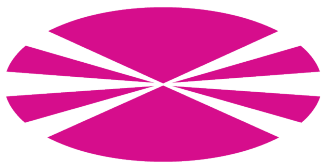
Practice 4: Git

Diseño Software (614G01015)

Eduardo Mosqueira Rey (Coordinador)

Departamento de Ciencias de la Computación y Tecnologías de la Información
Facultad de Informática



UNIVERSIDADE DA CORUÑA

Table Of Contents

Version Control

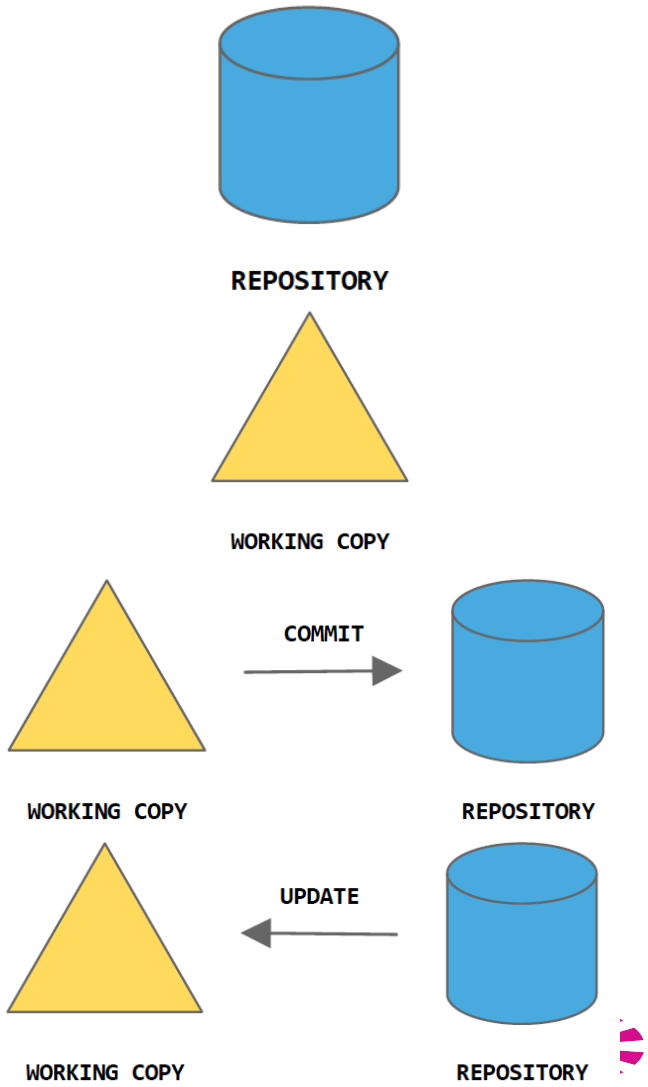
Version Control System (*VCS*)

A system that records the changes made over time in a file or set of files, so you can go back to specific versions later.

- Functioning is more effective with plain-text files that can be simultaneously modified by different users.
- Gives you a backup of the developed code.
- Gives you a log of the changes made in the code.
- Lets you share code easily, facilitating collaborative development.

Basic Concepts

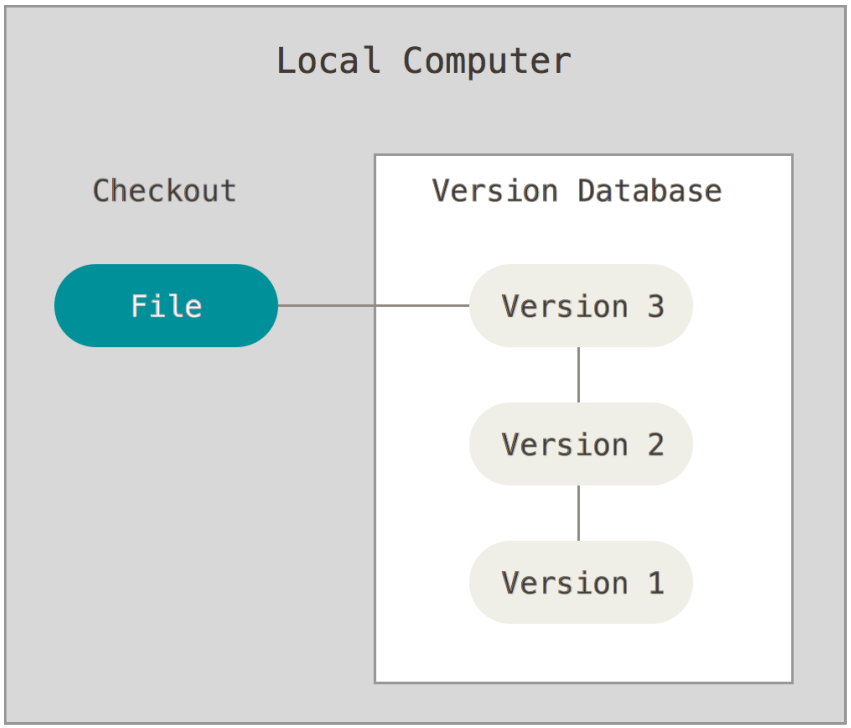
- **Repository**:
  - The place were the source code is stored in all its different versions.
- **Working copy**:
  - The current version of the code that is under development and subject to changes.
  - Visible only for the local user that is editing it.
- **Commit**:
  - Moves the working copy to the repository.
  - Includes an explainatory text and generates an identifier to indicate the version.
- **Update**:
  - Brings the working copy up to date with the repository, merging the changes in both.
  - Can result in conflicts if two users have modified the same line of code concunrrently.



Local Version Control

Local Version Control

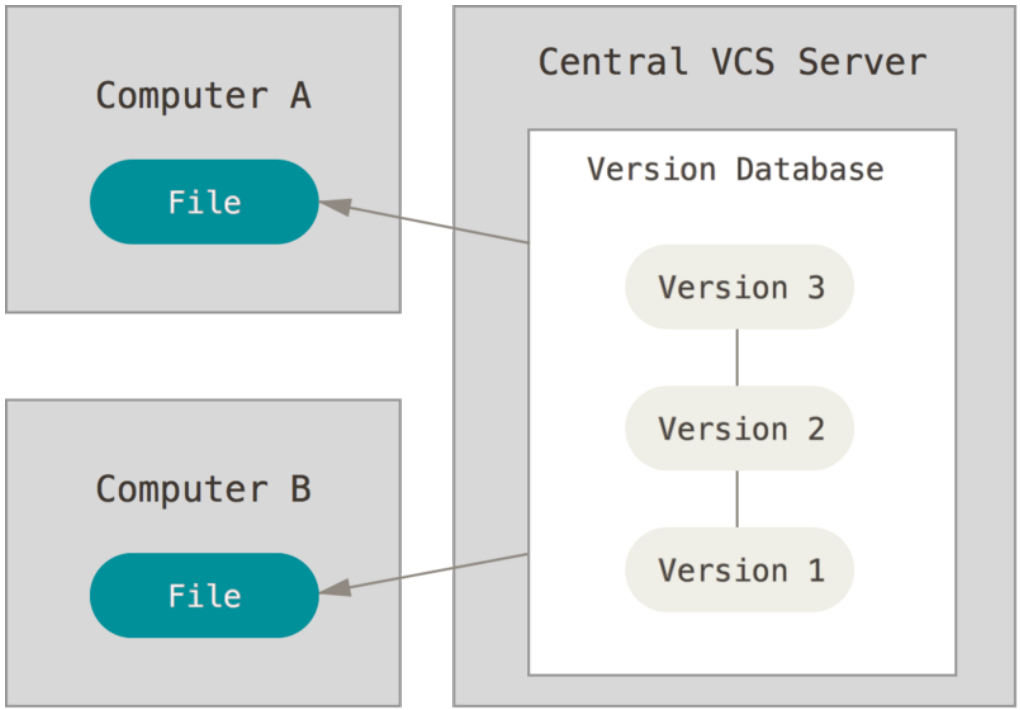
Both the working copy and the repository are stored in the local computer.



Centralized Version Control

Centralized Version Control (*Centralized VCS or CVCS*)

The repository is in a server in the cloud. Multiple clients download their own local copies from that place (Subversion).



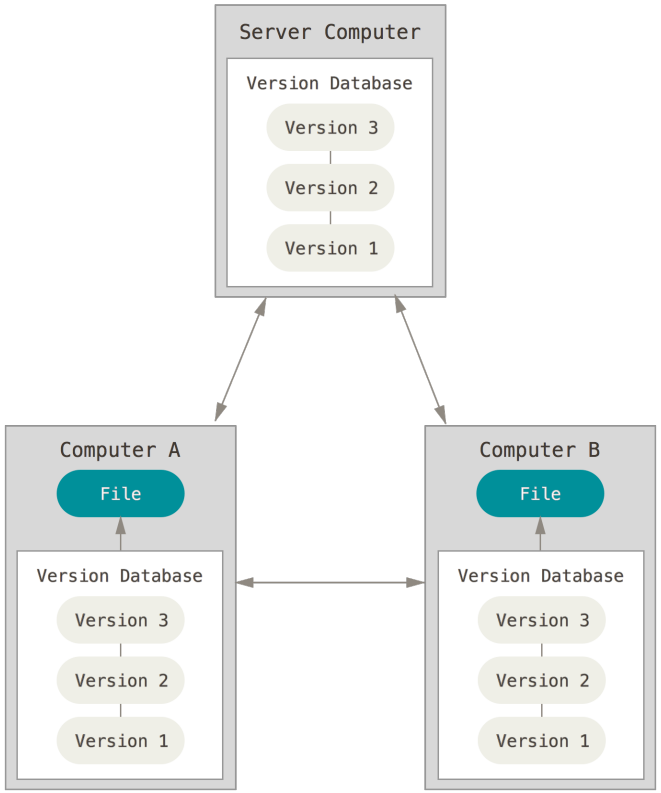
Centralized Version Control

- **Advantages**
  - Facilitates collaborative development.
  - Everybody can know what the other collaborators are working on.
  - Administrators have extensive control over what each user can do.
- **Disadvantages**
  - As development is centralized in the server, the workflow is not robust to server problems.
  - If the server goes down, the collaborative workflow is interrupted.
  - If there are storage failures in the server, and no *backups* have been made, the project's history may be lost (only local snapshots would survive).

Distributed Version Control

Distributed Version Control (*Distributed VCS or DVCS*)

It also consists of a repository in a remote server and local working copies for the clients but, additionally, clients also store a copy (a clone) of the remote repository (Git or Mercurial).

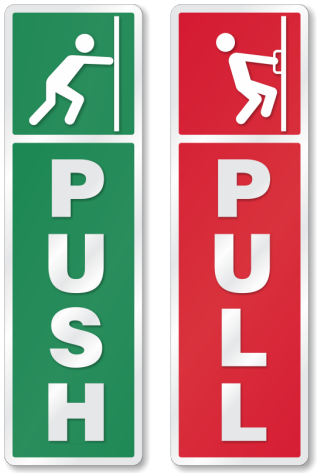


Distributed Version Control

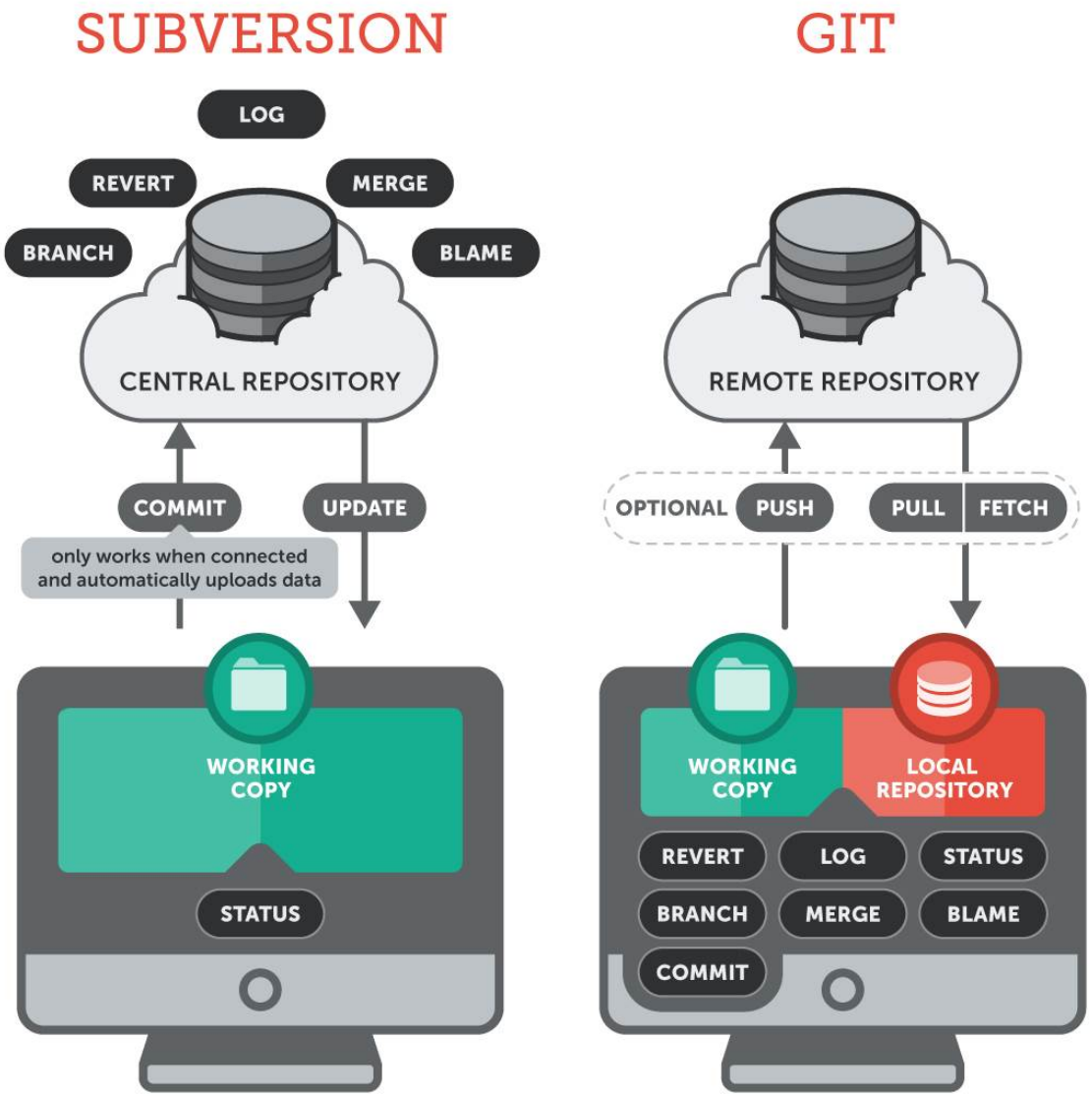
- **Advantages**
  - If a server stops working, any of the repositories available on the client side could restore it (each clone is complete copy of the repository).
  - Allows establishing workflows that are not possible for centralized systems.
  - Most of the operations are done locally, so you don't need an internet connection to use it.
- **Disadvantages**
  - Adds an extra layer of complexity compared to centralized versions.
  - You must follow an orderly workflow to avoid problems when dealing with the different options that are available.

Distributed Version Control

- Distributed VCSs add new operations to the usual *commit* and *update* operations. Namely, they add two operations called *push* and *pull* for communicating with the remote repository.
  - **Push**: Updates the remote repository with the contents of the local repository.
  - **Pull**: Updates the working copy and the local repository with the new changes at the remote repository.



