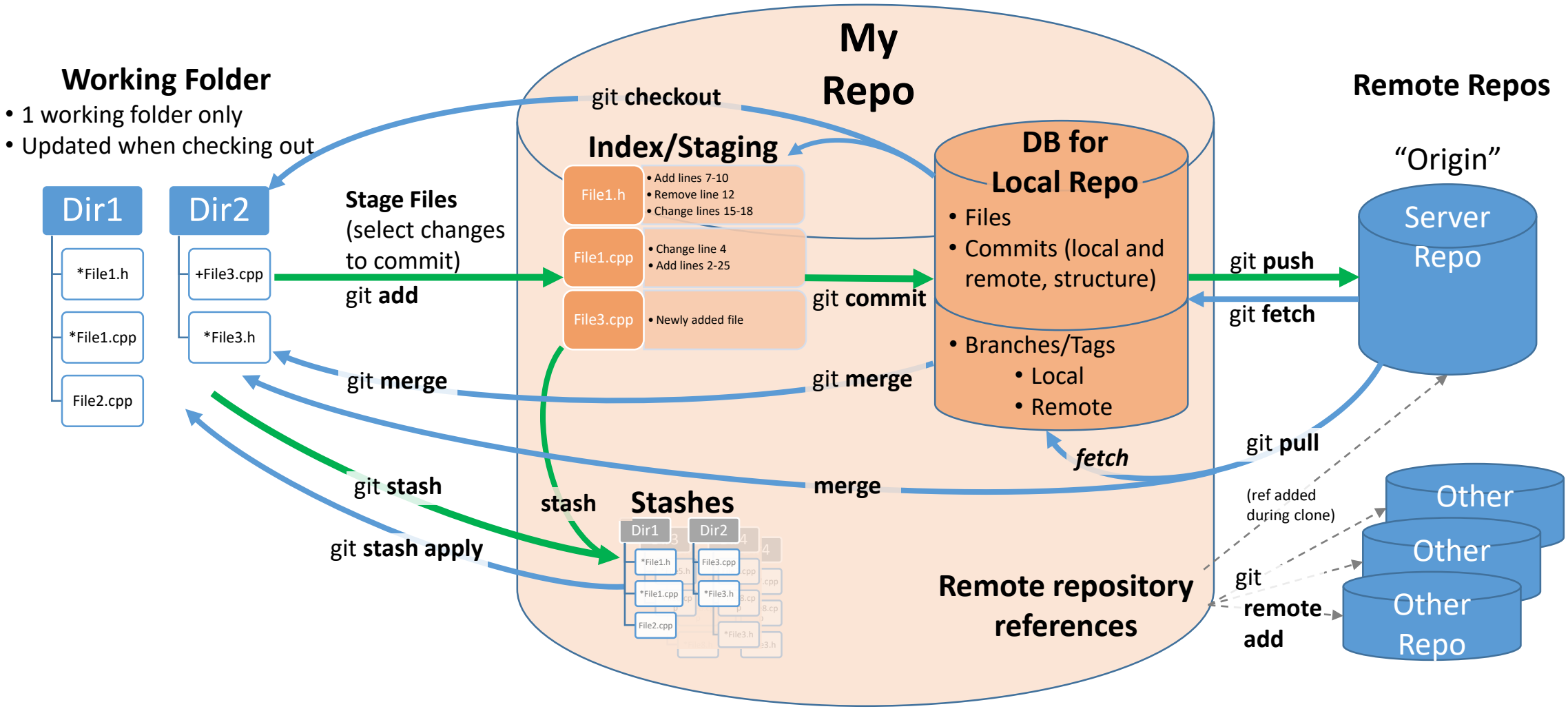
- **Cloned** repositories created from the Server Repository
- Work is done here, in the ***local repository***
- Refer to the originating repository as "Origin" (by default)
- Changes the Dev makes are ***committed*** to the *local repository*
- Those changes are ***pushed*** to the *remote repository*
- Other Dev's changes are ***fetched*** from the *remote repository* to update the *local repository*

- **NOTE:** The Dev repositories are usually on your local machine *and NOT BACKED UP.* **Commit and push your changes frequently** (1-2x per week) to ensure you do not lose work if your hard drive fails.

Git is a Database

- Git saves your work
- Git allows you to share your work
- Git allows you to merge your work with others
- Git allows you to roll back your changes
- You can save your work without sharing your work (commit only)
- You can save your work and share it (commit and push to remote)

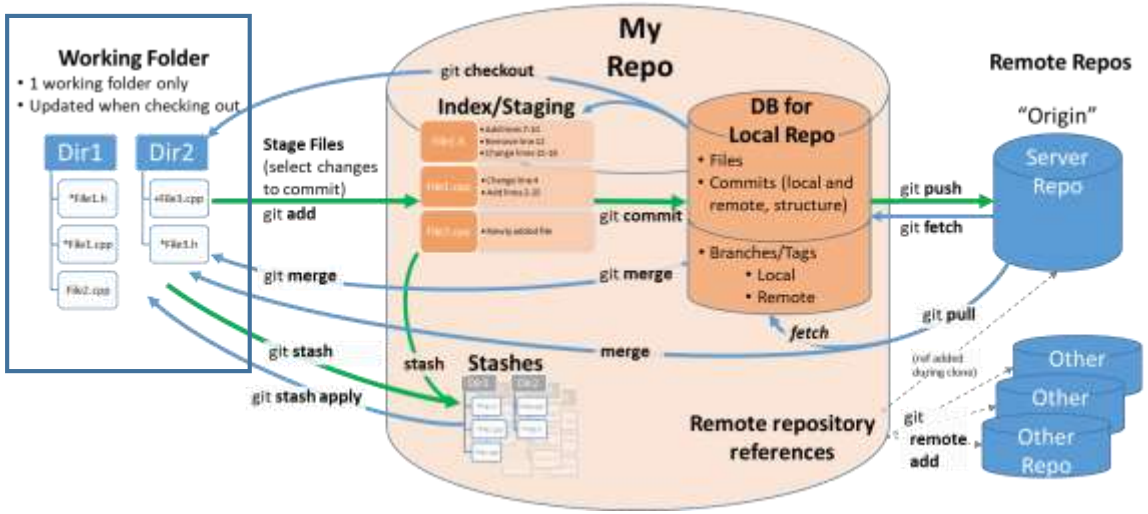
Git Repositories and Interface Elements



Git – The Working Folder

A place in the file system where you make changes

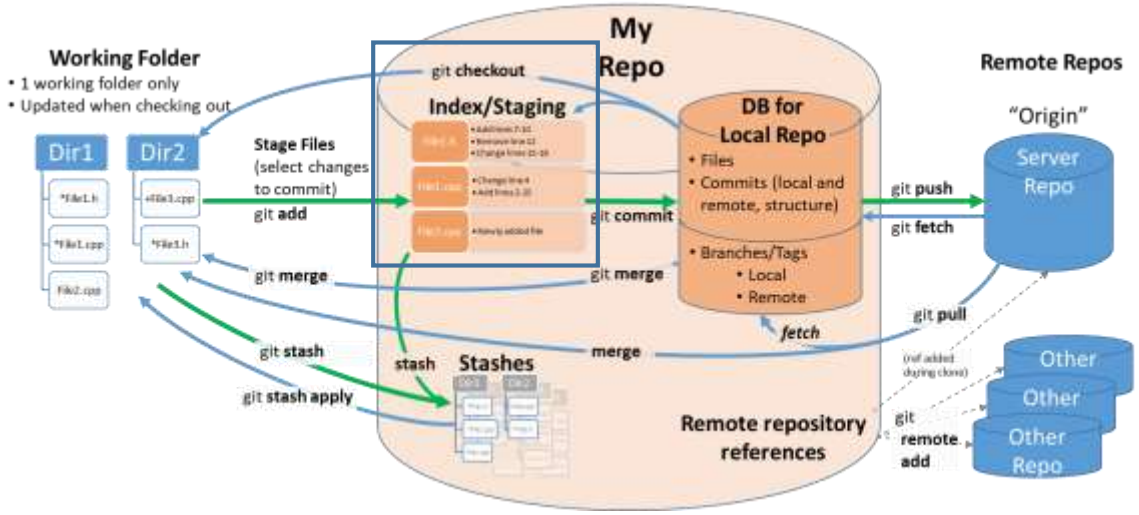
- Add, delete, or edit files
- Git monitors the working folder for changes, unless you tell it what files not to track with a ***gitignore*** file
- Updated when you ***checkout***, ***pull***, or ***merge***
  - Files are essentially unpacked from the Repo into the working dir whenever you checkout
- Only 1 Working Folder for the repo
- Where your `.git` repository folder resides (usually)
- Can be stashed – moved to a temporary location (the stash) if you need to be interrupted



Git Repositories – Index/Staging Area

The Index (or Staging area) keeps track of what you have selected to commit.
Also manages what files are being tracked or ignored.

- Add items to the Staging area by performing a ***git add***
- You can add files to the Index as you make changes, or add them all at once just prior to a ***commit***
- Git knows what is in your working folder and what has changed.
  - To add everything changed (not otherwise ignored) in your Working Folder use
    - `git add -A` or
    - `git add .`
  - Or, select everything in a tool such as *Git Extensions* or *TortoiseGit*



- You do not have to commit everything – you can select what to ***add*** by selecting or specifying named files. You can use * wildcards to be interpreted by the shell.