Centralized vs. Distributed Workflow

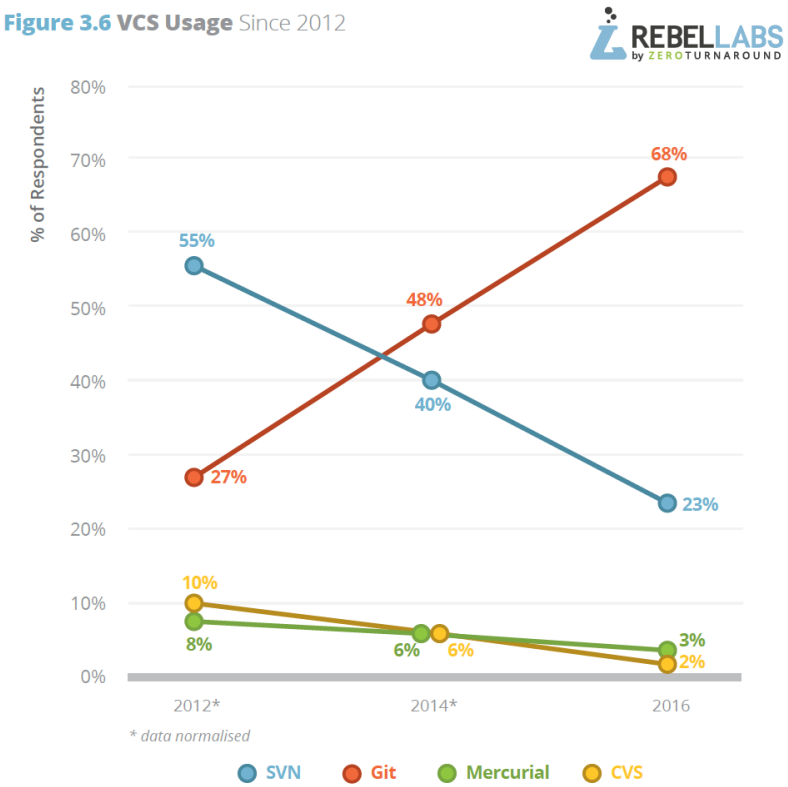


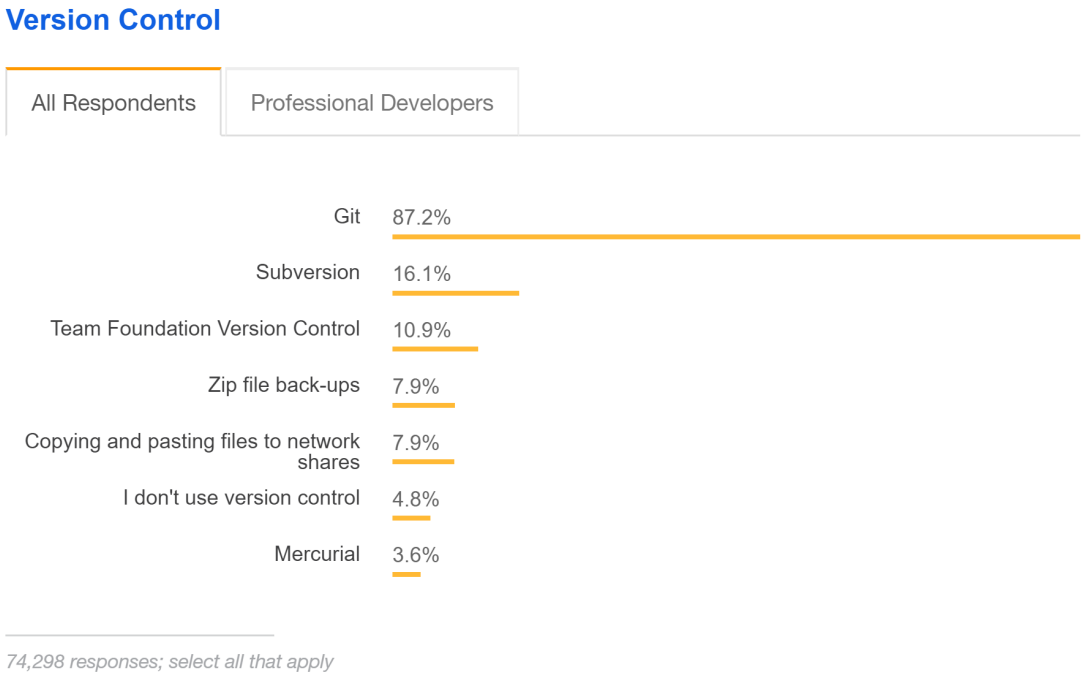
The Git version control system

- Created by Linus Torvalds in 2005 for *Linux* development.
- It is a distributed version control system.
- Its goals are influenced by the needs of a large-scale project like Linux, i.e., speed, simple design, big support for non-linear development (thousands of parallel branches), suitable for managing big projects, etc.

VCS Technologies

- Git currently dominates VCSs, to the detriment of the formerly more popular SVN[1].





[1] https://www.jrebel.com/blog/java-tools-and-technologies-2016
https://insights.stackoverflow.com/survey/2018

Git solutions in the cloud

- Git became popular mainly thanks to cloud services, which add a social layer to code development.
- The most popular one is **GitHub** but widely-used alternatives exist, such as **BitBucket** or **GitLab**.
- GitLab has an *open source* version that you can download and install at your own server for free.

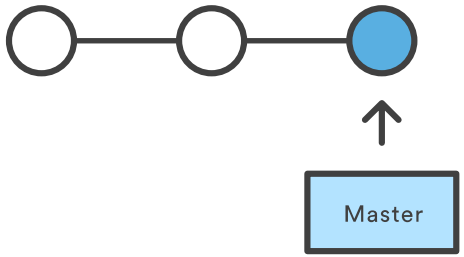
GitHub

Bitbucket

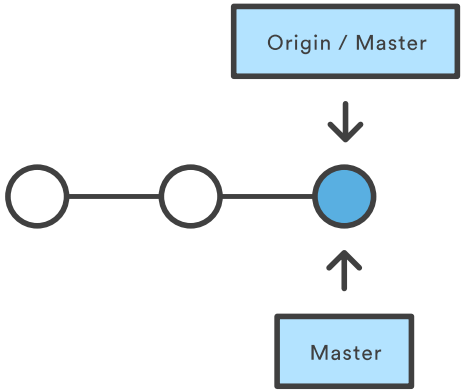
GitLab

Git Concepts

- **`Master` branch**:
  - Git structures source code in branches. There is always a default branch, usually called `master`.
  - `Master` basically points to the latest revision of the code committed to the local repository.



- **`Origin/Master` branch**:
  - Git has a remote repository called `origin` by default.
  - The remote repository has its own *master* branch, called `origin/master`.
  - When we download a repository, both *master* branches point to the same place.

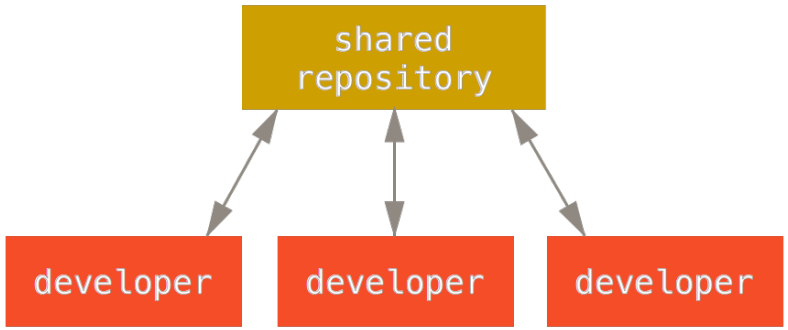


Centralized workflow

Centralized workflow

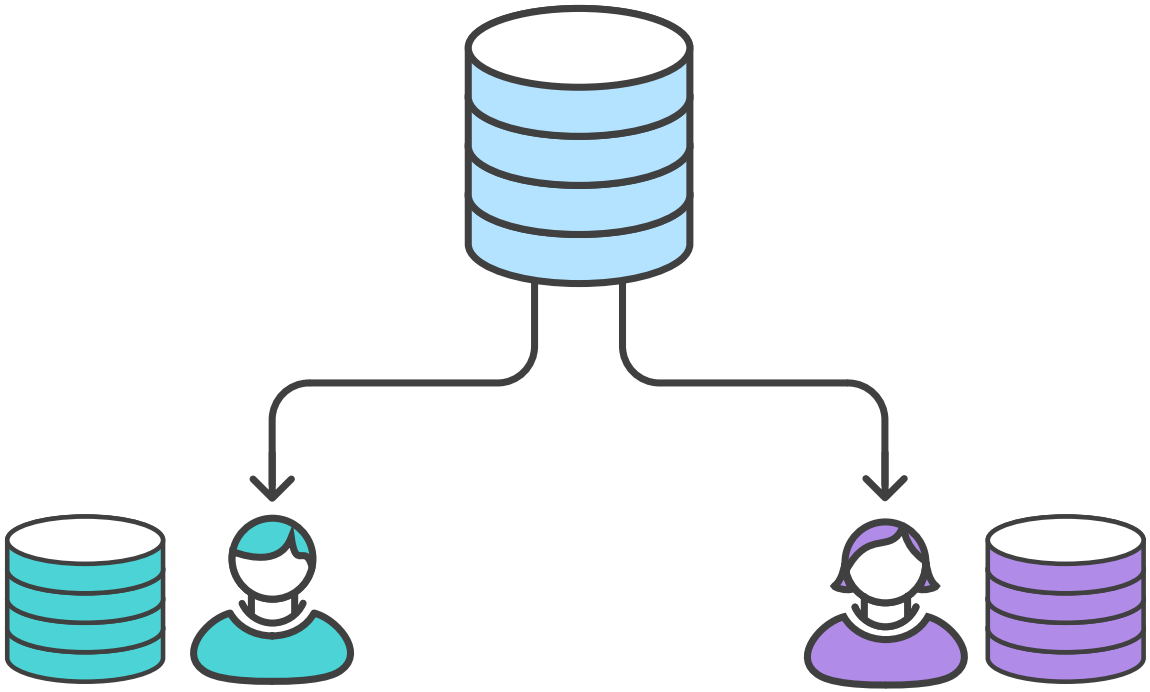
A central repository maintains the work of all developers and all of them synchonize with it.

- It is a similar workflow to that of centralized systems like Subversion.
- Useful por simple projects with few developers.
- Does not really exploit Git's capabilities, but it is suitable for this course.
- Other, more complex workflows can be consulted at: `https://www.atlassian.com/git/tutorials/comparing-workflows`



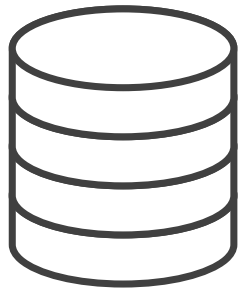
Centralized workflow

- You create a central repository that is locally cloned by all users.



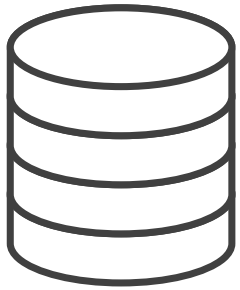
Centralized workflow

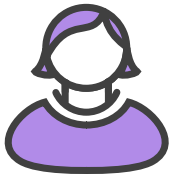
- Imagine John is working on a particular feature.



Centralized workflow

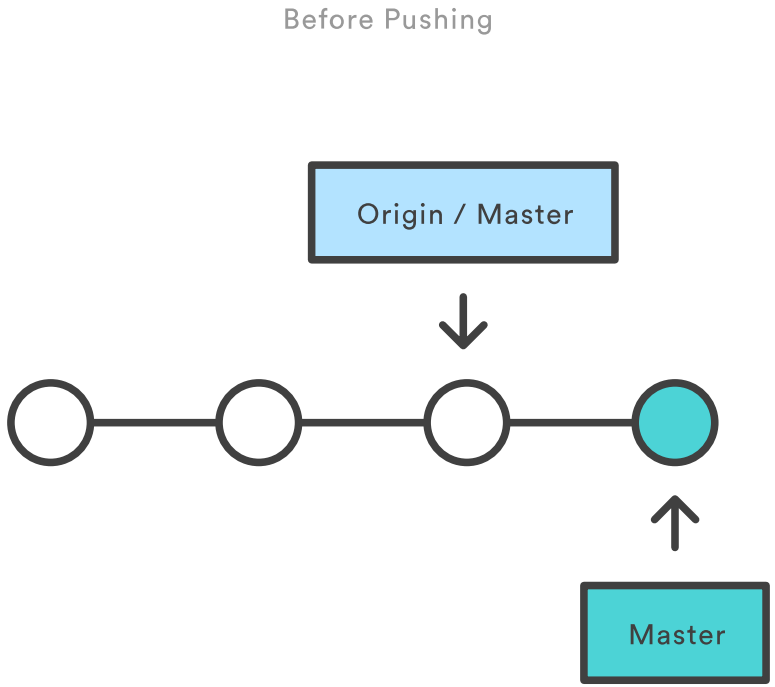
- Mary is working on another feature in parallel.





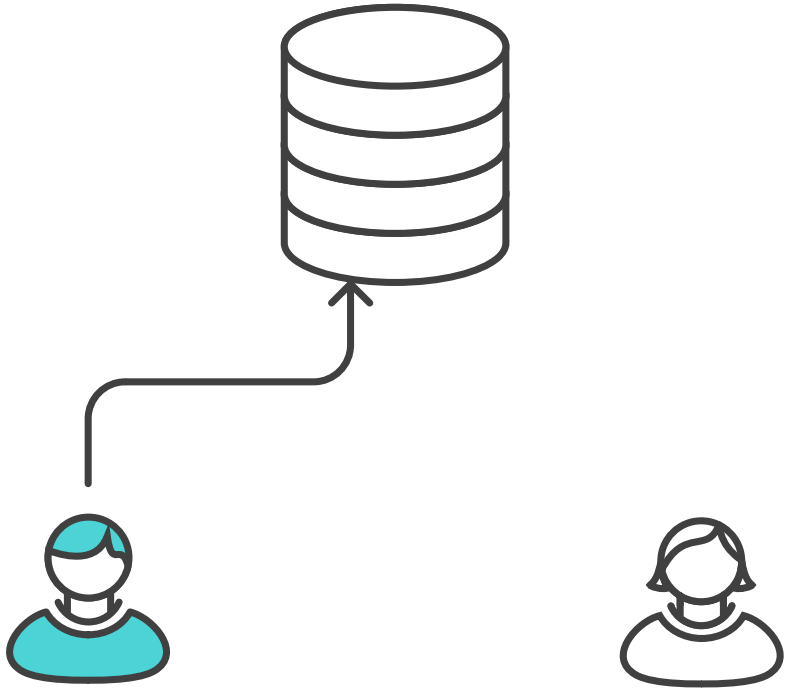
Centralized workflow

- John *commits* his changes to his local repository.
- Is `master` branch is ahead of the `origin/master` branch.



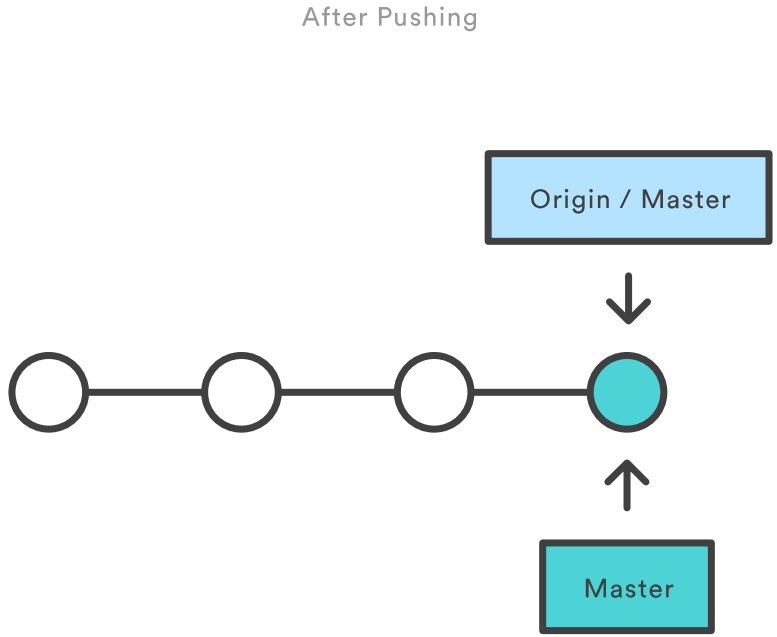
Centralized workflow

- John *push*es his changes to the central repository.



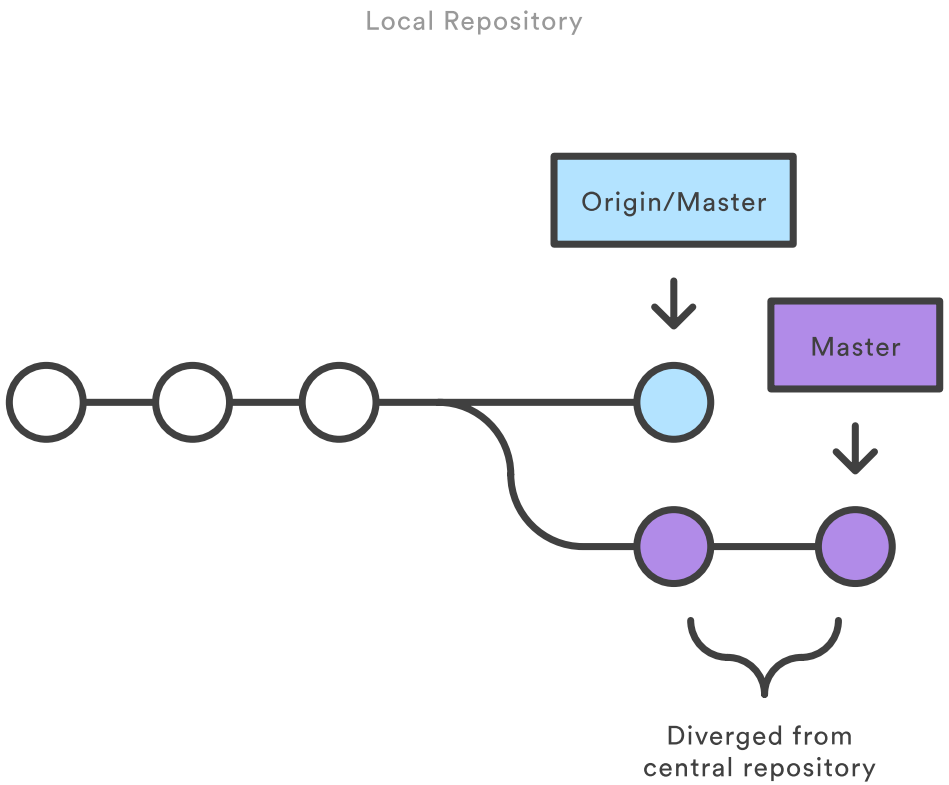
Centralized workflow

- To John, his local `master` branch and the `origin/master` branch point to the same place.



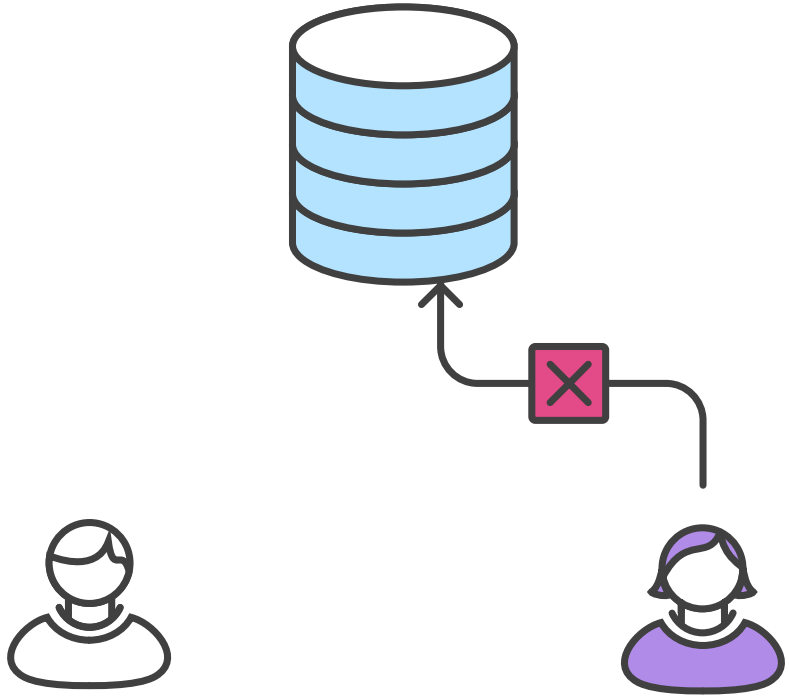
Centralized workflow

- Mary *commits* her changes to her local repository, unaware that the remote repository has changed.
- This means that Mary's changes diverge from the central repository.



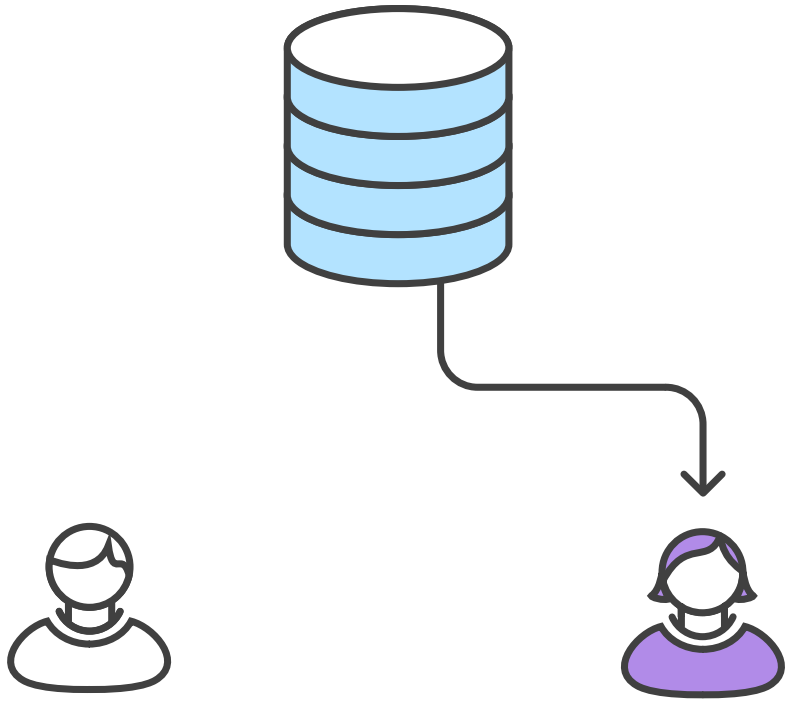
Centralized workflow

- When Mary tries to *push* her changes to the remote repository, she gets an error, because it has changes that are not in her local repository.



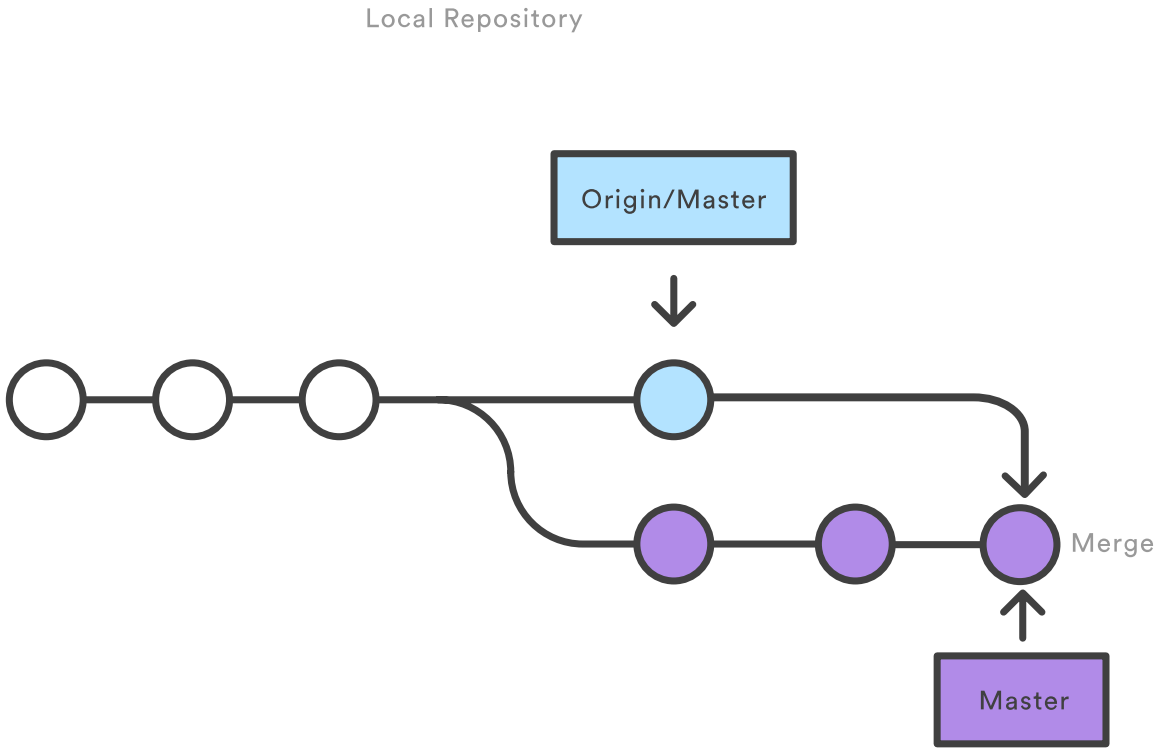
Centralized workflow

- Mary has to *pull* the remote changes and *merge* them with hers.

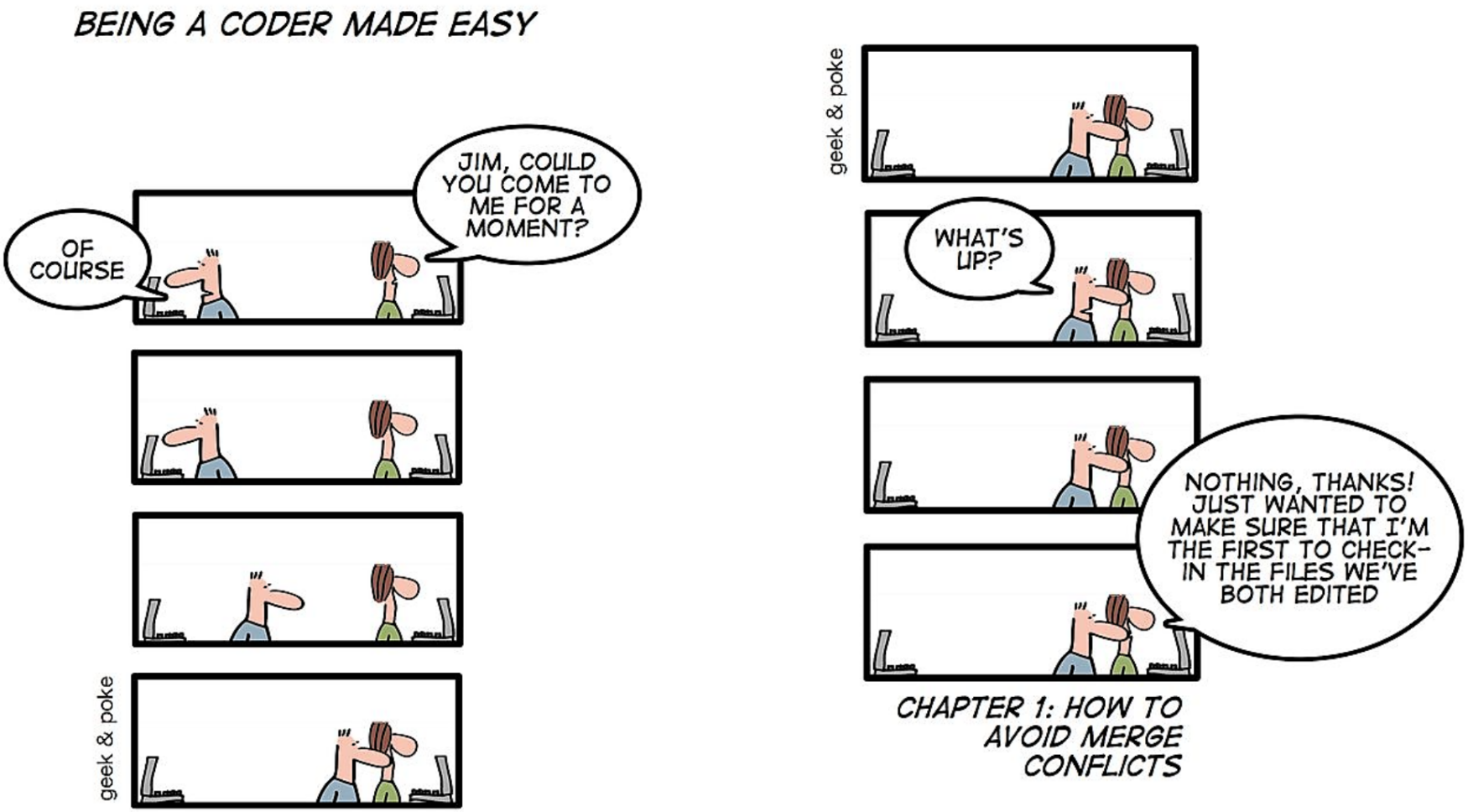


Centralized workflow

- The *merge* may work correctly, or it may cause conflicts that Mary has to resolve.
- Once they've been resolved, a new local *commit* is created containing both John and Mary's changes.

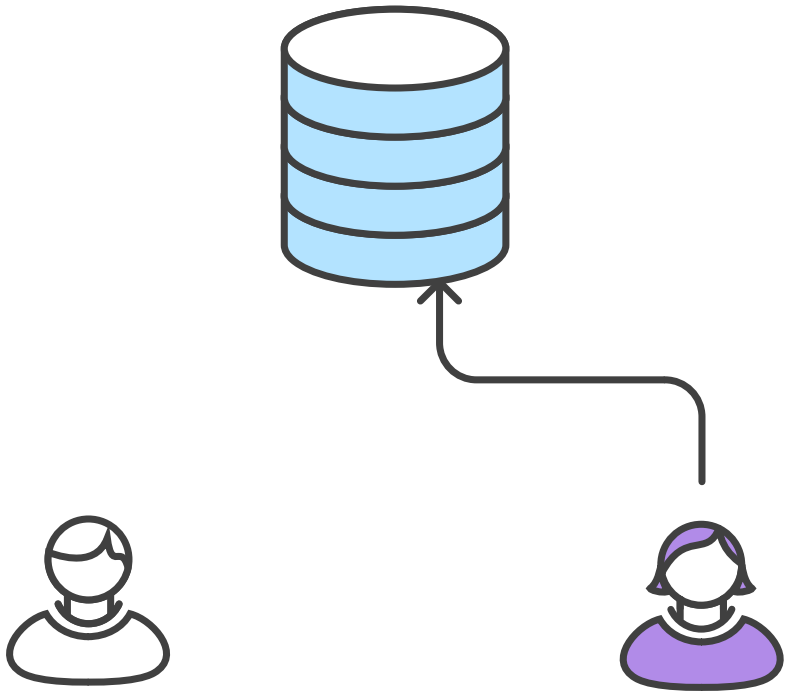


It is important to be the first to *commit* changes...



Centralized workflow

- When the *merge* has been made successfully, Mary must *push* it to upload the resulting changes to the central repository for the other developers.



Centralized workflow

- Now, John's repository is the one that is behind, compared to the remote repository, so John must update it doing a *pull*.