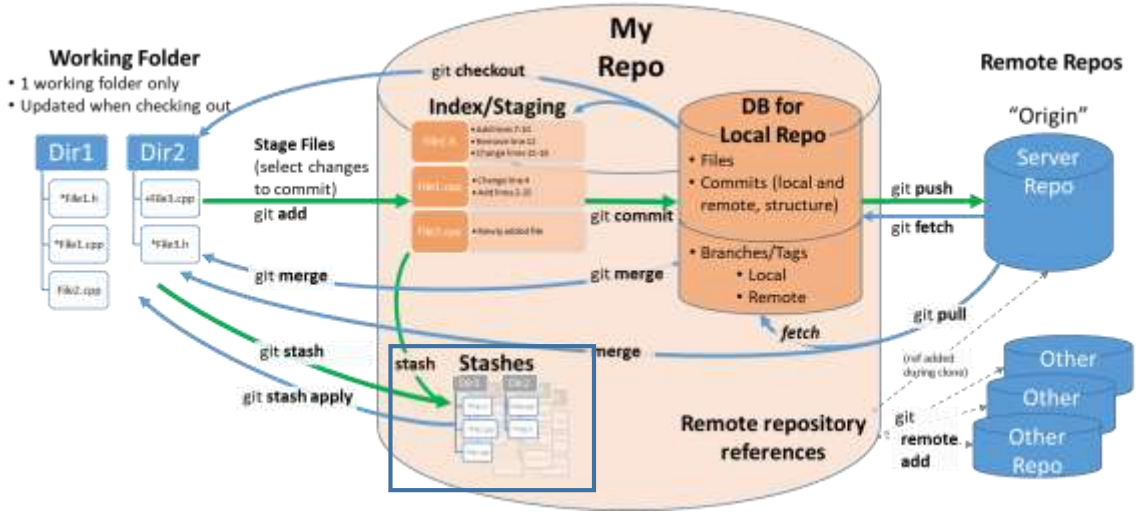
- When you `checkout`, the Index is updated to track the files appropriate to the branch.

Git Repositories - Stashes

Used when you want to save the current state of the working directory and the index, but need to go back to a clean working directory

- Saves your local modifications away **and reverts the working directory** to match the current branch *in the repo* – it's essentially like wiping out your directory and checking out again from the current branch
- It's a *move* not a *copy*
- Can behave like a stack
- Has individually accessible stashed elements
- Elements are usually listed as WIP on *branchname*
  - WIP = Work In Progress
- Can provide a message (description) when stashing
- Can't stash with unresolved merge conflicts

- Normal stash
  - `git stash` or
  - `git stash push`

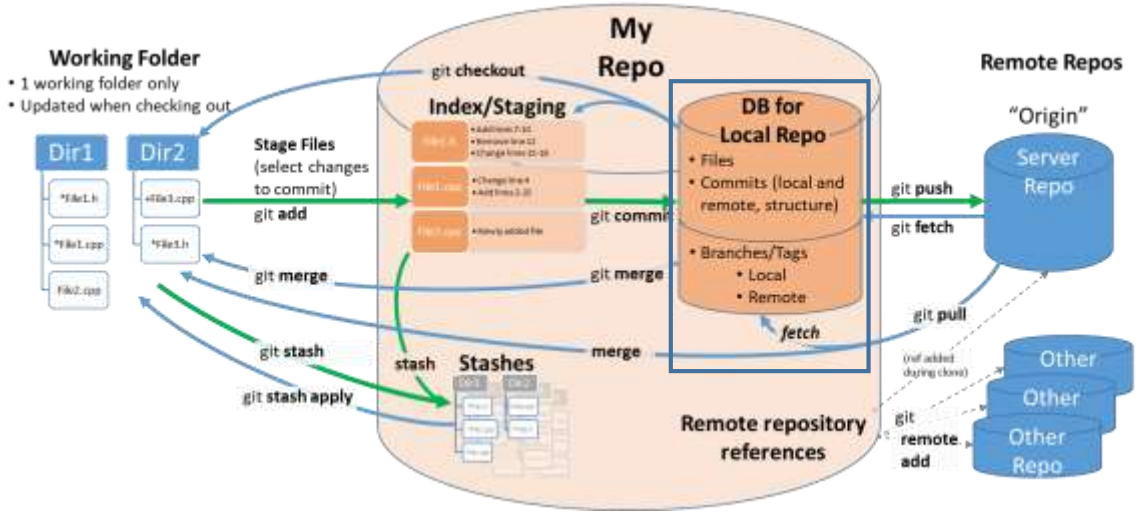


- Stash everything including untracked files with:
  - `git stash --include-untracked` or
  - `git stash --all`
- Reload from the stash
  - `git stash apply` (apply stash to Working folder but leave at front of Stashes)
  - `git stash pop` (apply stash to Working folder and remove from Stashes)

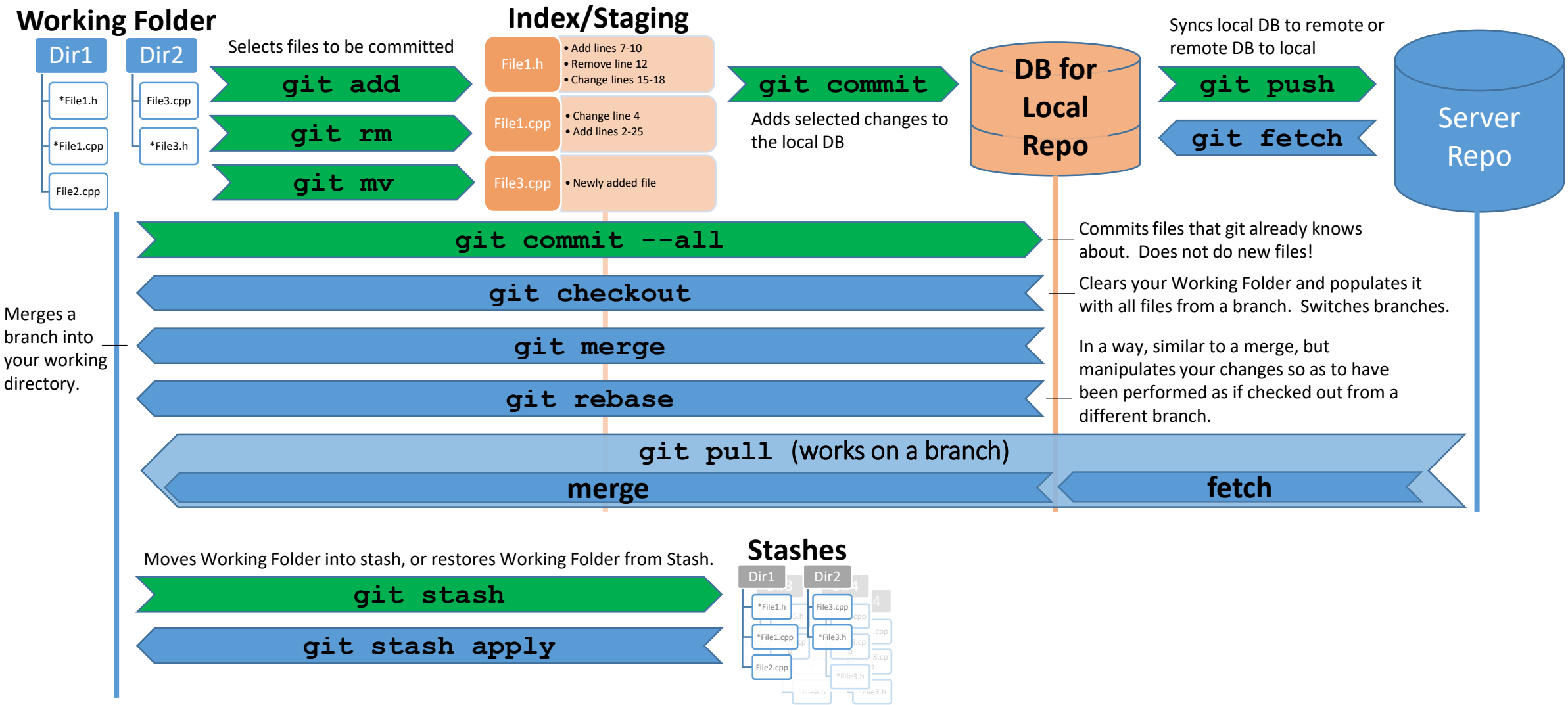
Git Repositories – DB for Local Repo

Stores files and change history, as well as commits for both local and remote repositories

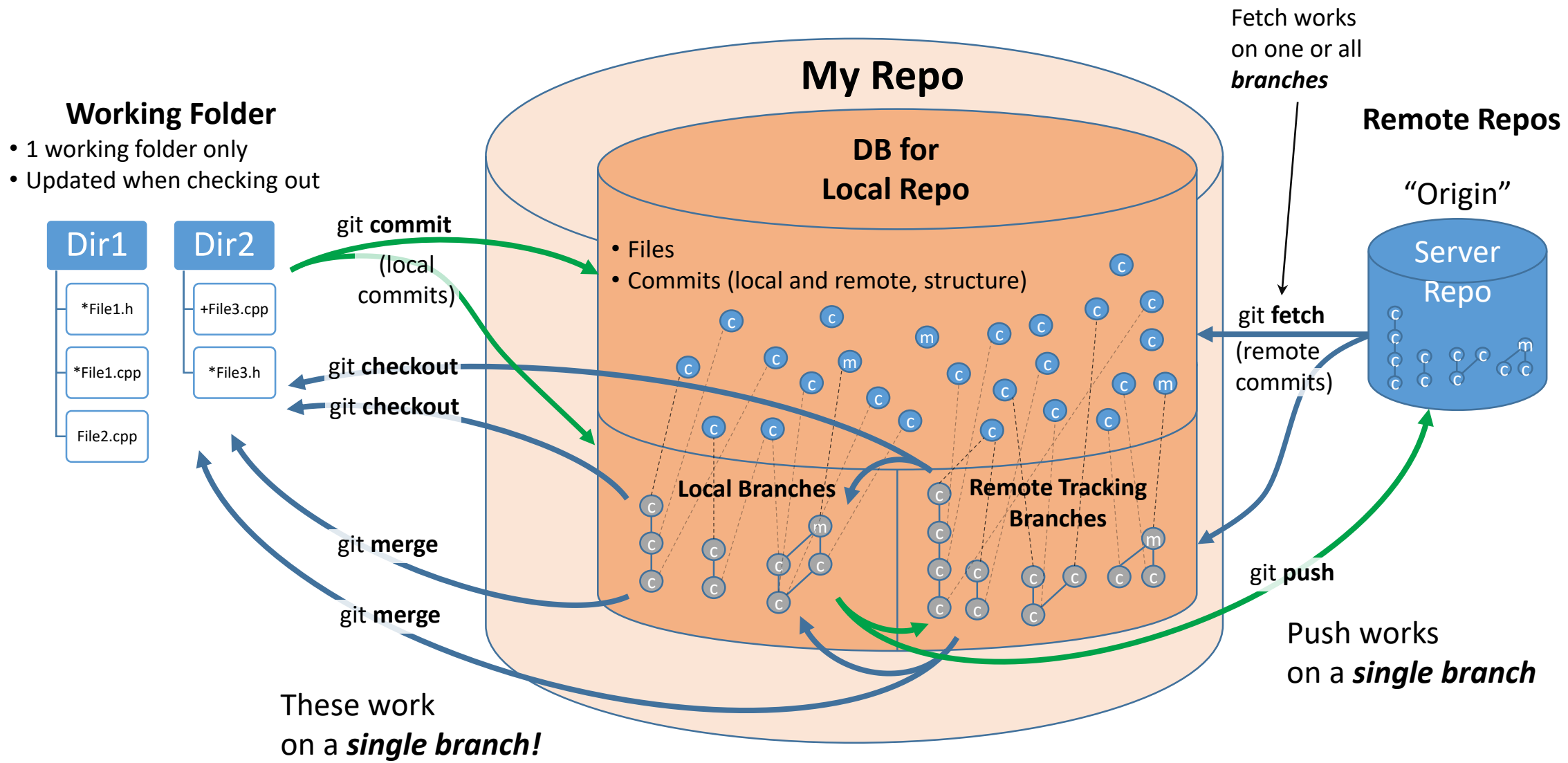
- Stored in the `.git` folder which resides in your working folder
- Maintains information about Branches and Tags (though not directly in the object database)
- Updated when you ***add*** then ***commit*** your changes
- Updated when you ***fetch*** or ***pull*** from a remote
- Use ***push*** and ***fetch*** to keep your local repo and the remote repo in sync
- Manages local branches distinctly from remote branches



Git data transport commands



Commits and Branches



Branches

- **Master branch** – this is the main branch for all of development
  - Protected from pushes on the Server Repo so it can be controlled
  - As such, you must issue a pull request to get anything in there
- **Terminology:**
  - **Local Branch** – One of your branches *in* your *local repository*
  - **Remote Branch** – A branch *in* one of your *remote repositories*
  - **Remote Tracking Branch** – A branch *in your local repository*, that's tied to your remote repository, and updated when you `fetch`
  - **Tracking Branch** – A tracking branch is an attribute that can be created *for your Local Branch*, to tie it to a *specific Remote Branch*, so that push and pull default to there.
- **Local Branch**
  - You can work on a local branch (only)
  - One of the local branches (the active/checked out one) represents your working folder

Branches

- **Remote Tracking Branch**
  - Although this is *in* your local repository, you cannot work on it directly
  - It is a snapshot of the branch of the same name in the remote repo
  - You can `checkout` from it, but it creates a local branch that you DO work on
    - Automatically creates a tracking branch to tie your local branch to the remote branch – even if you give a new name to your branch
- **Tracking branches – convenience for *pushing* and *pulling***, to let git know what destination branch to default to
  - In Visual Studio, `pull` **only** works with the default, which is usually just your branch. **Do not pull in VS** and expect it is getting anything from origin/master.
  - In Git Extensions, `pull` allows you to pull from anywhere, but if you leave the field blank, it will pull from the tracking branch
  - When checking out from a local branch with a tracking branch, it will indicate whether you are ahead or behind the remote tracking branch

Using Branch Commands

Note: all commands are done from your local repository

- **Creating branches**
  - Branches are lightweight and you should not hesitate to create a branch
  - You should *create a branch* directly, and **not** by *checking out* and giving a new name
  - You can use the *branch* command in Git-Extensions to also *checkout* **in a single step**
- **Updating branches** (pushing, fetching, pulling)
  - **Local Branches are not updated via fetch, only Remote Tracking branches are**
  - Local Branches only updated when you merge (or pull, which does a fetch and a merge). In fact, only the *active* (checked-out) Local branch can be updated.
  - You can fetch a single branch or fetch all branches
  - You can fetch or pull from the remote branch of the same name – you will *only* get what others have put into *that* branch
  - You can fetch or pull from origin/master, to get the latest from the server's master branch
  - You can only push a single branch at a time
  - Push your branch to get your commits up to the Server Repo

Using Branch Commands - Merging

Note: all commands are done from your local repository

- **Merging branches** (standard merging)
  - Merging a branch is how you import other's updates into your code
  - It works from a *specified* or *default branch* into your active (checked out) branch in your working folder
  - You can merge from the branch of the same name, in which case, you only get what others have put into your branch. This can be useful for collaboration branches.
  - You can merge from any other branch
    - Usually this is to pull from origin/master
    - Can be to get needed code from someone else's branch that has not yet made it into master
    - Merging only affects your local branch
  - When you merge and commit, it creates a *merge commit* which uses *your last commit* and *the other branch's last commit* as parents. All commit history is retained.

Using Branch Commands – Squash Merge

Note: all commands are done from your local repository

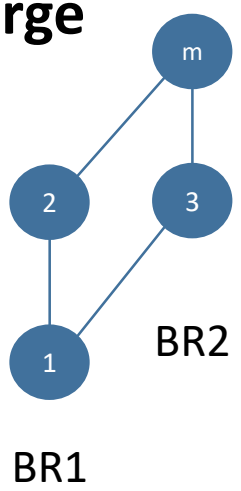
- **Squash merging branches**
  - A *squash merge* is performed when you wish to condense commit history
  - A *squash merge* does not retain all commit history, it reduces the branch to only a single commit
  - This is useful on the Server Repo in combination with deleting branches, where commits from everyone could clutter the views
  - A single commit is placed on top of the branch, with only a single parent (the prior commit to that branch)
  - The single commit is a summation of all of the commits in the branch
  - This is how your commits are merged into origin/master

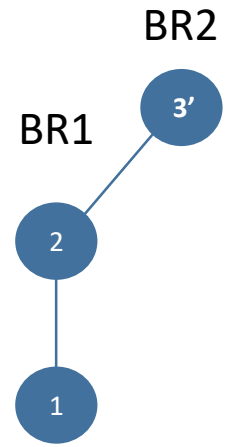
Using Branch Commands - Rebasing

Note: all commands are done from your local repository

- **Rebasing branches**
  - A different type of merge – it manipulates your changes to perform as if they were done on top of a different branch (or a different point in the branch)
  - Reorganizes commit history to change a branch's divergent point
  - Essentially uses the code from the new branch, and applies your changes to *that* codebase, instead of the one you used originally
  - Applies your changes on top of the branch commit specified
  - **Rebasing is not recommended** – a regular merge is preferred because rebase conflicts can be more difficult to resolve, and cannot be performed after having pushed your branch to a remote repo
  - **NEVER rebase after having pushed! (You'll break other's who have pulled it.)**

Standard Merge





Rebase

Collaboration Branches

- Use when working together on the same functionality
- Not a git term. This is just a term for our own use.
- Create a branch for collaboration
  - **Very important** to know you are collaborating, and who you are cooperating with. Communicate with each other.
  - If working closely, `checkout` from the same collaboration branch
  - If pretty distinct from each other, can make separate branches and `checkout` from the separate branches. To share, you need to explicitly push your changes back to the collaboration branch.
- Push to the collaboration branch to share information regularly
- Pull, or fetch and merge regularly from the Server Repo collaboration branch to get the latest from your partners
- At least one of you needs to merge in updates from origin/master, and push those to the collaboration branch

Branch Commands
Limitations and Restrictions in IntelliJ

- TBD

Branch Commands
Limitations and Restrictions in Visual Studio

- Visual Studio does a poor job identifying where you pull from
- DO NOT `pull` in Visual Studio, only from Git Extensions
  - Visual Studio will not likely do what you want – there is no easy way to pull from origin/master (based on our branch directives)
- DO NOT `checkout` in Visual Studio, only from Git Extensions
- DO NOT create branches in Visual Studio, only from Git Extensions
- `push` and `fetch` are ok in Visual Studio, but better in Git Extensions