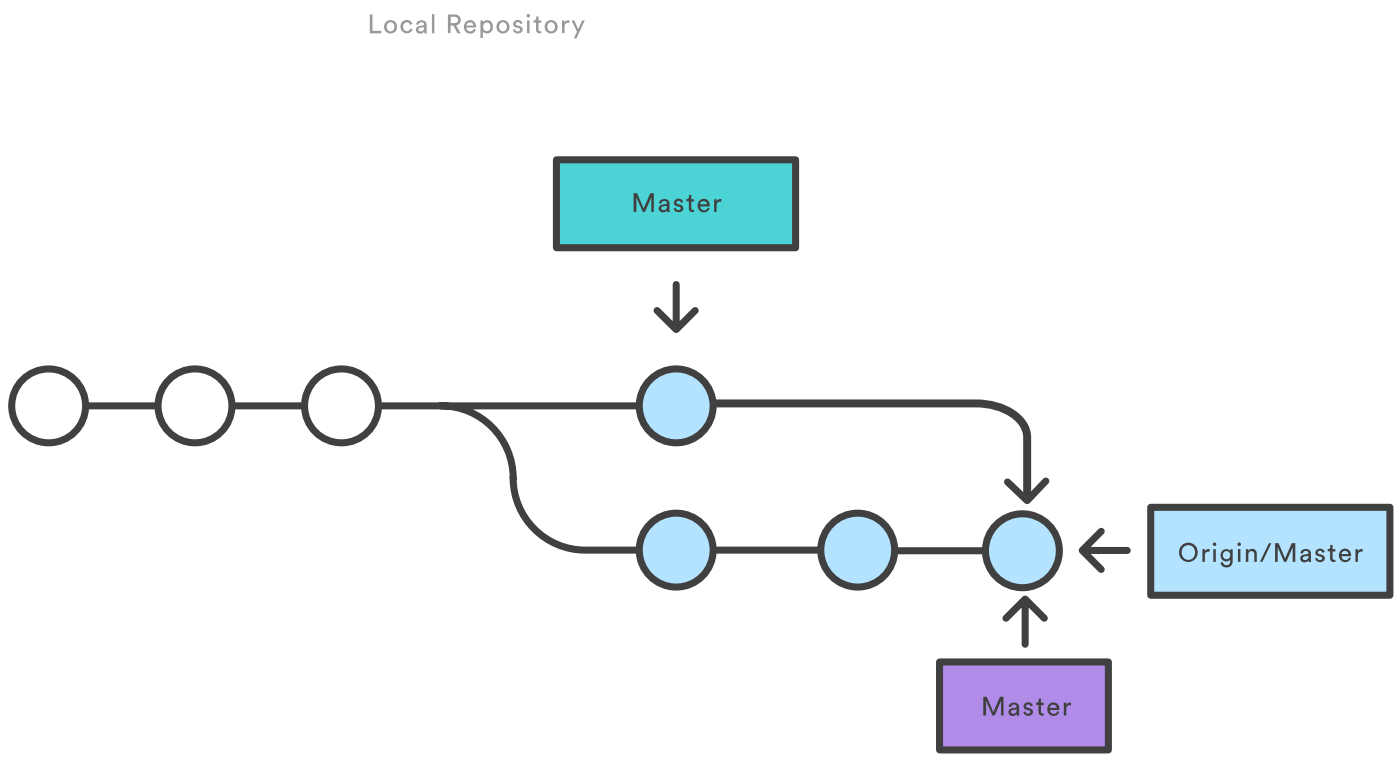


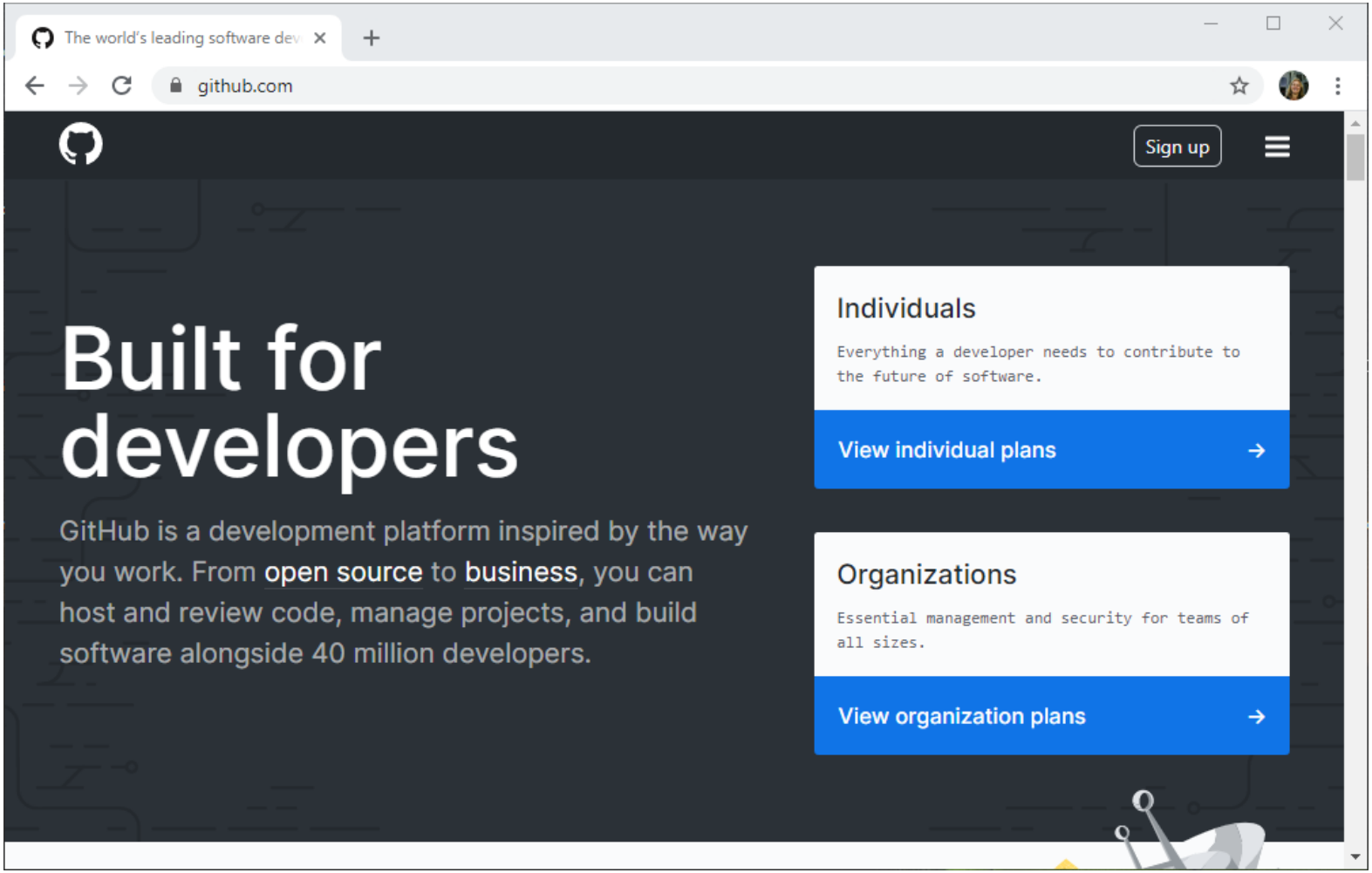
Table Of Contents

Git Repository on GitHub

- We will create a repository in Git by following the steps below:
  1. We will create a GitHub account (if not already created).
  2. One of the group members will create the repository with the name of the group.
  3. He or she will then include the other member of the group as a collaborator.
  4. Finally, the teachers of the subject as collaborators must be included as collaborators too.

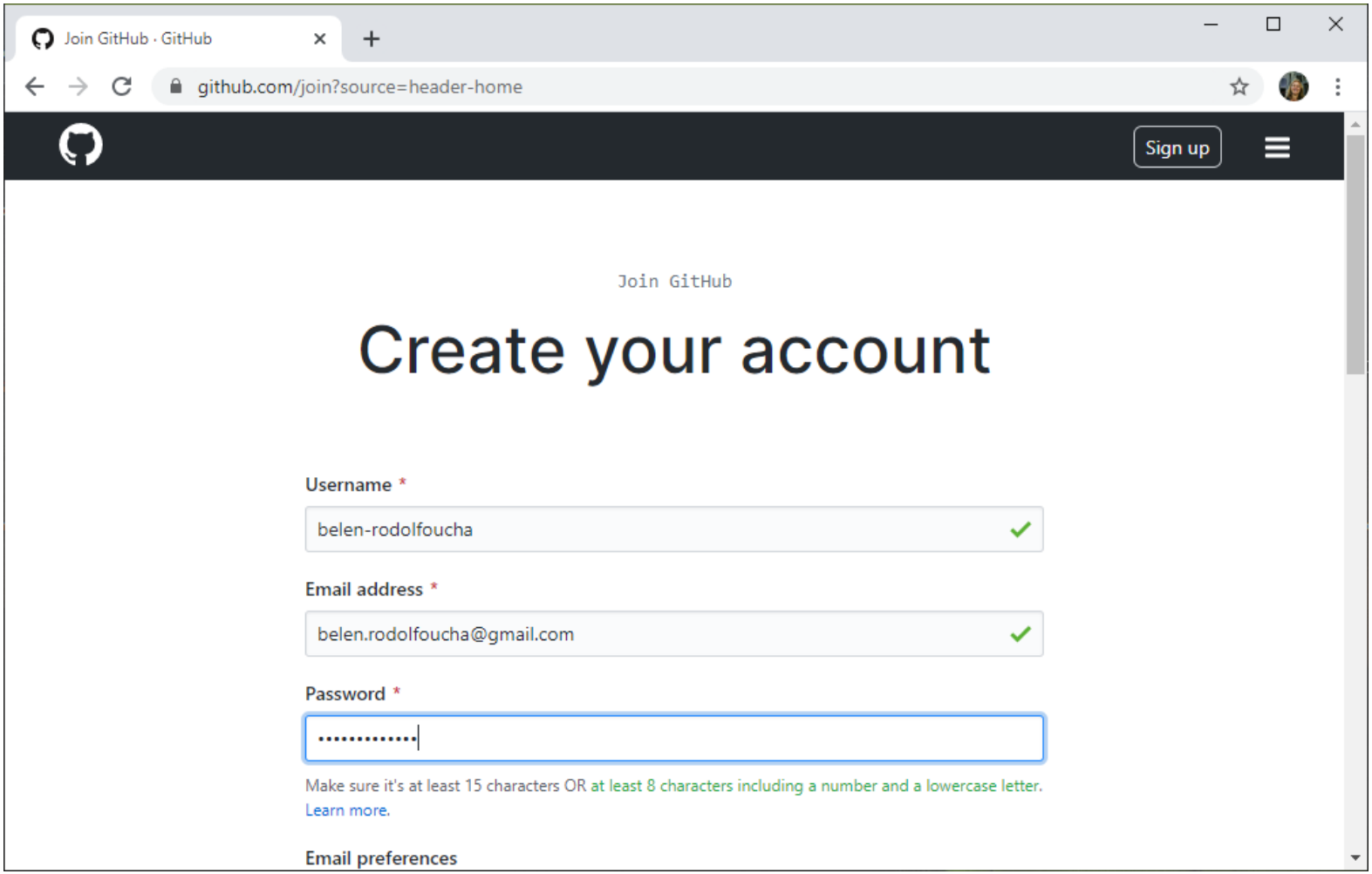
Creating an account on GitHub

- We will go to `https://github.com/` to create an account (if we don't already have one).



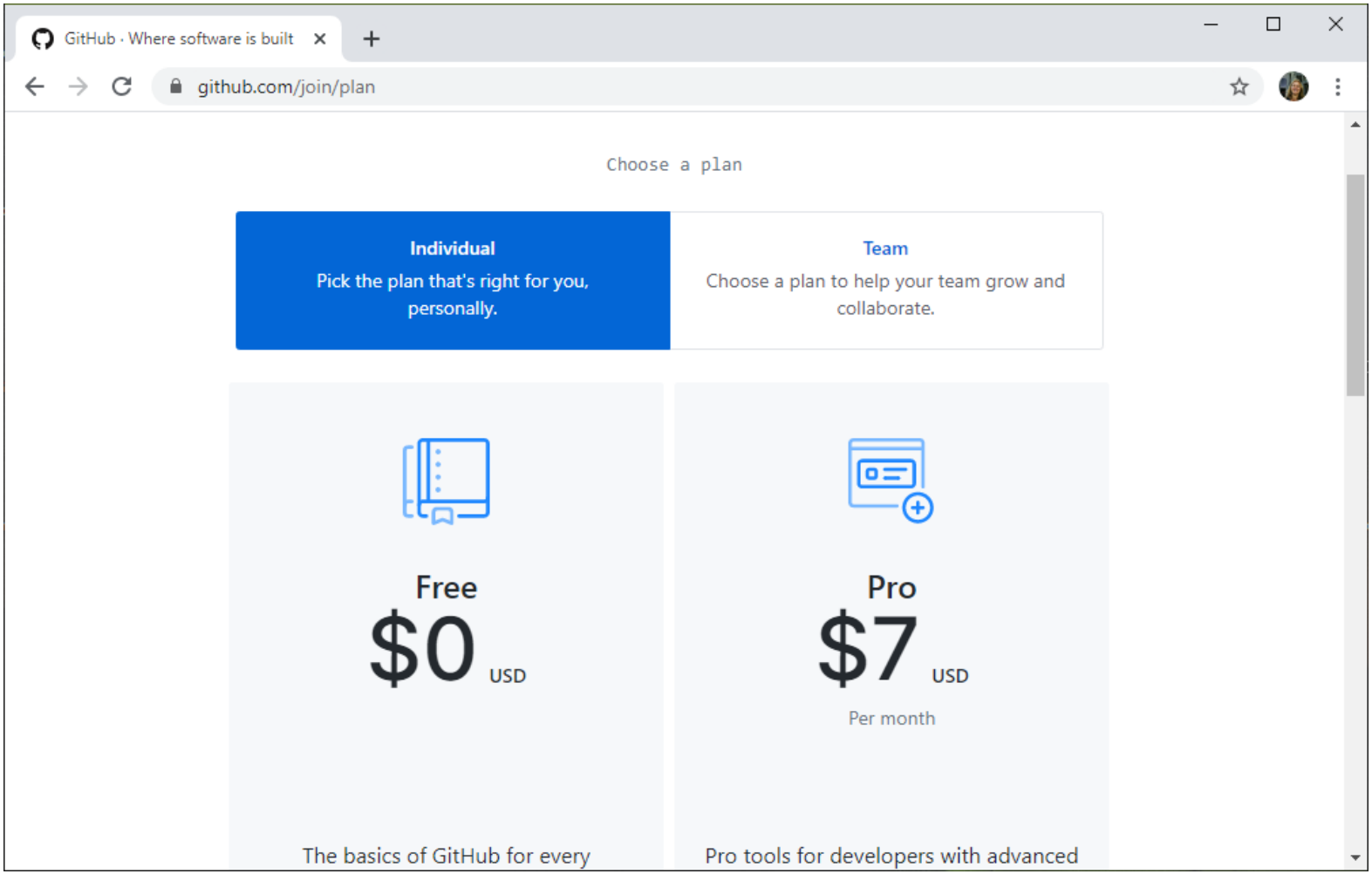
Creating an account on GitHub

- We will need to provide a username, email address and password.



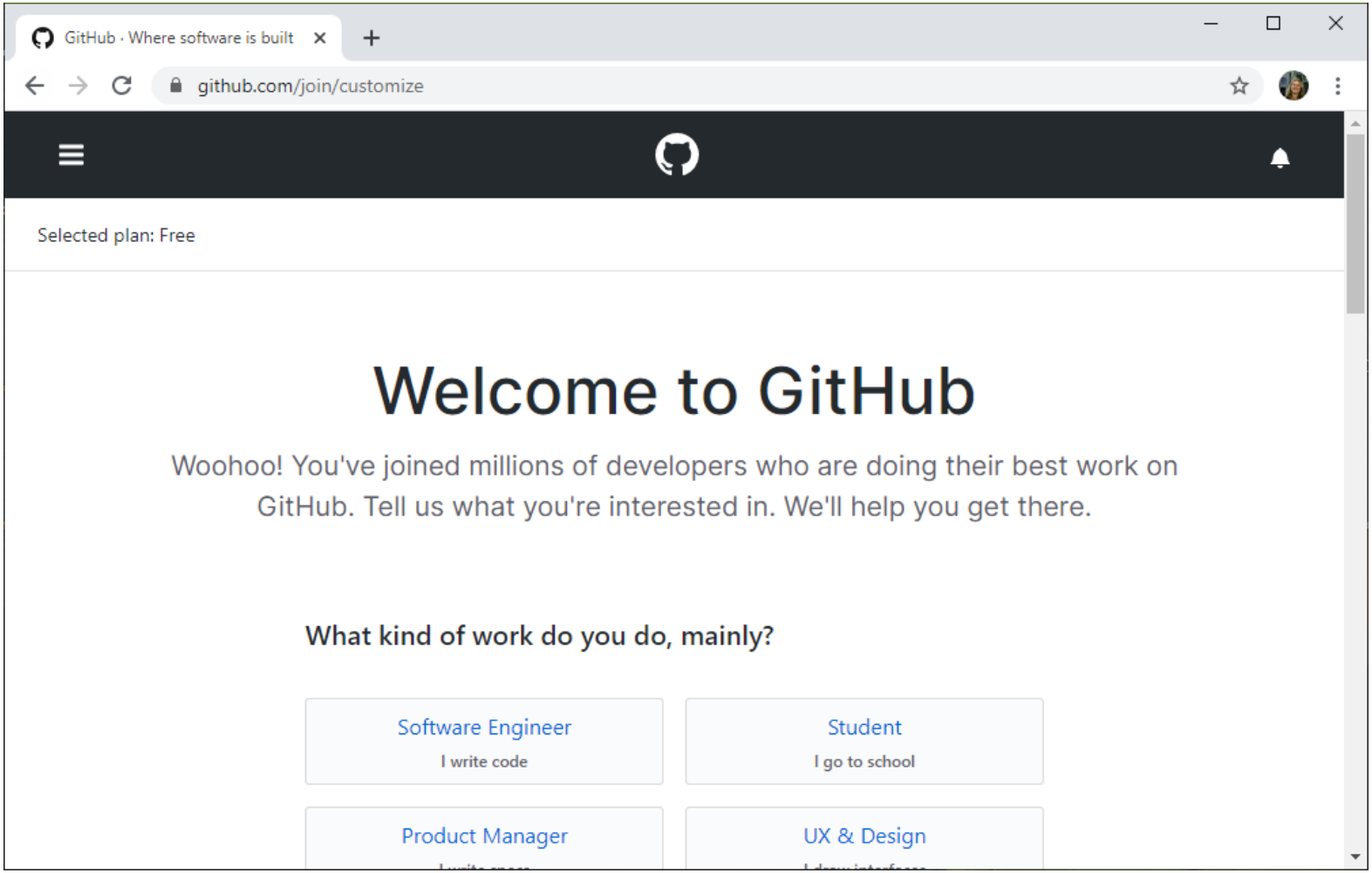
Creating an account on GitHub

- We will create a free account. It has certain restrictions but they do not affect us.