>>> USE Git Extensions instead of Visual Studio <<<

Tags

- Tags are ref's that point to specific points in Git history
- Tagging is generally used to capture a point in history that is used for a marked version release (i.e. `midnight_mayhem_staging`).
- A tag is like a branch that doesn't change. Unlike branches, tags, after being created, have no further history of commits.
- Creating tags
  - Lightweight – like a branch that doesn't change, just a pointer to a specific commit, can be used as an ID to label it
  - Annotated – full checksummed object, tagger name, email, date, tagging message, can be signed. Use for important tags.
- Check out from a tag – `git checkout Detroit_staging`, gets everything for viewing, exporting, or experimenting (committing potential changes requires and extra checkout step)
- Need to explicitly push tags
- Filter by tags

Pull Requests

- Mechanism to get your updates into the *Master branch* on the Server Repo
- Done using ~~Team Foundation Server (TFS)~~ TBD

Pull Requests in TBD

Show a nice set of pics

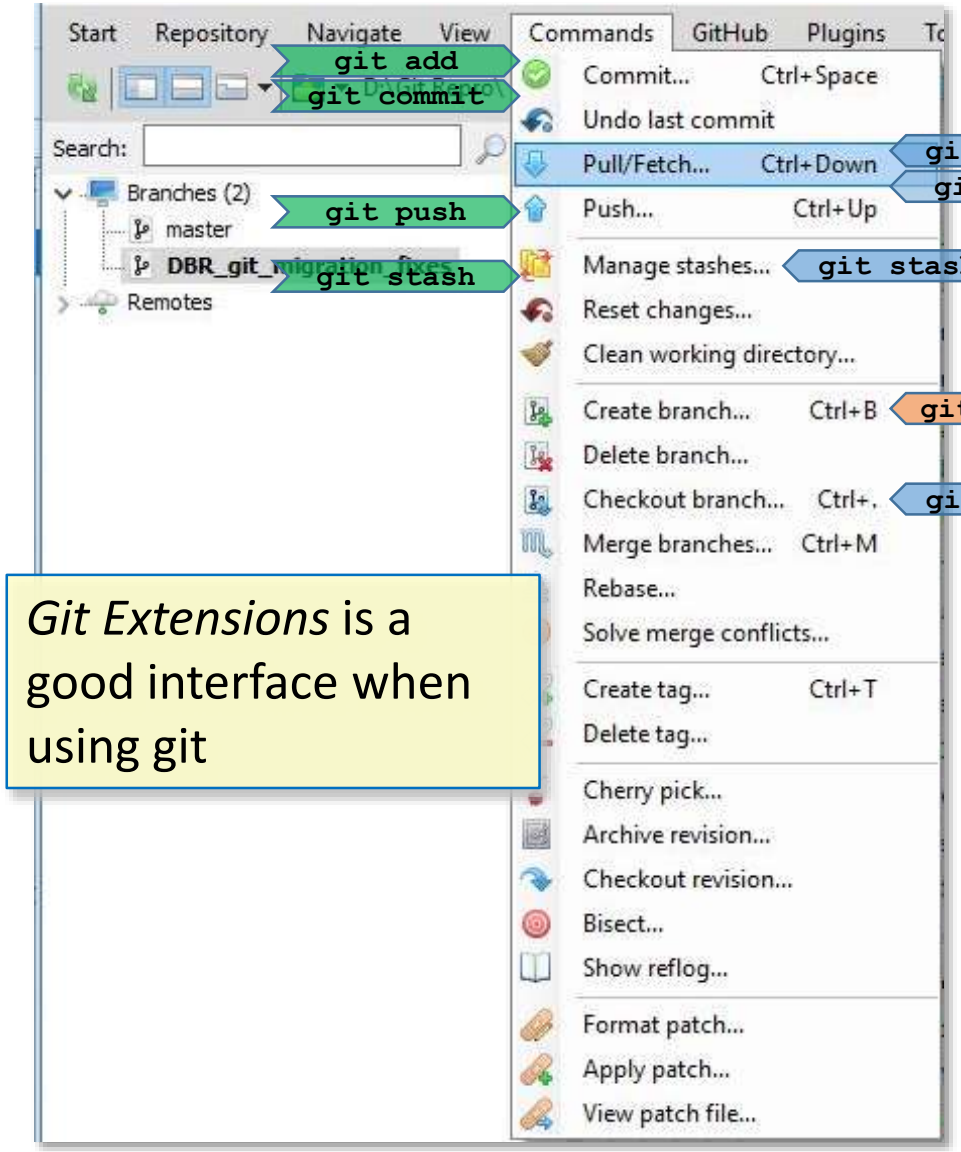
Other Commonly Used Commands

- Git init – create a new, empty repository
- Git clone – clone a repository from another source
  - Copies and migrates all data into new repo (leaving old repo intact)
  - Sets up source repo as *Origin* in new repo
- Git remote add – adds a remote reference to an additional repo
- **Undo in Git**
  - `git reset` – undo by *deleting* the last commit(s)
  - `git revert` – undo by adding a commit to undo prior commit(s)
  - DO NOT change any commit (via `commit --amend`) except for the head commit
  - DO NOT make any CM corrections *after having pushed* a mistake to the Server Repo (get Mike or Tech Lead involved)
- Git cherry-pick – Given one or more existing commits, apply the change each one introduces, recording a new commit for each

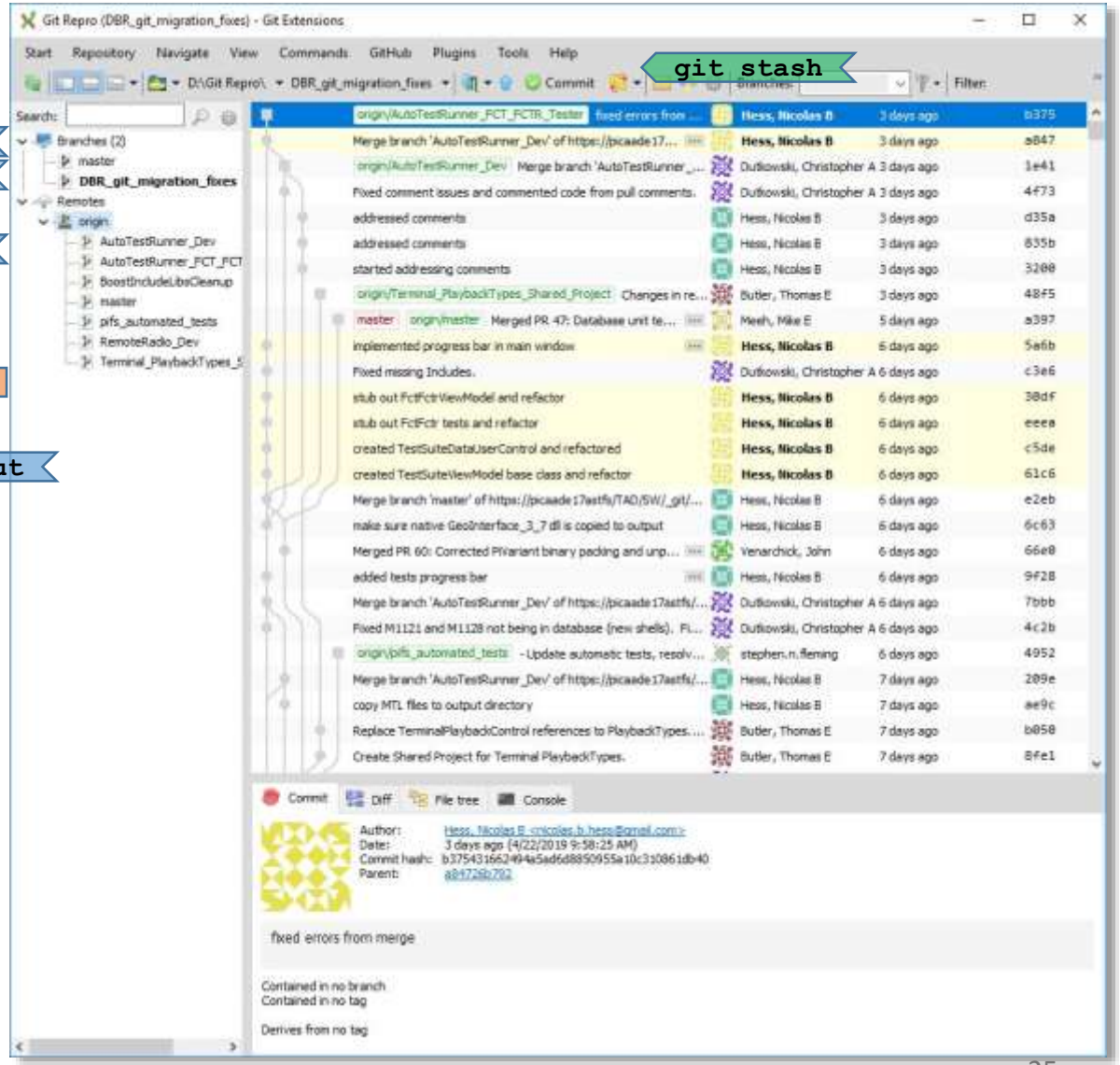
Other Commonly Used Commands

- Git status commands
  - `git log` – shows the commit logs. Can be filtered by:
    - a revision range
    - a path name (can be strings or regexps based on flags)
    - Only merges, or not merges, or with a specific number of parents
    - others
  - `git status` – shows files (paths) that have differences between the working folder and the index, the index and what has been committed, and untracked files. Can show just filename or also textual changes for some.
  - `git diff` – shows changes between pairs of commits, between a commit and working folder, pairs of paths, etc.
    - Flags can be provided to change diff behavior, such as to use *minimal* diff instead of *greedy* diff

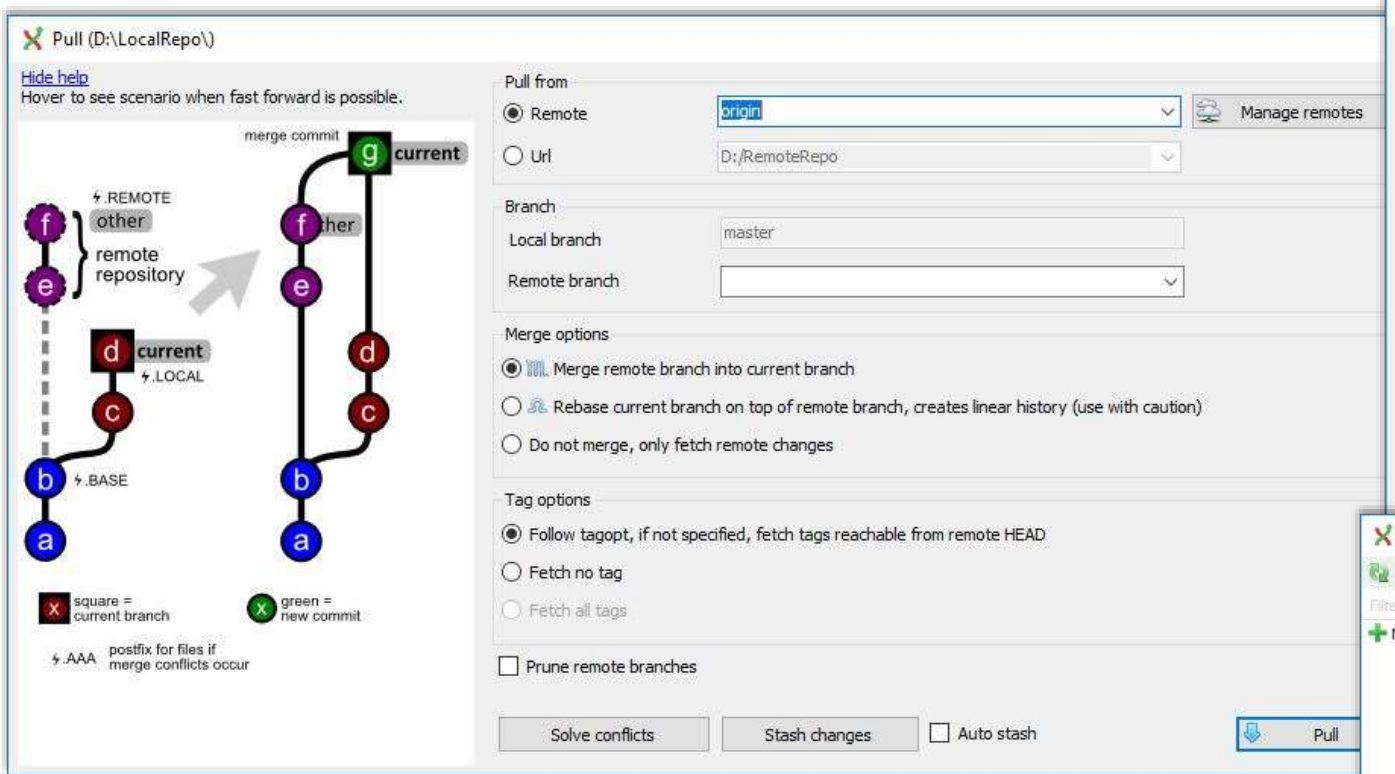
Intro to *Git Extensions*, a GUI Interface for Git

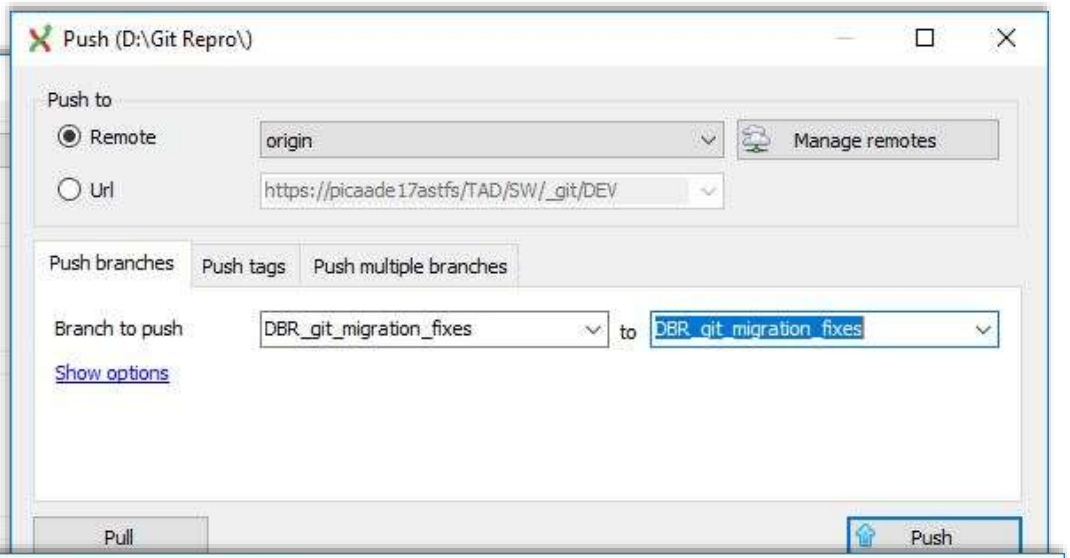


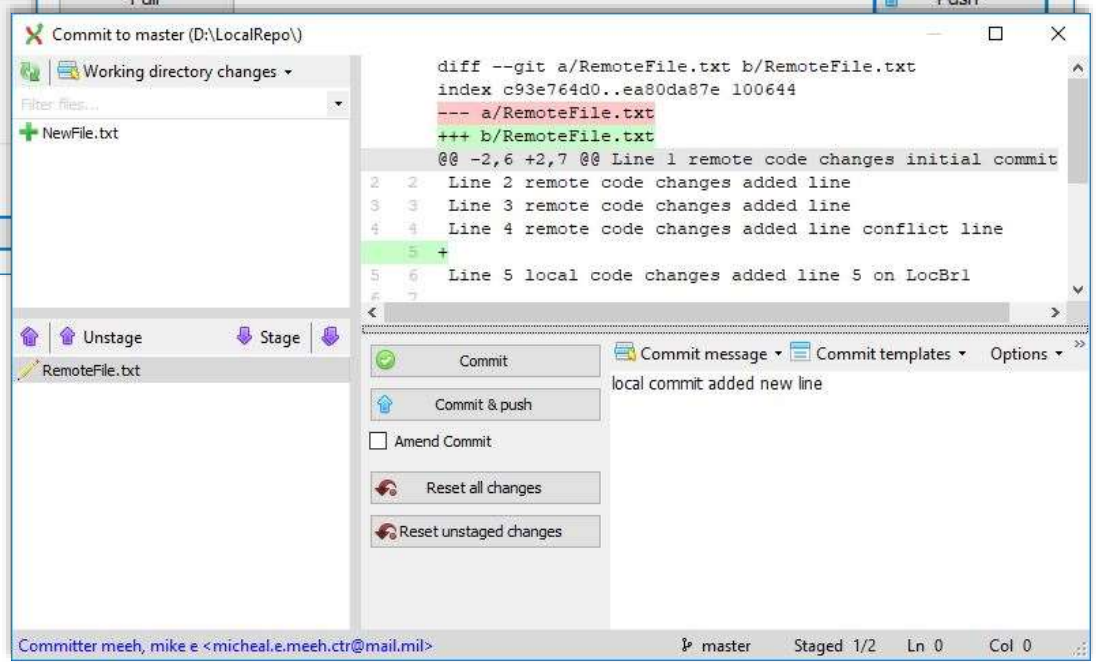
Git Extensions is a good interface when using git