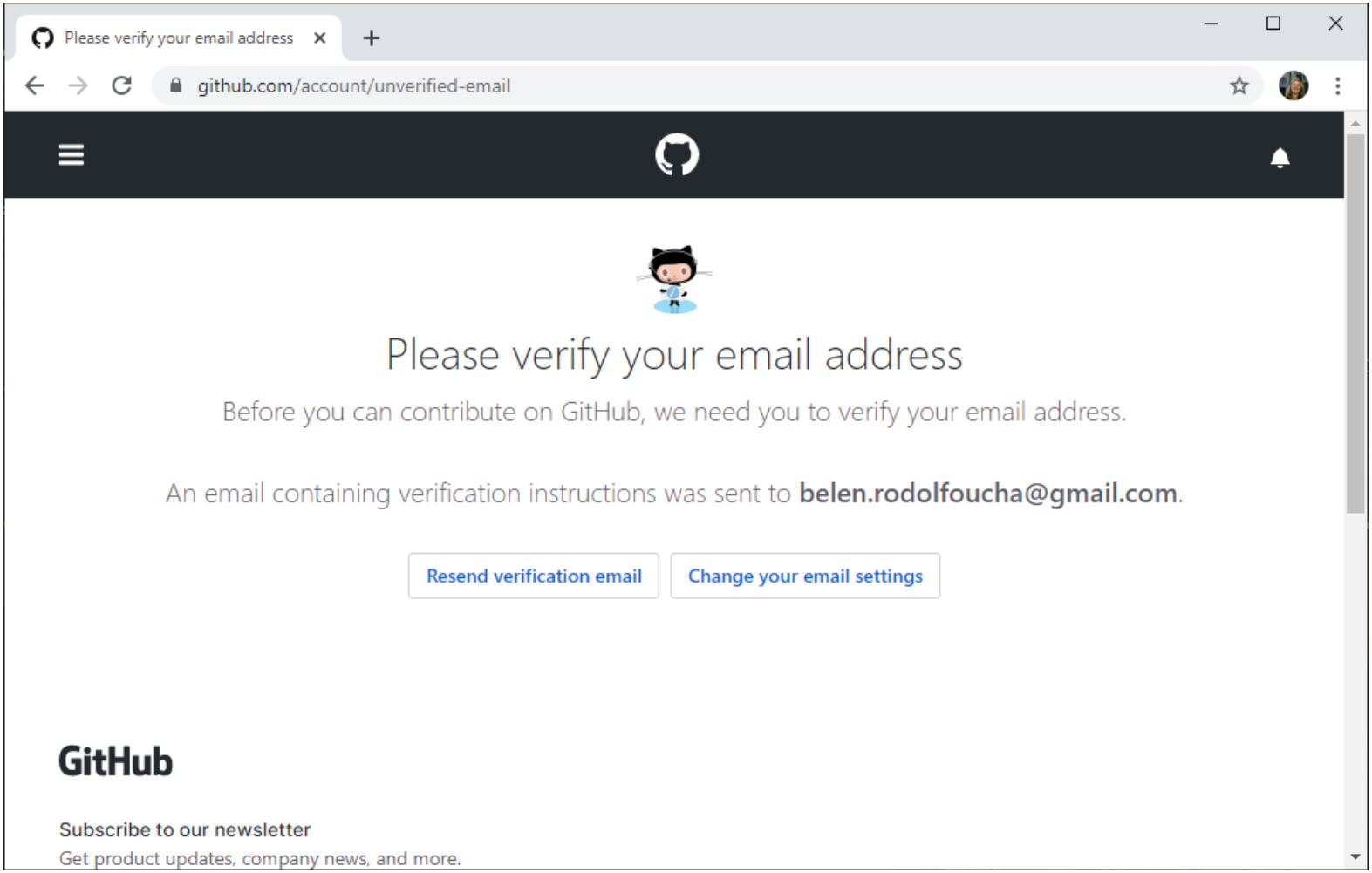
Creating an account on GitHub

- GitHub asks you questions to get to know you better, but you can skip this step if you're not interested in answering them.



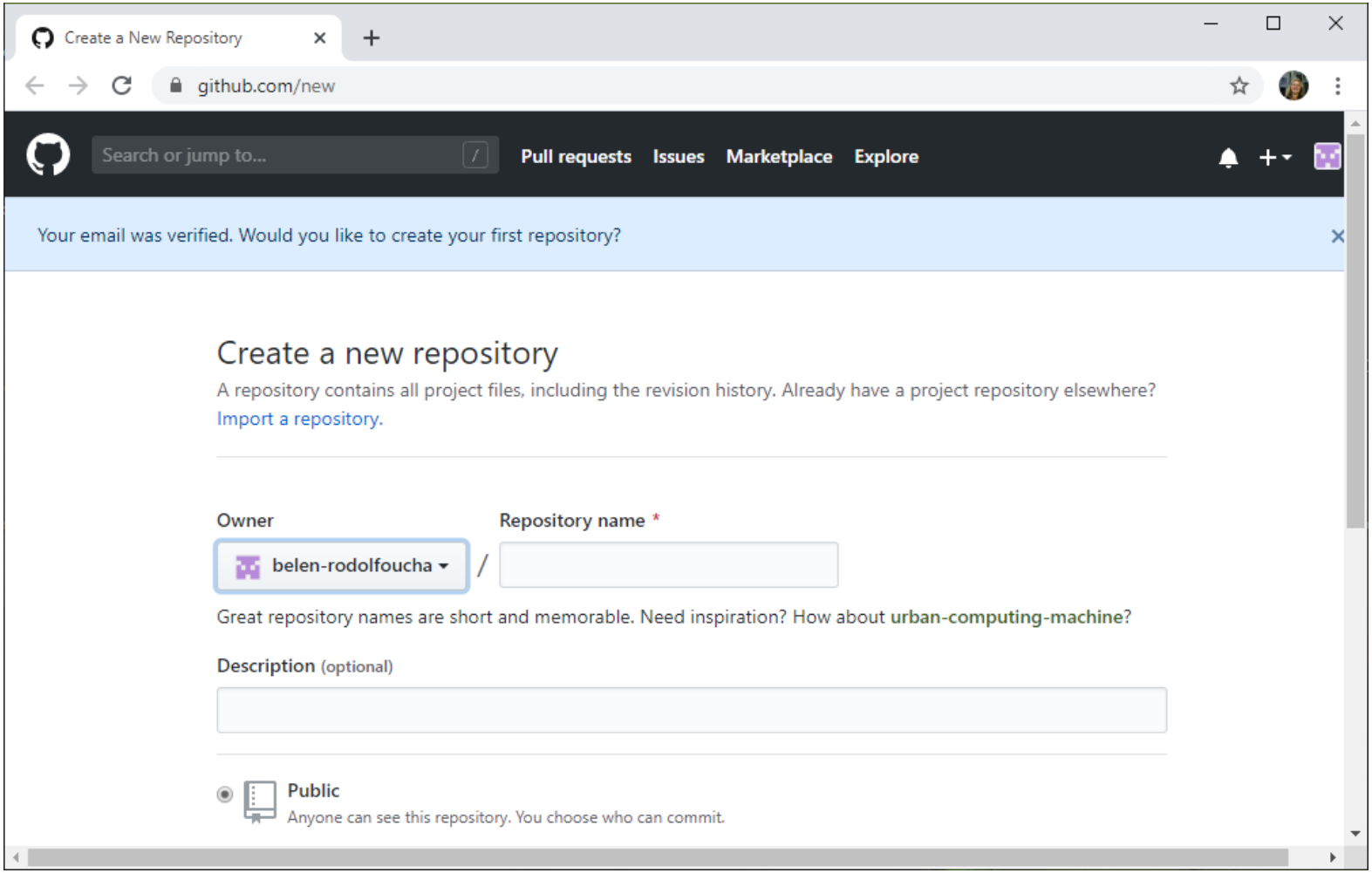
Creating an account on GitHub

- When you finish creating your account, you'll be asked to verify your e-mail address before you start using GitHub.



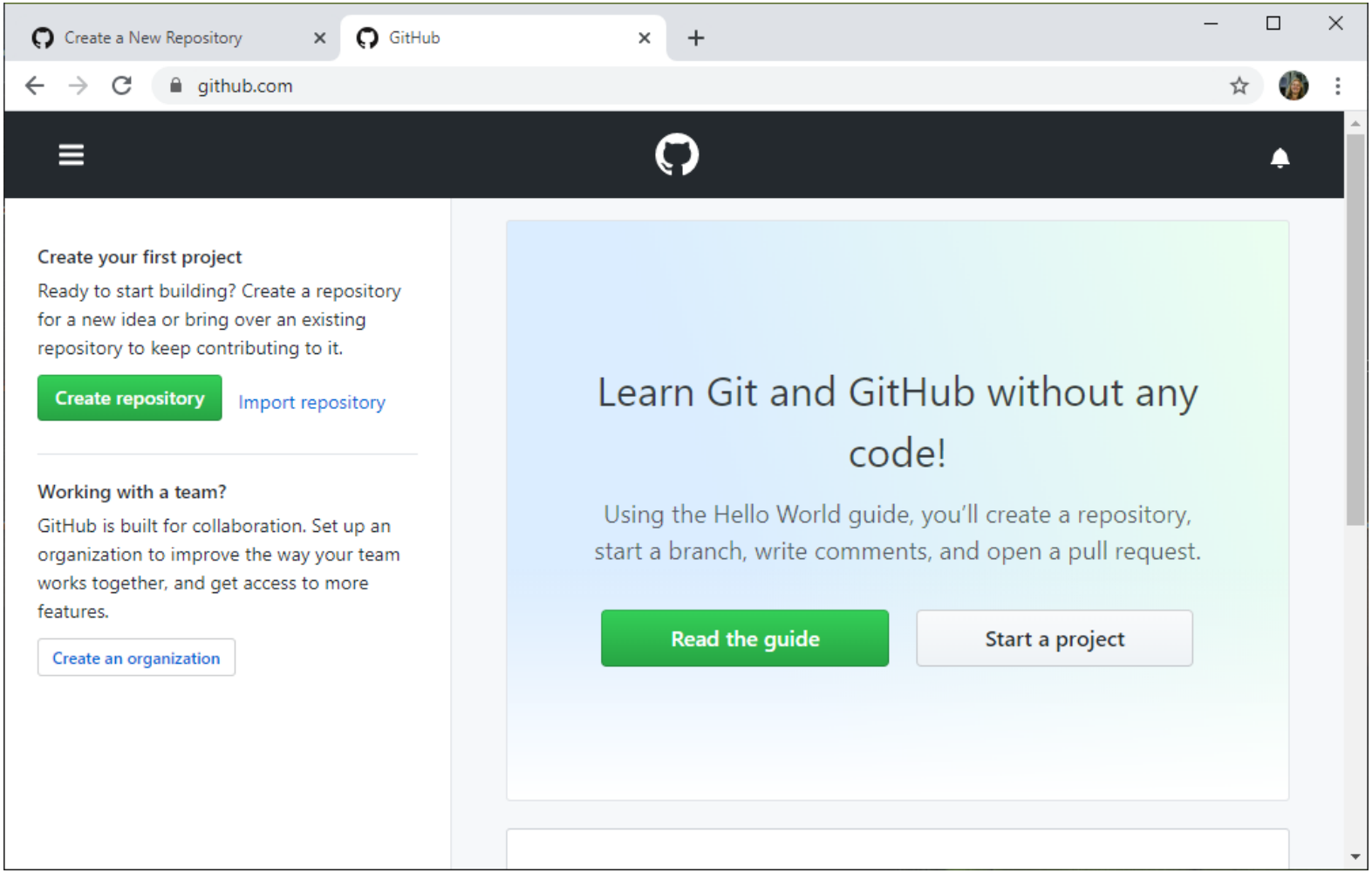
Creating an account on GitHub

- Once the GitHub address is verified, it automatically goes to the option of creating a new repository.



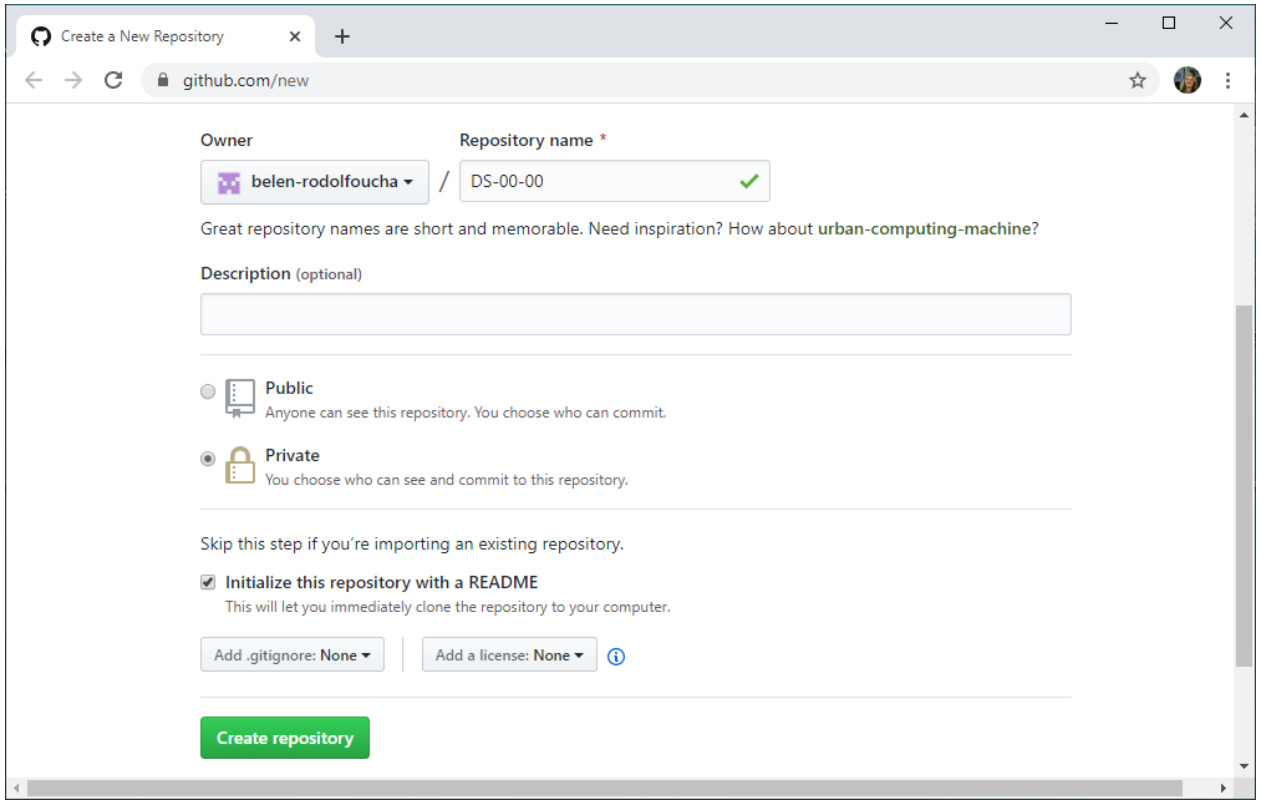
Creating an account on GitHub

- If you don't want to create it now you can always create it later by going to the option *Create repository*.