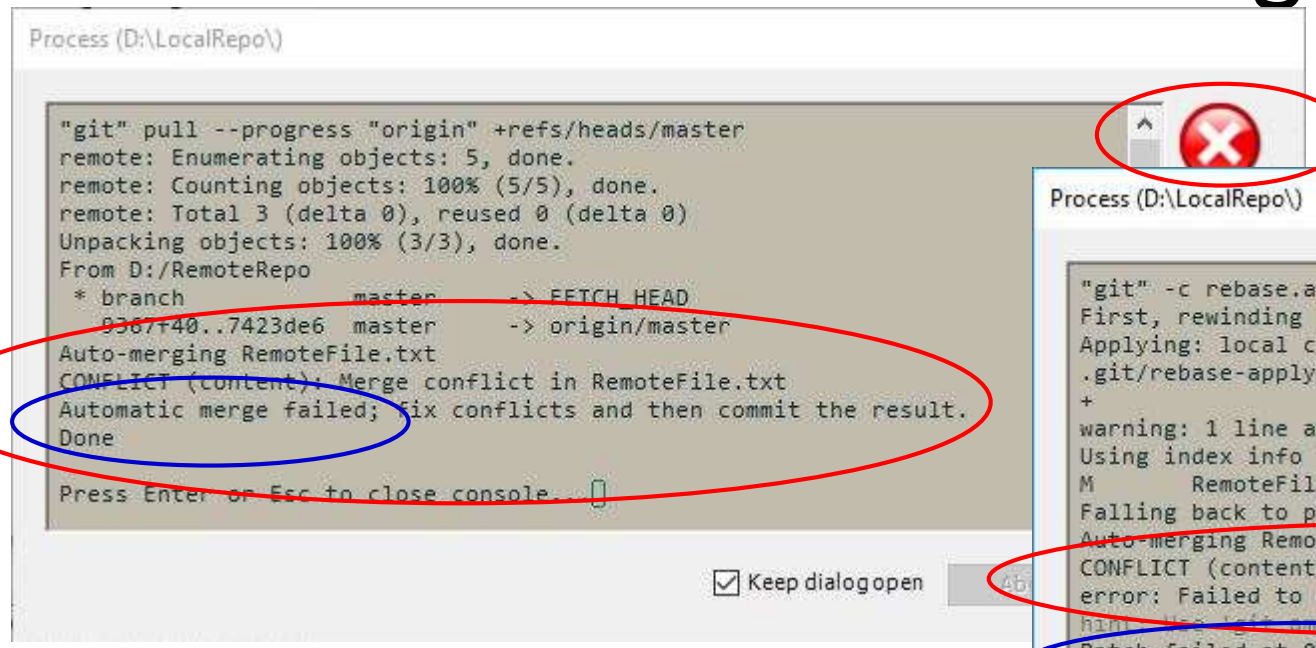
Git Extensions – Push, Pull, Commit





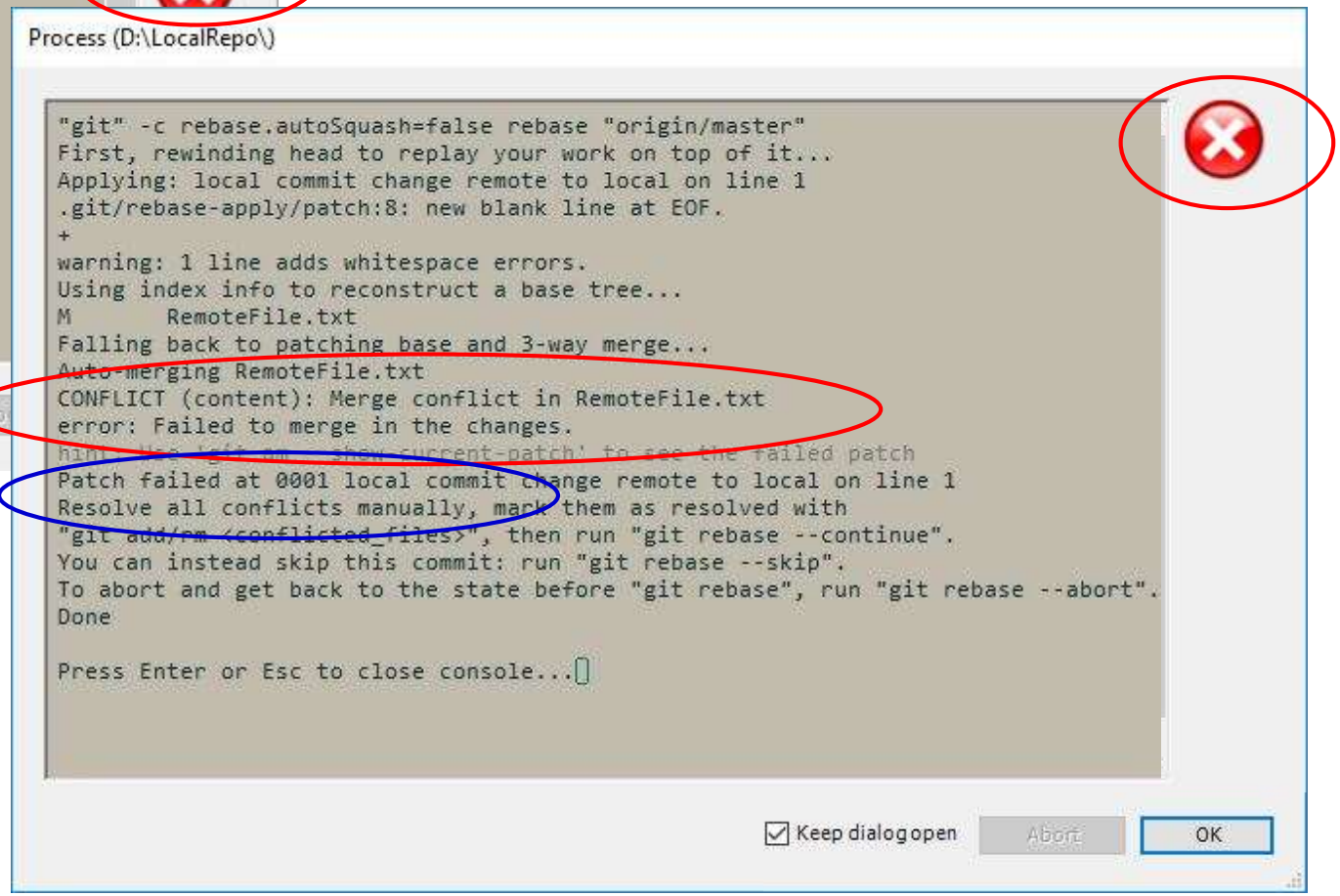


Git Extensions – Git Errors
Read all Errors and Warnings in the Shell Output

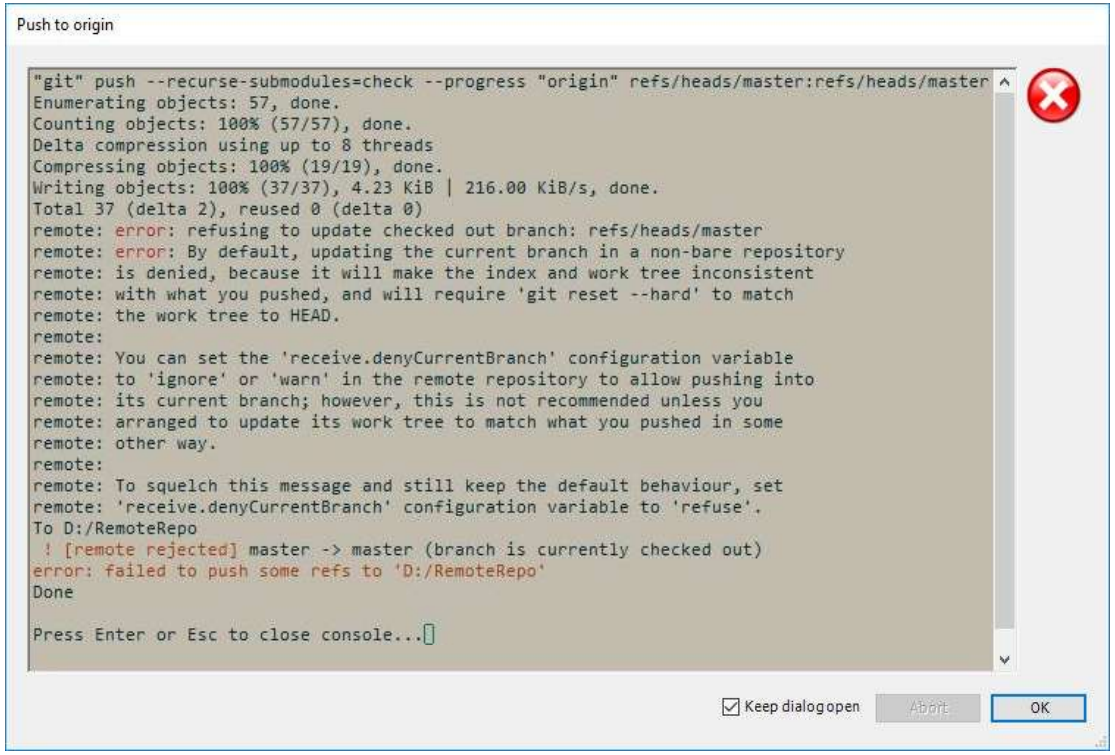


Merge conflict

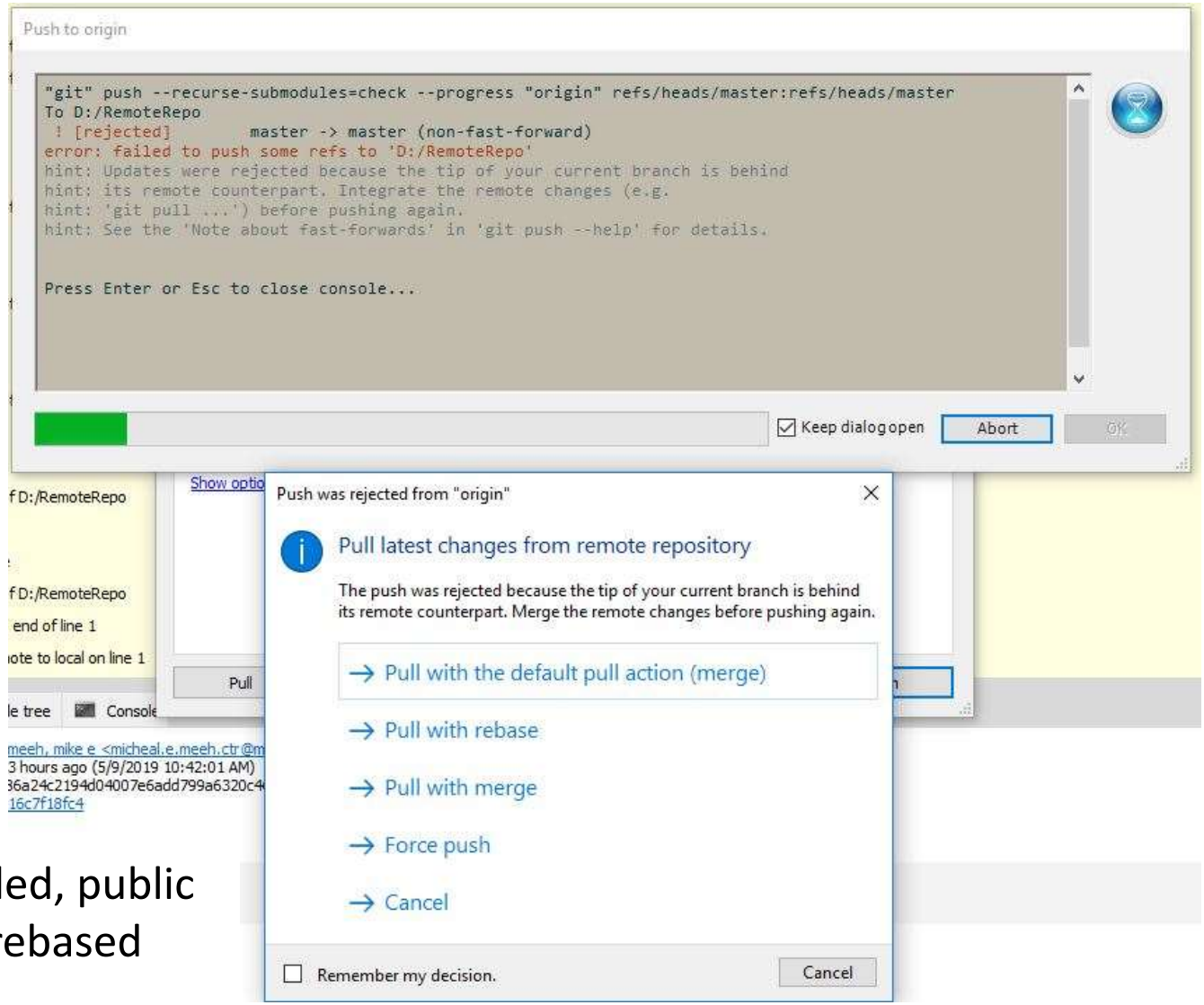
Rebase merge conflict



Git Extensions – Git Errors
Read all Errors and Warnings in the Shell Output



Push failed,
uncommitted work



Push failed, public
branch rebased

Starting New Work

1. **Refresh and Clean**
   a. *If you have work you want to save without sharing, BEFORE you fetch, commit your code (don't push it)*
   b. `fetch --prune` from the Server Repo (origin) for all branches (the default) in *Git Extensions*
   c. Examine the remote tracking branches for the Server Repo (under remotes in GIT Extensions) to ensure your prior completed branches are not shown (if still shown, see the programming lead)
2. **Create New Branch and Checkout**
   a. Create a new branch first from the Server Repo/master (origin/master) branch, using proper naming conventions
   b. `checkout` from *new branch* by keeping the *Checkout after create* box checked
   c. DO NOT `checkout` directly from origin/master

Notes

- After creation of a branch, the Working Folder is ready for your changes
- If you have to switch tasks that you are working on, deal with it by *commiting* locally if ready, or by using the stash. Be aware that the stash cannot be pushed to the Server Repo, and you can lose the stash if your disk crashes.

Branch Naming Conventions

- **Do not reuse a prior branch name**
  - To add capability to a feature, add an extension like _Part_2.
- No spaces or special characters except: `_ - ( ) .`
- Dot `.` is acceptable, except at beginning
- No double dots `..`
- Name the branch with the name of the Feature, Defect or Purpose.
- Examples:
  - `Tweak_Tank_Drive(TBD)`
  - `Initial_Vision_Processing`
  - `Defect_218`
  - `Midnight_Mayhem_Tweaks`

Getting Latest While Working

- Important to get the latest frequently while developing, so that you:
  - Stay up to date
  - Incorporate latest changes as early as possible
  - Merge in easy bite size elements
  - Avoid a nightmare merge that causes delays when you need to push
  - Simplifies the *Pull Request* process
- `git commit` to your local repo first (to prep for merge), then
- `git pull` from the Server Repo/master (origin/master) branch

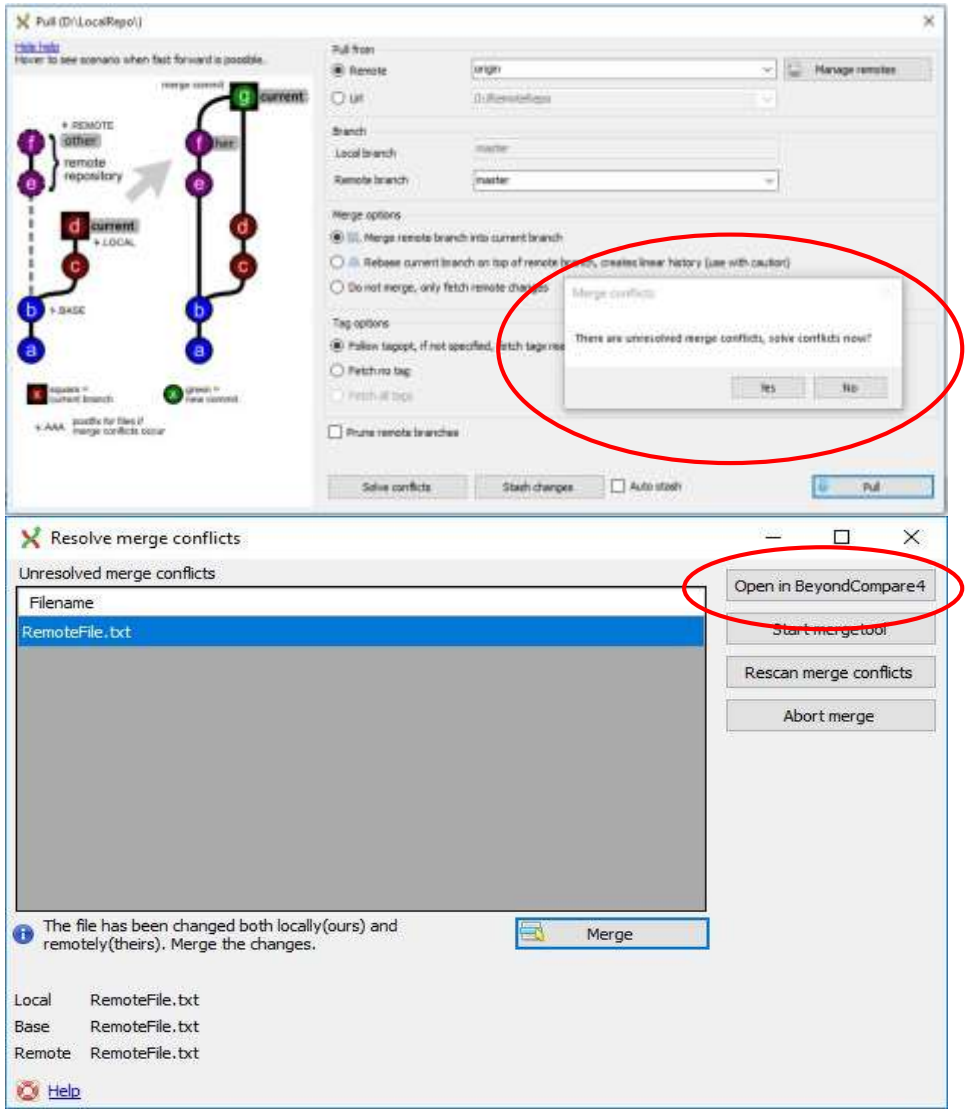
  (This will allow you to merge and resolve any conflicts)
  - May be more clear to `fetch` all and then `merge` specifically from origin/master
- Don't forget to commit and push the changes in your branch 1-2x per week so your work is backed up.
- Note that pushing your own branch *does not affect anyone else* using the Server Repo. Your changes stay in your own branch.

Pushing Latest While Working

- First time pushing the branch in Git Extensions, you should get a notice
  - “The Branch <branchname> does not have a tracking reference. Do you want to add a tracking reference to <branchname>?”
  - You should usually answer YES
    - This allows you to push to your own branch as the default, without any further warnings
- Don’t forget to commit and push the changes in your branch 1-2x per week so your work is backed up.
- Note that pushing your own branch *does not affect anyone else* using the Server Repo. Your changes stay in your own branch.
- If you are working on a collaboration branch, the other developers using that branch will get access to your updates