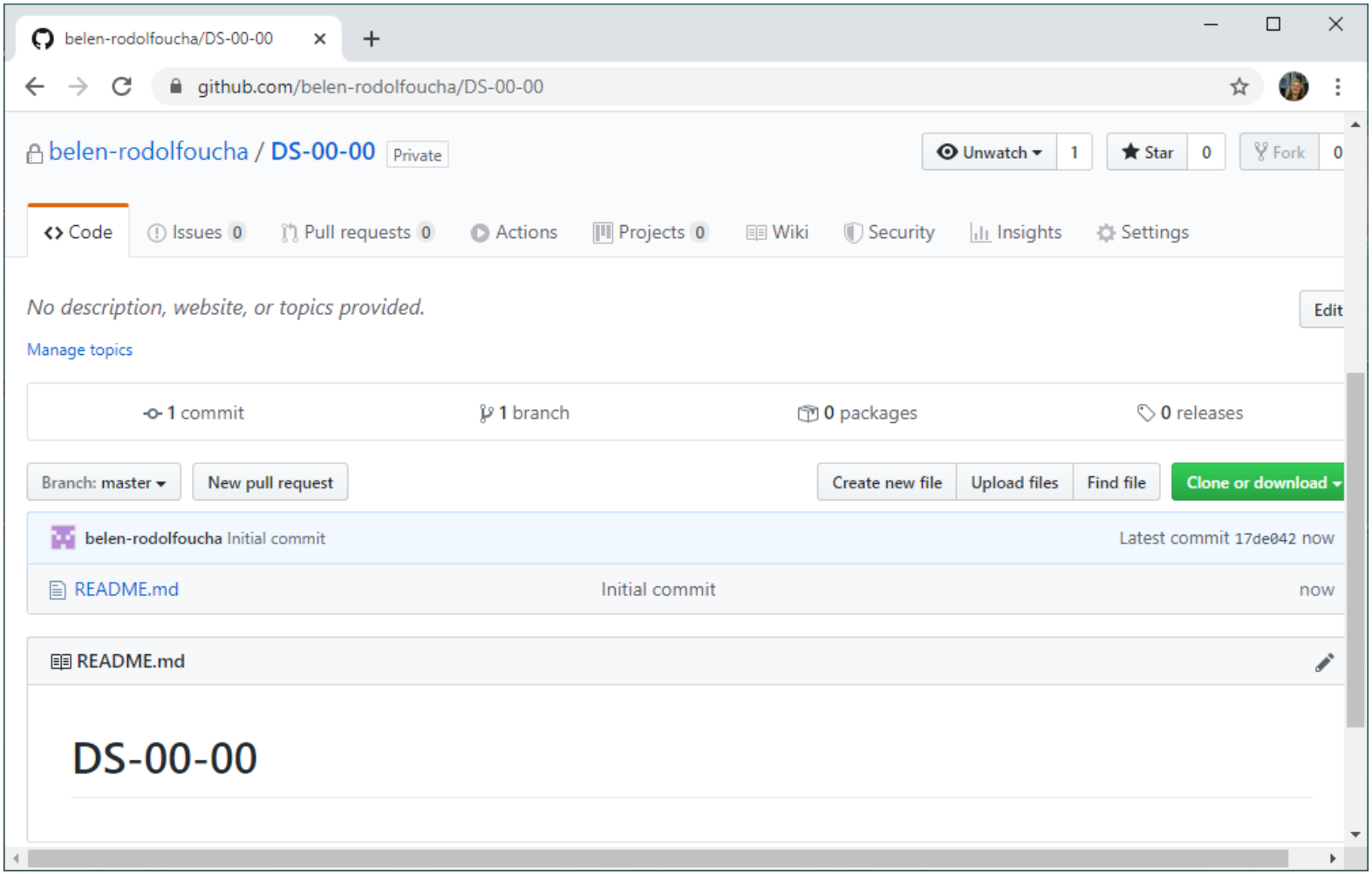
Creating an account on GitHub

- We are now going to create a repository with the name of your practice group (as you have registered it in the Moodle wiki), for example: `DS-00-00`, which should be **private**.
- We will indicate that it initializes the directory with a `README`.



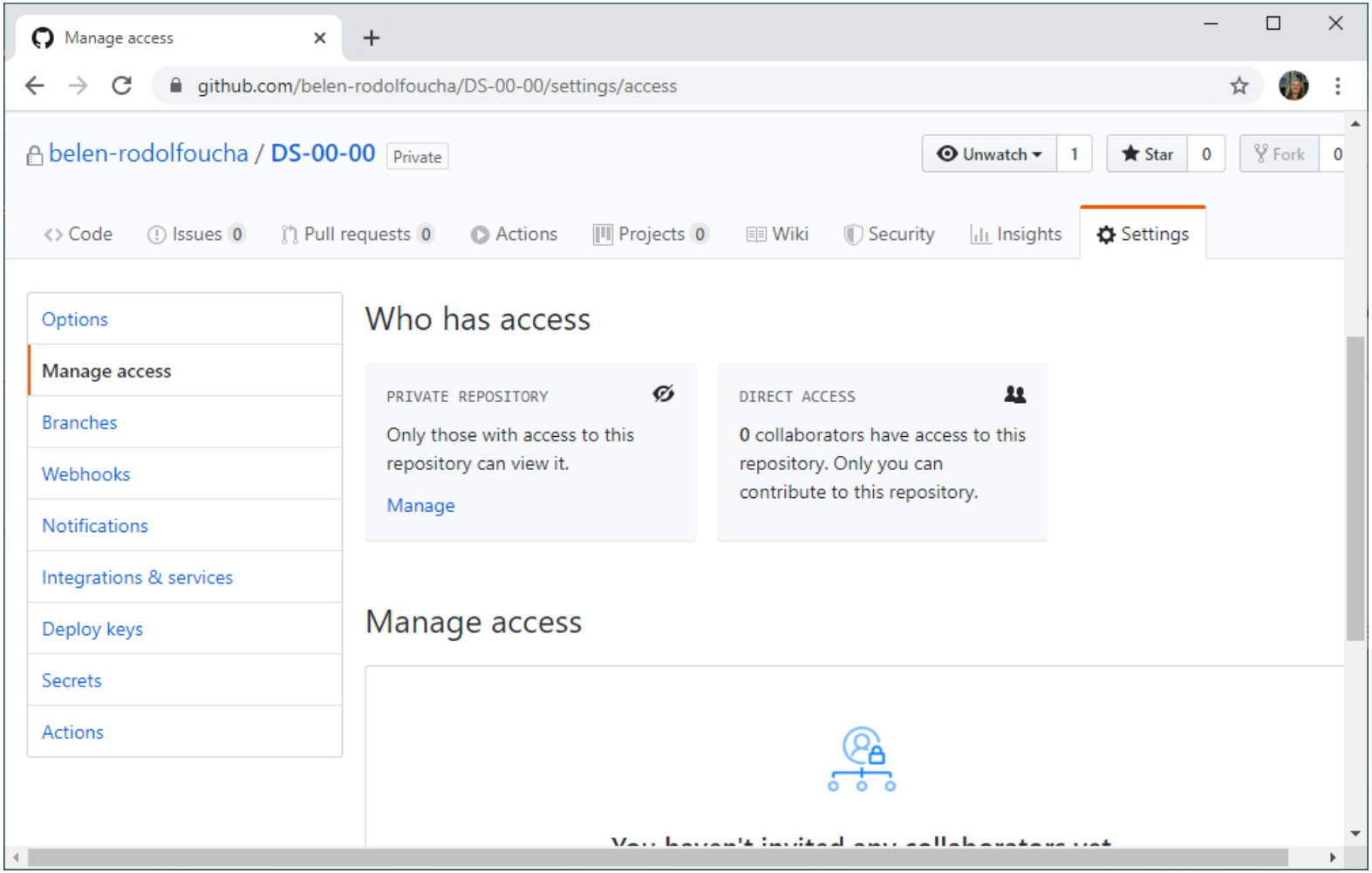
Creating an account on GitHub

- We already have the repository in GitHub created, a `commit` has been made that has included the `README.md` file in it.



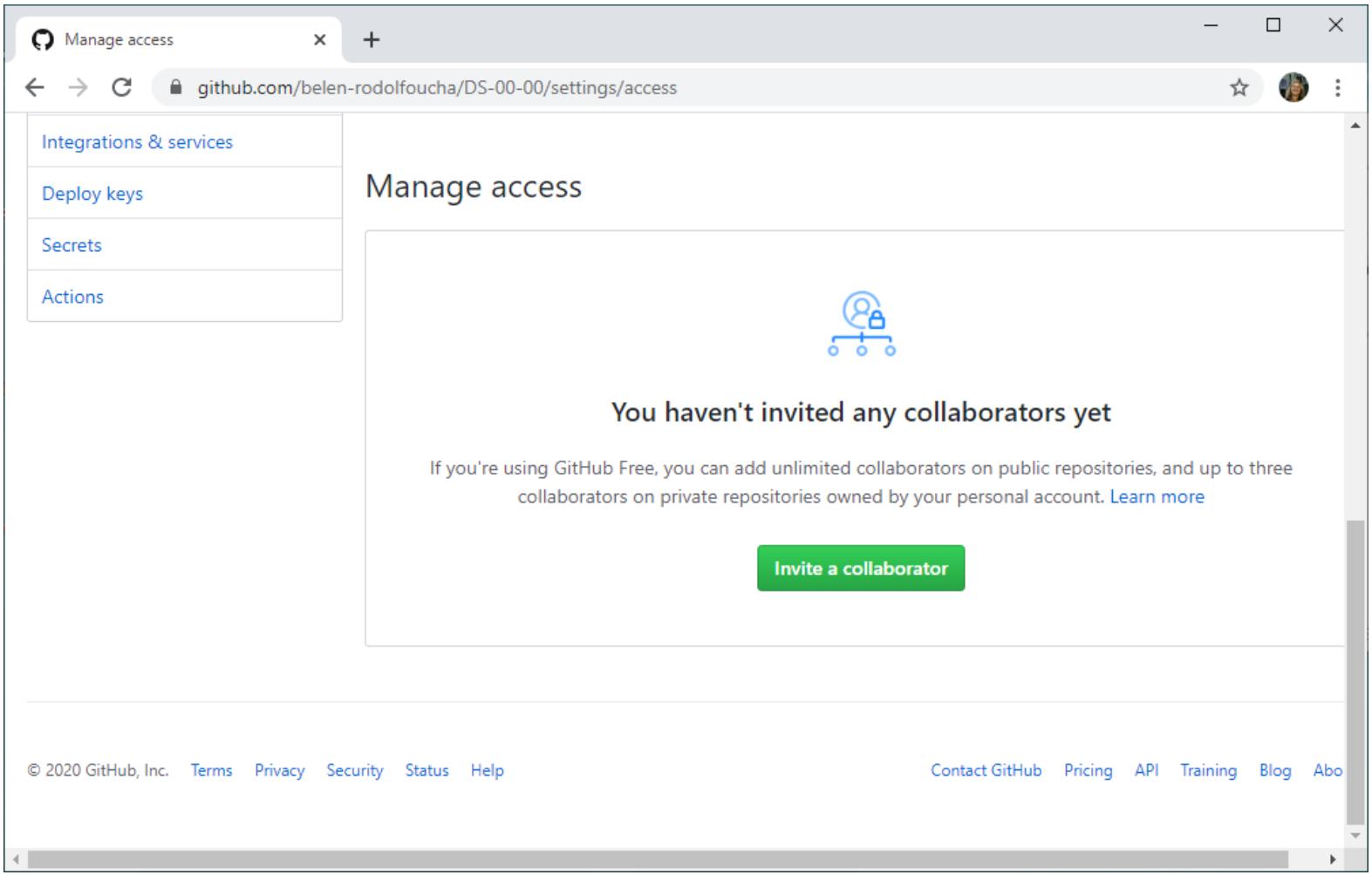
Adding your partner to your Git repository

- The next step is to give your partner access to the repository. You will need to go to *Settings* → *Manage access*...



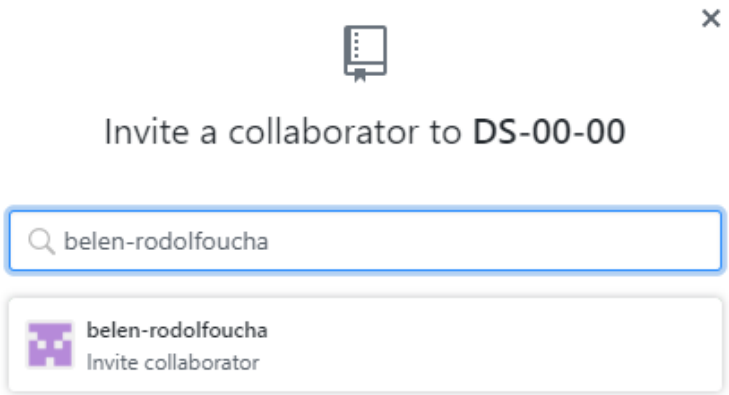
Adding your partner to your Git repository

- ...and make *scroll* and click on *Invite a collaborator*.



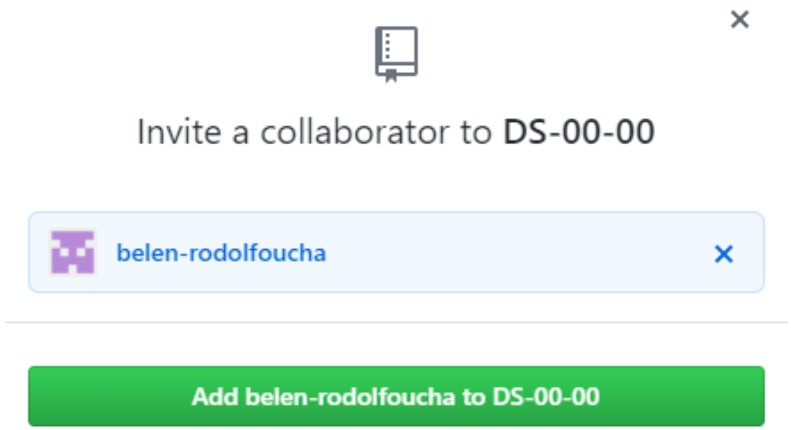
Adding your partner to your Git repository

- You look for your partner by its username...



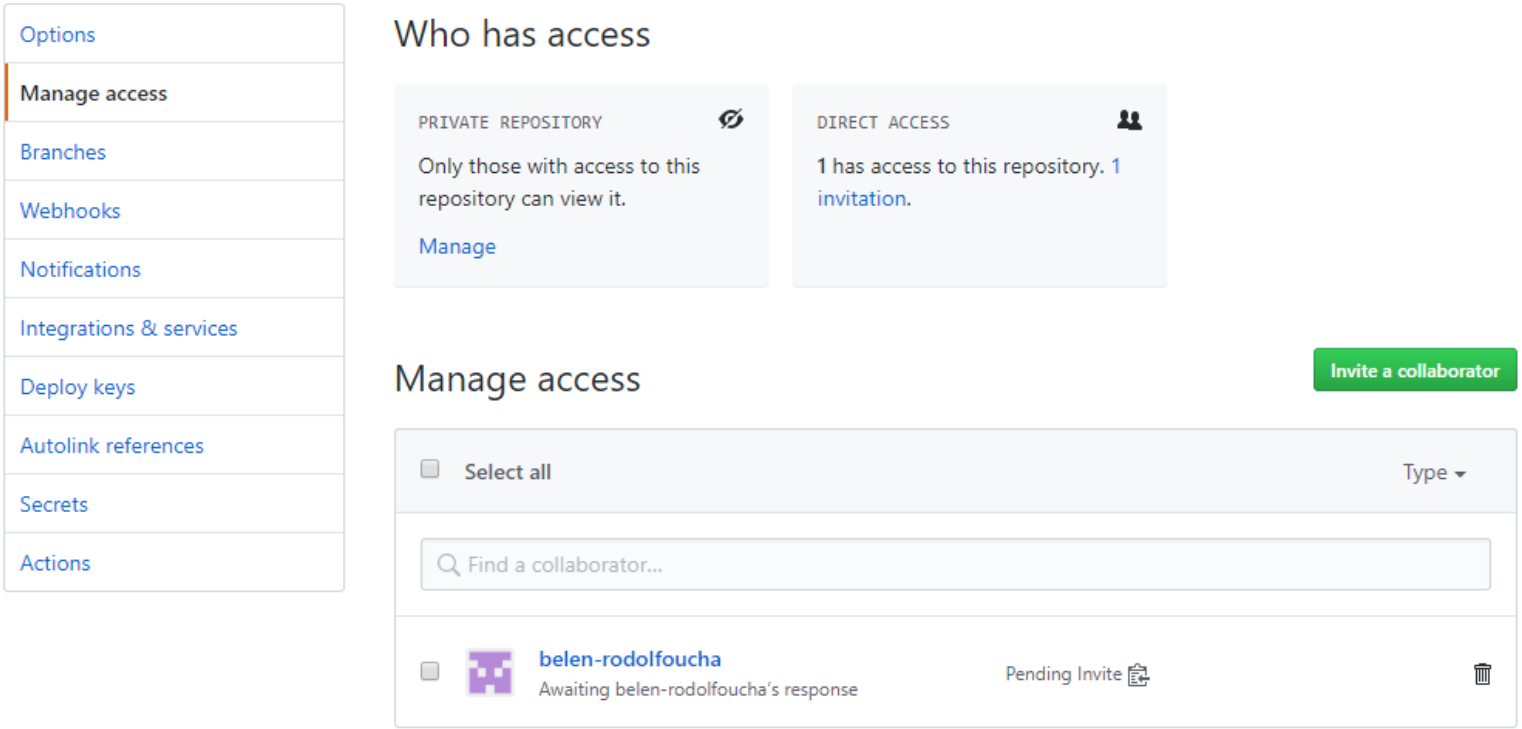
Adding your partner to your Git repository

- ...and you add it as a repository collaborator.



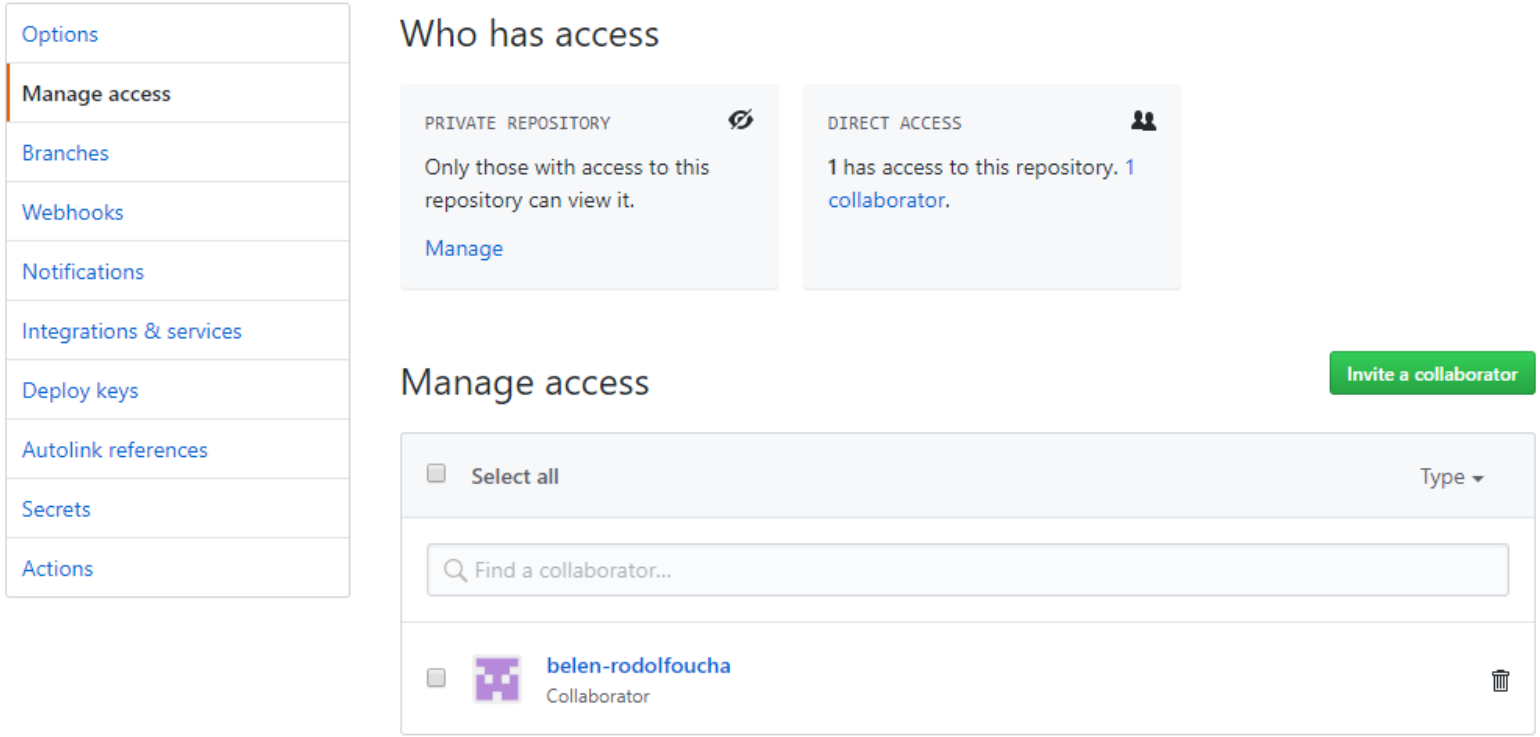
Adding your partner to your Git repository

- At first, the repository waits (*Pending Invite*) for the response from your partner (who will receive an email asking if he or she accepts or not).



Adding your partner to your Git repository

- As soon as the invitation is accepted you will see that his or her status is updated to *Collaborator*.

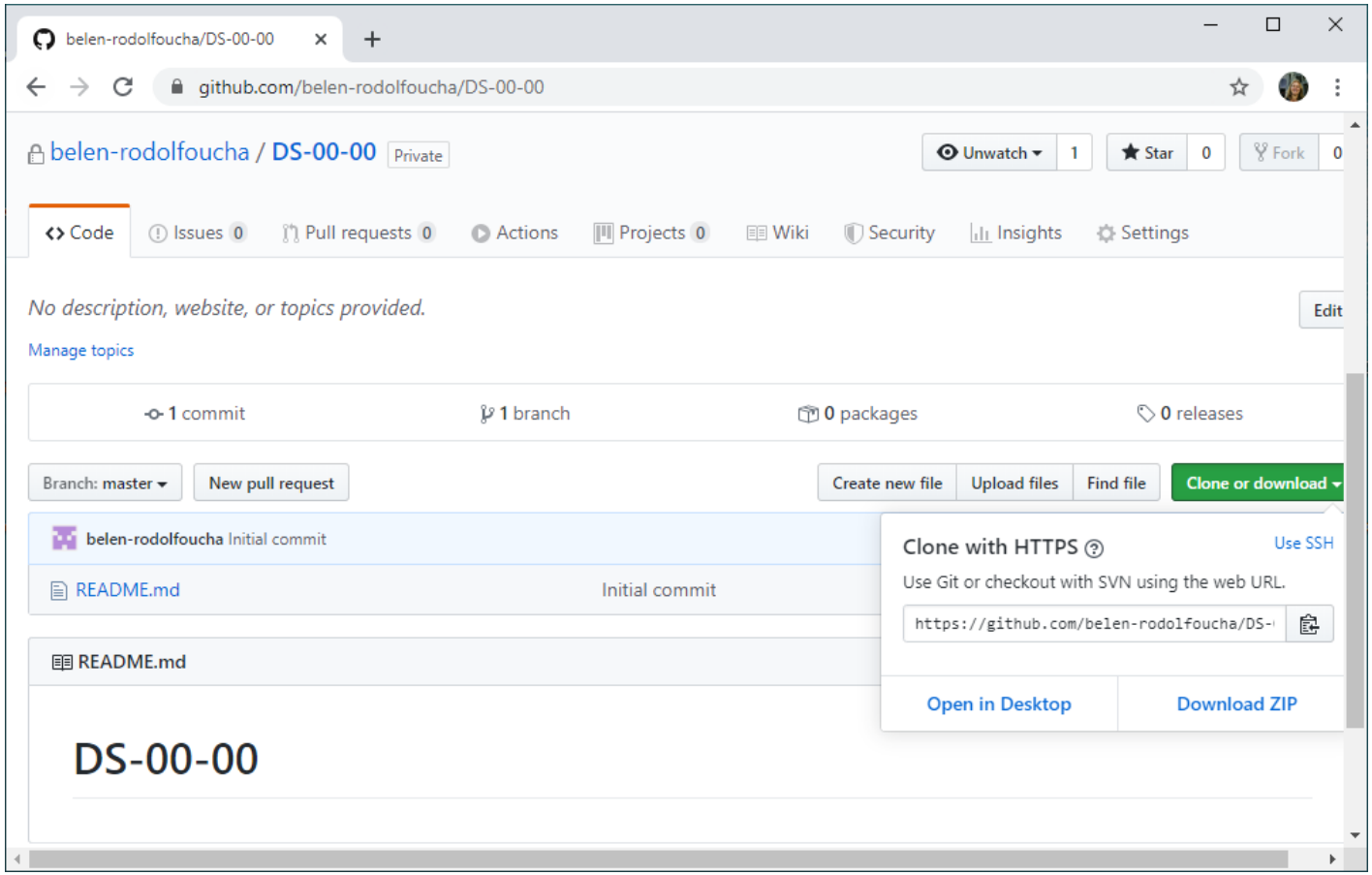


Adding teachers to your Git repository

- You will use the same procedure as to add a partner.
- **Importante**:
  - **It is your responsibility to create the repository and to share it with your practice teacher**.
  - In Moodle you have the list of teachers for each group and their corresponding users in GitHub.
  - After the deadline, teachers will clone, as soon as possible, the repositories that you have shared with us. If this has not been done, it will be understood that the practice has not been submitted.

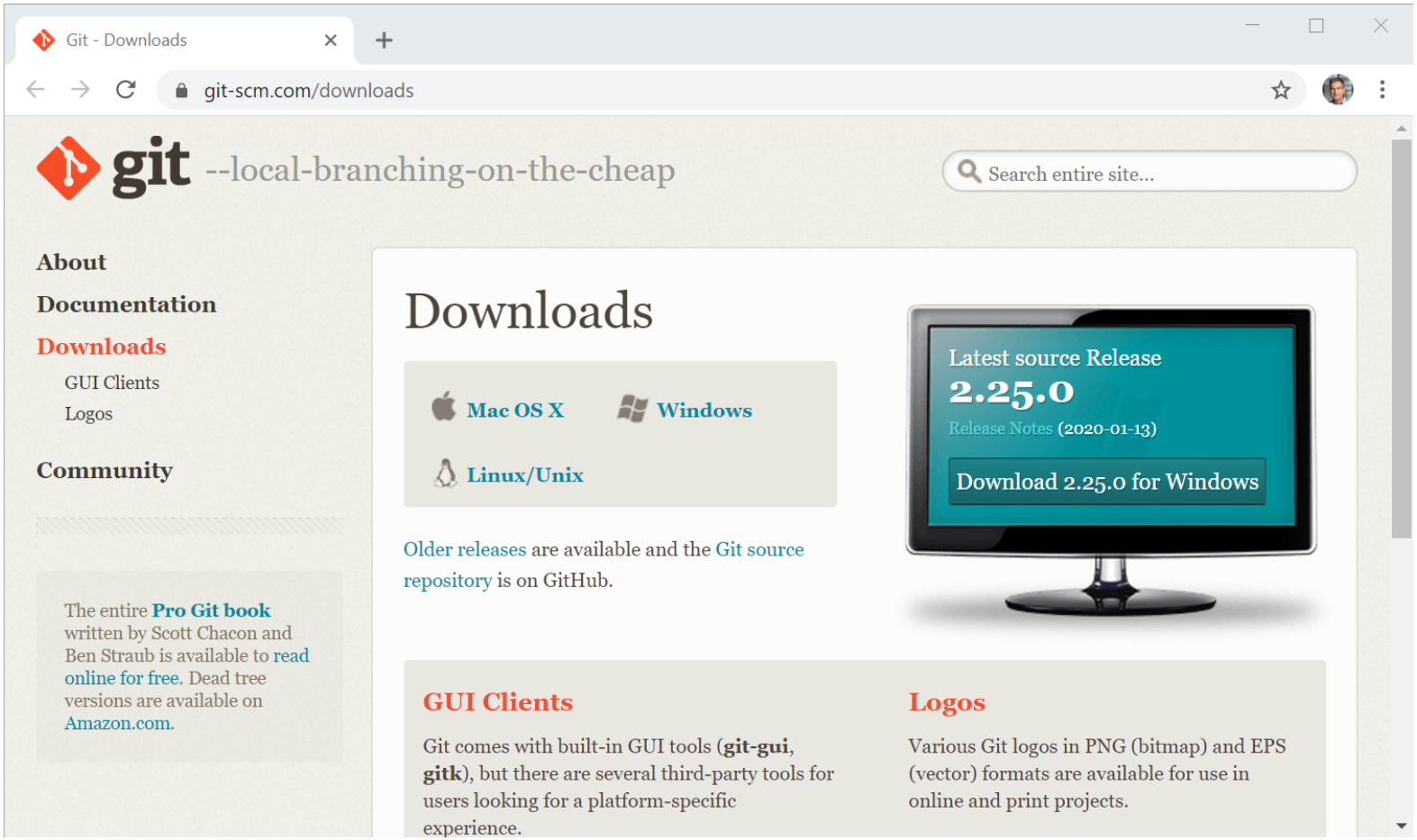
Git Repository on GitHub

- The last step in GitHub will be to keep the address of the remote repository that we will need to clone it locally. We can get it by clicking on the clone button.



Installing Git

- The next step is to install Git in your computer (if it's not already) from `https://git-scm.com/downloads`.



Installing Git

- Install Git with the default options. Make sure that allowing *3rd-party software* is enabled (it shoudl be by default).

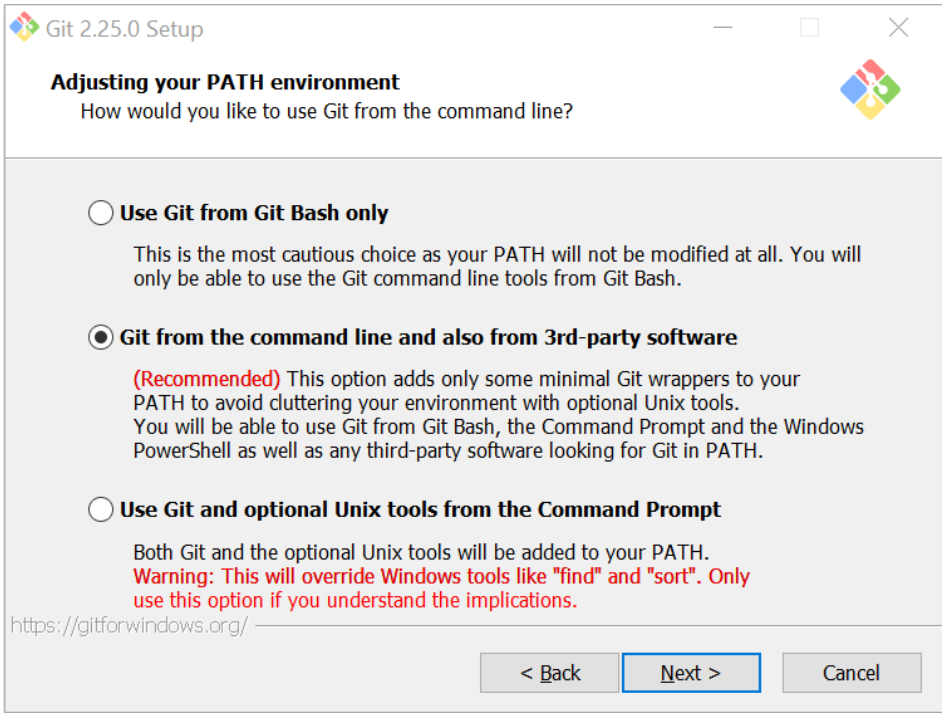


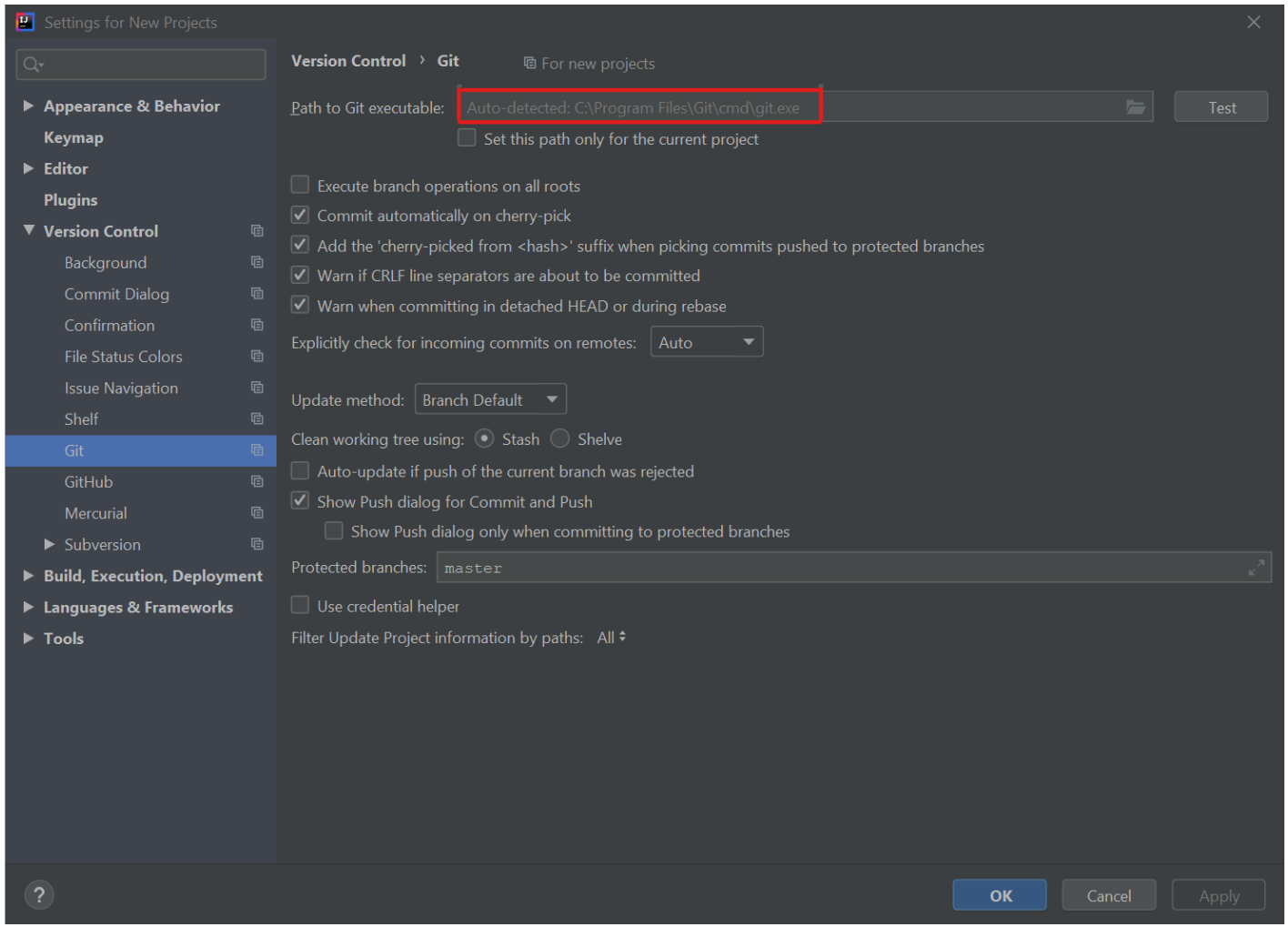
Table Of Contents

FIC's Git Repository with IntelliJ IDEA

- IntelliJ IDEA will auto-detect the Git installation, but we can check its correctness opening the configuration...