Merge Conflict Resolution

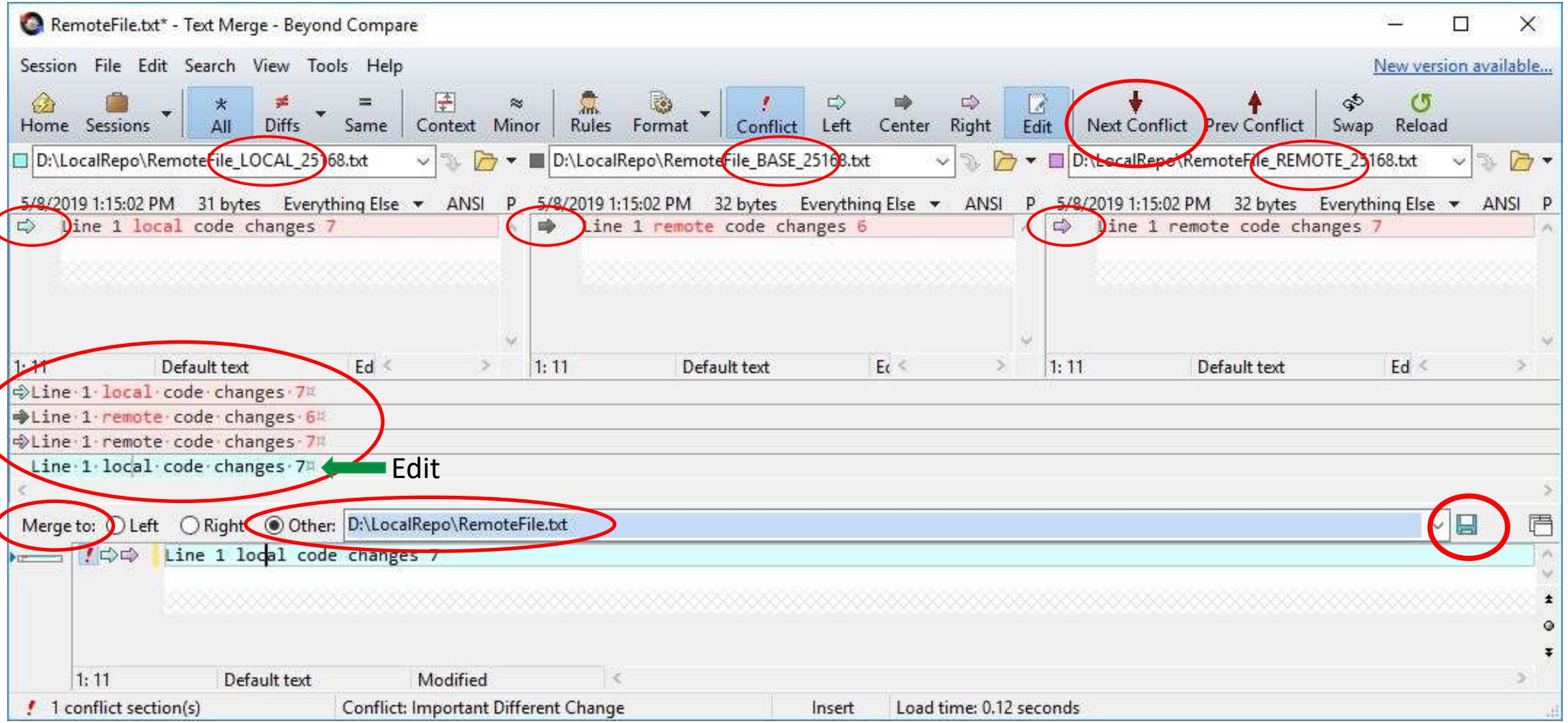
- When a merge conflict occurs during a pull (or merge), GIT Extensions displays a *Merge Conflicts* message window
  - You can choose to merge now, or later. Usually you should complete it now.
  - If you do not complete it now, ***you can*** *continue it later.* GIT Extensions will indicate that you are in the midst of a merge conflict until you complete it or abort it.
- When you choose to resolve the conflicts, it presents the *Resolve merge conflicts* window, you can choose to open in the configured tool, which could be *Beyond Compare* in MS Windows
  - Choose *Open in BeyondCompare4*
- If using Beyond Compare, 3 frames are presented
  - **LOCAL**, which holds to destination of the pull/merge, usually your local directory
  - **BASE**, the common ancestor of the two commit being merged
  - **REMOTE**, the source of the merge, usually your remote repo



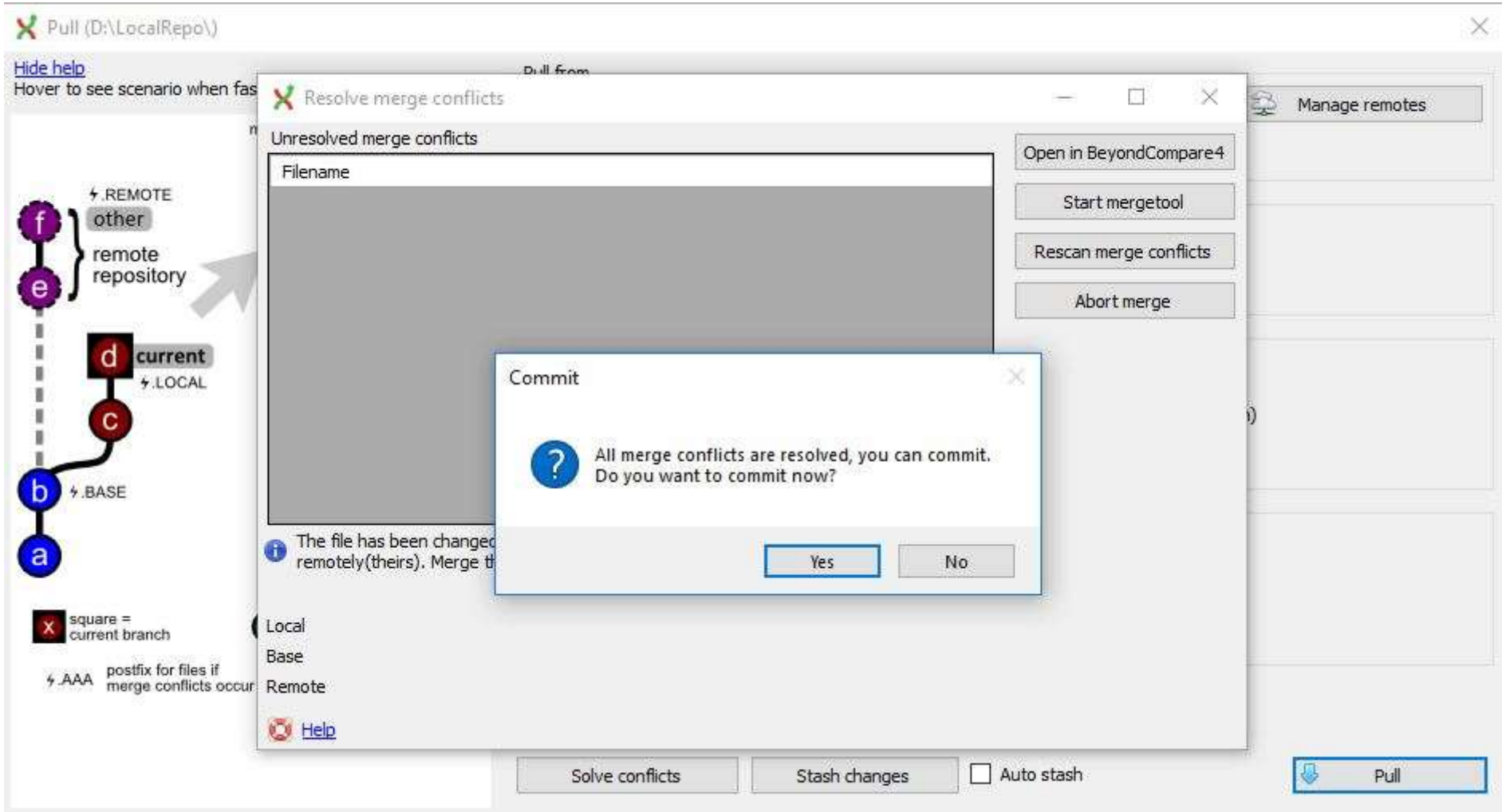
Merge Conflict Resolution in Beyond Compare

- The bottom frame is the *Merge to file*, where your selection of conflict resolution actions is placed
  - You are offered a selection of Left/Right/Other. **You MUST leave it as *Other***, with the default file.
- Scroll the upper frames or use the *Next Conflict* button to navigate to conflict sections
- Press the arrows to the left of the conflict you wish to choose – LOCAL, BASE, or REMOTE
- There is also a section in the middle for single line selections and *modifications*
  - You can pick specific lines to choose, in order to mix and match selections
  - In the last line, you can edit the resulting resolution!

Merge Conflict Resolution in Beyond Compare



Completion of Merge Conflict Resolution



Completing Work – First Steps

Overview: Commit, Get Latest (pull), Build and Test, Push to Server, Complete Pull Request, Clean Up Local Repo

1. **Commit**
   a. Use Git Extensions to **select all items** (`git add`) to add to the commit (usually all)
   b. Use Git Extensions to **commit changes** (`git commit`)
2. **Get Latest (pull) from Server**
   a. `git pull` from the Server Repo/master (origin/master) branch
   (This will allow you to merge and resolve any conflicts)
3. **Build and Test (again)**
   - If any changes, return to Commit - Step 1
4. **Push branch to Server Repo**
   - You are pushing your development branch up to the server, still identified as your branch name

Completing Work – the Pull Request

5. **Generate and Complete a *Pull Request* in TBD**

   (This is to bring your commits into *master* in the Server Repo, which is protected)

   a. **Use** *Pull Requests* in TBD to initiate a pull request on the *pushed branch*
   b. This initiates an automated build and starts the review process (TBD)
   c. Confirm that your build has succeeded (If not, TBD indicates *Build All Solutions Failed)*
   d. After the *Pull Request* is Approved, **Complete** the request by pressing the *Complete* button in TFS (without performing further work on the branch). TBD
   e. This completes the *squash* merge into master. Your branch is then automatically deleted from the Server Repo. TBD

Completing Work – Cleanup

6. **Clean up Local Repo**
   a. In Git Extensions `fetch --prune` from the Server Repo (origin) for all branches (the default)
   b. Examine the remote tracking branches (under remotes in GIT Extensions) for the Server Repo to ensure your completed branch is not shown (if still shown, see the Programming Lead)
   c. Switch to the master branch (via checkout) in the local repo, and
   d. Delete the branch you developed on (force delete)