FIC's Git Repository with IntelliJ IDEA

- ...and looking at *Version Control - Git*.

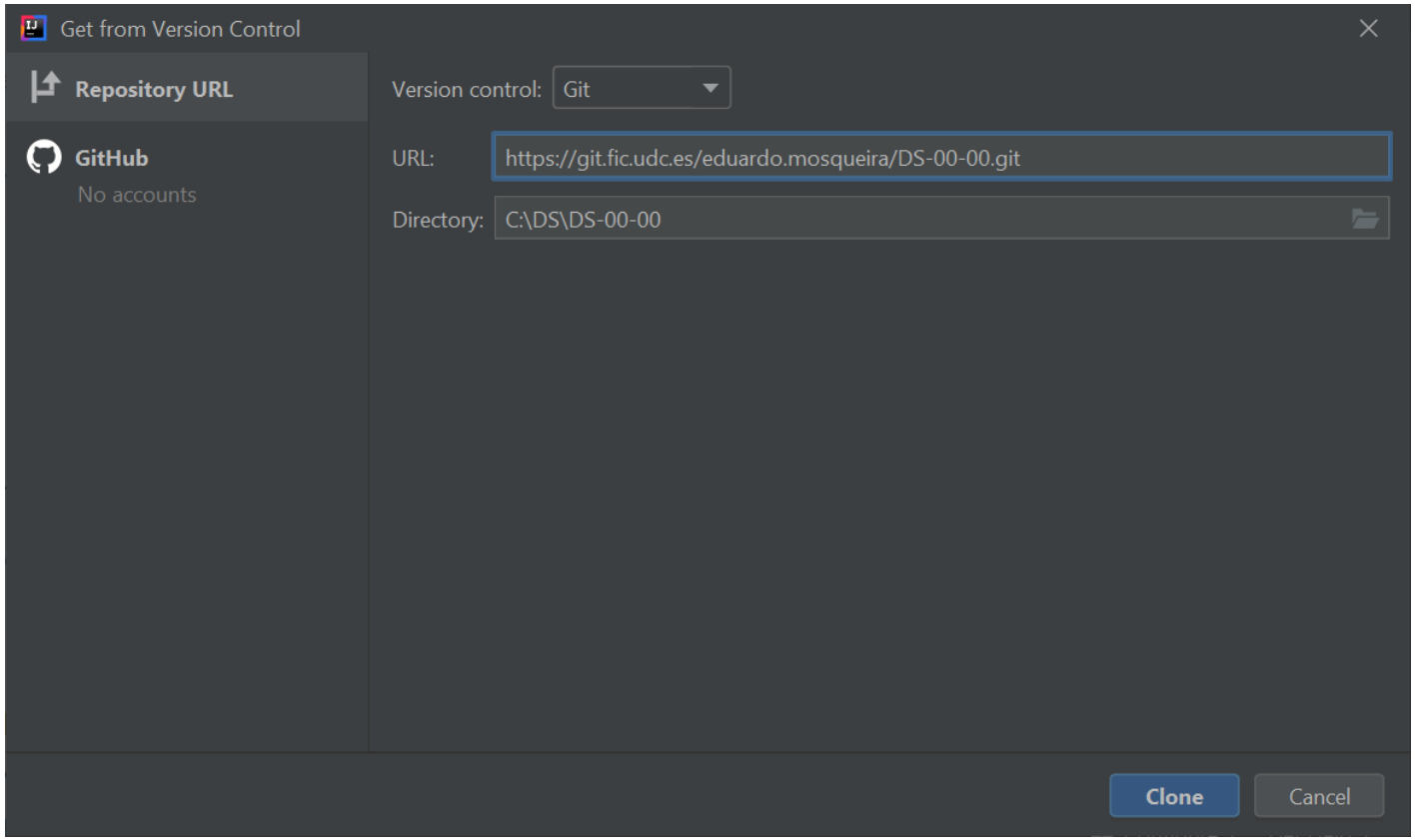


FIC's Git Repository with IntelliJ IDEA

- Once Git is installed, we must copy (clone) our GitLab repository (empty, so far) into a local computer.
- To do that, we select *Get from Version Control* on IntelliJ IDEA.

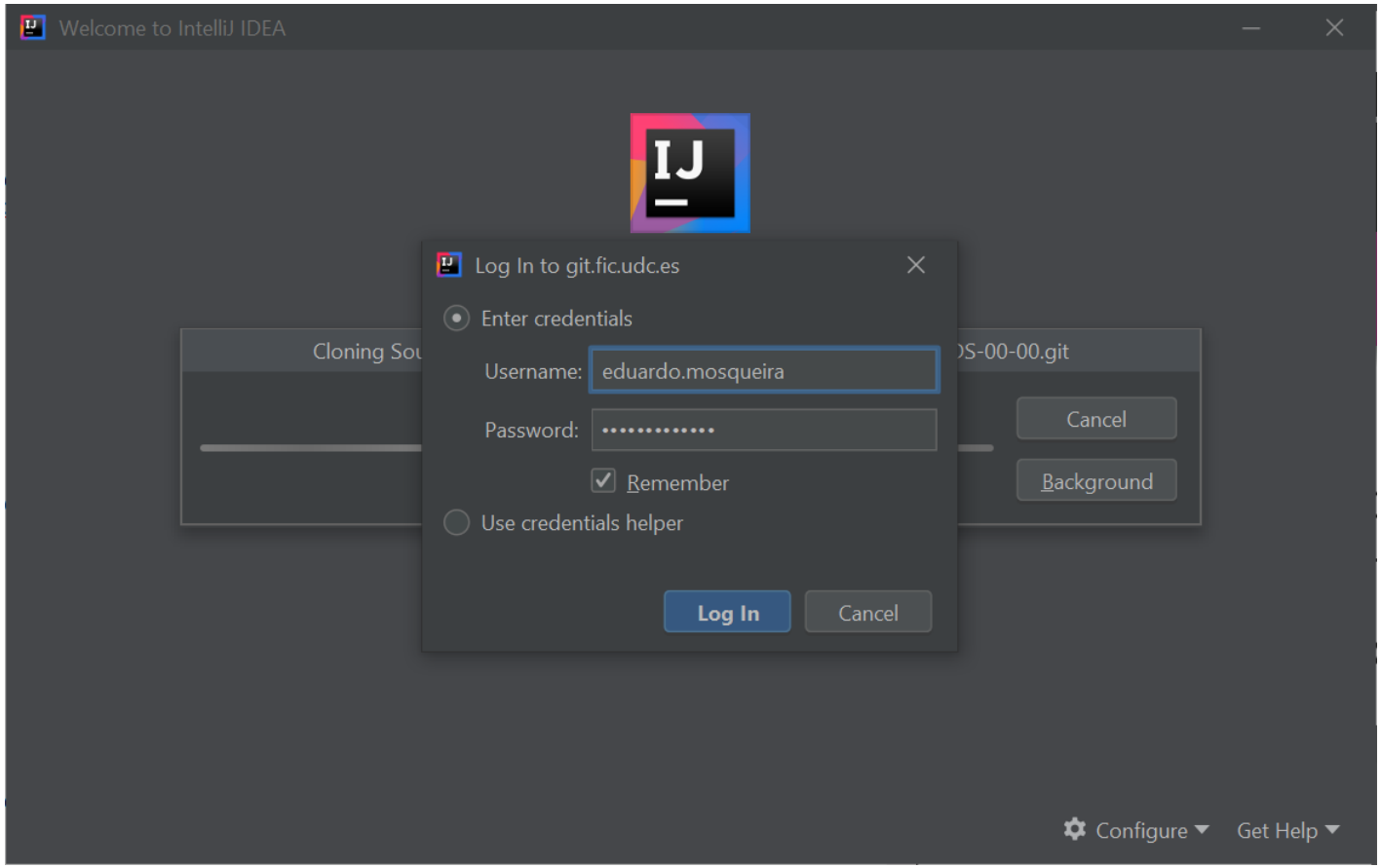
FIC's Git Repository with IntelliJ IDEA

- We paste that URL on IDEA, chosse the local folder for the repository, and click *Clone*.



FIC's Git Repository with IntelliJ IDEA

- It will ask for our *Username* and *Password* (simply your UDC credentials) and we tell it to save them.

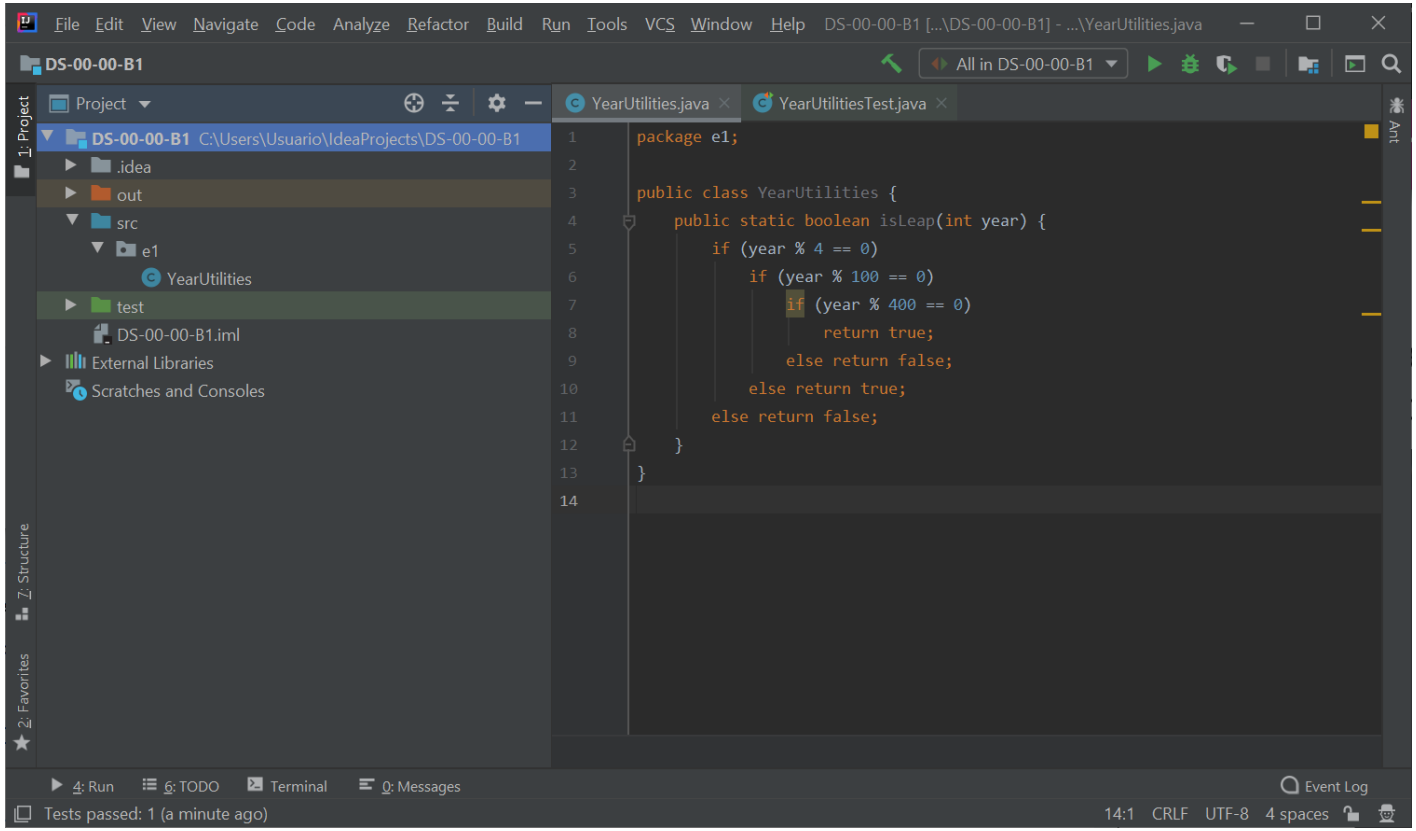


FIC's Git Repository with IntelliJ IDEA

- It will ask whether we want to create an IntelliJ IDEA project in the recently created local copy.
- If the project is already created in other location, it is more convenient to copy it from the browser. Therefore, we choose no.

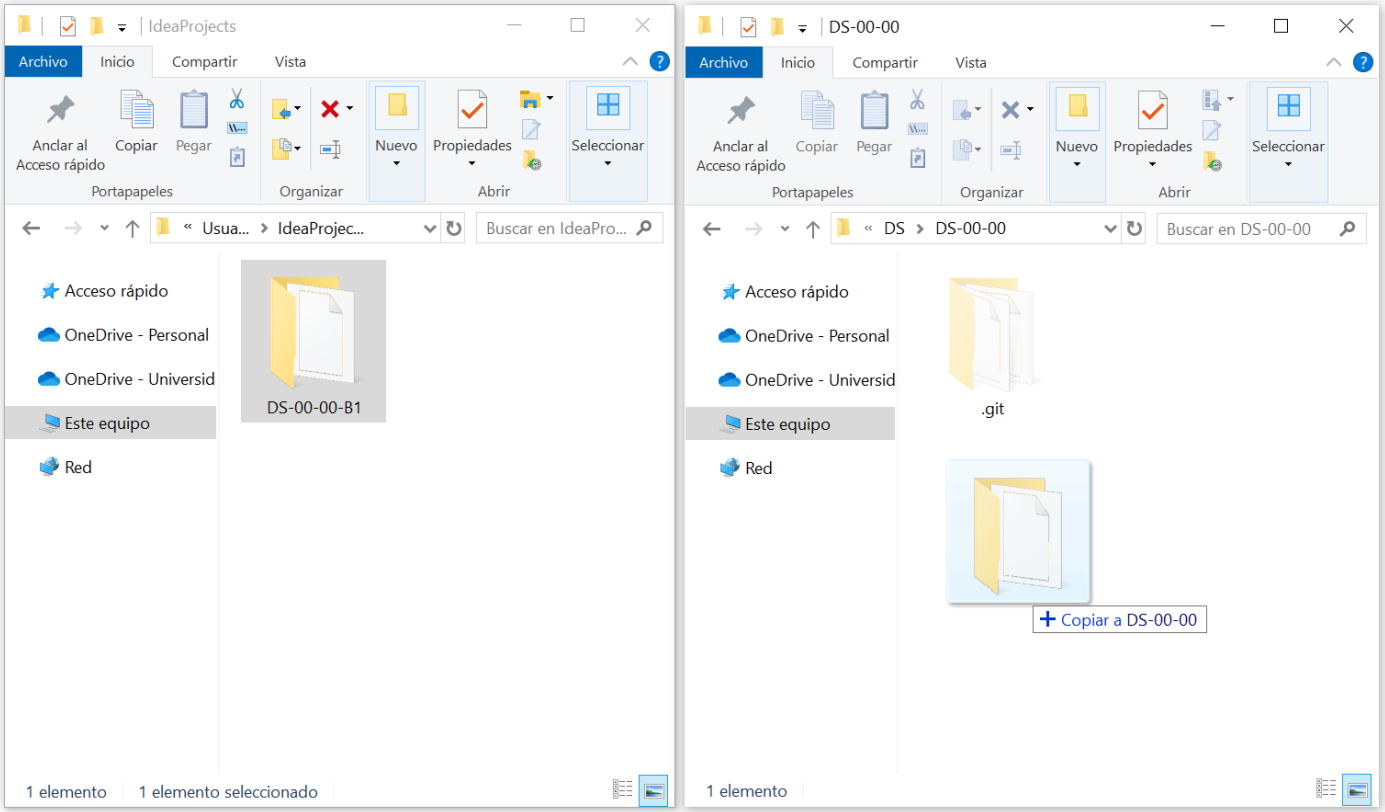
Uploading the project into the Repository with IntelliJ IDEA

- We now go to the created project (e.g., the one for the leap years renamed as `DS-00-00-B1`)...



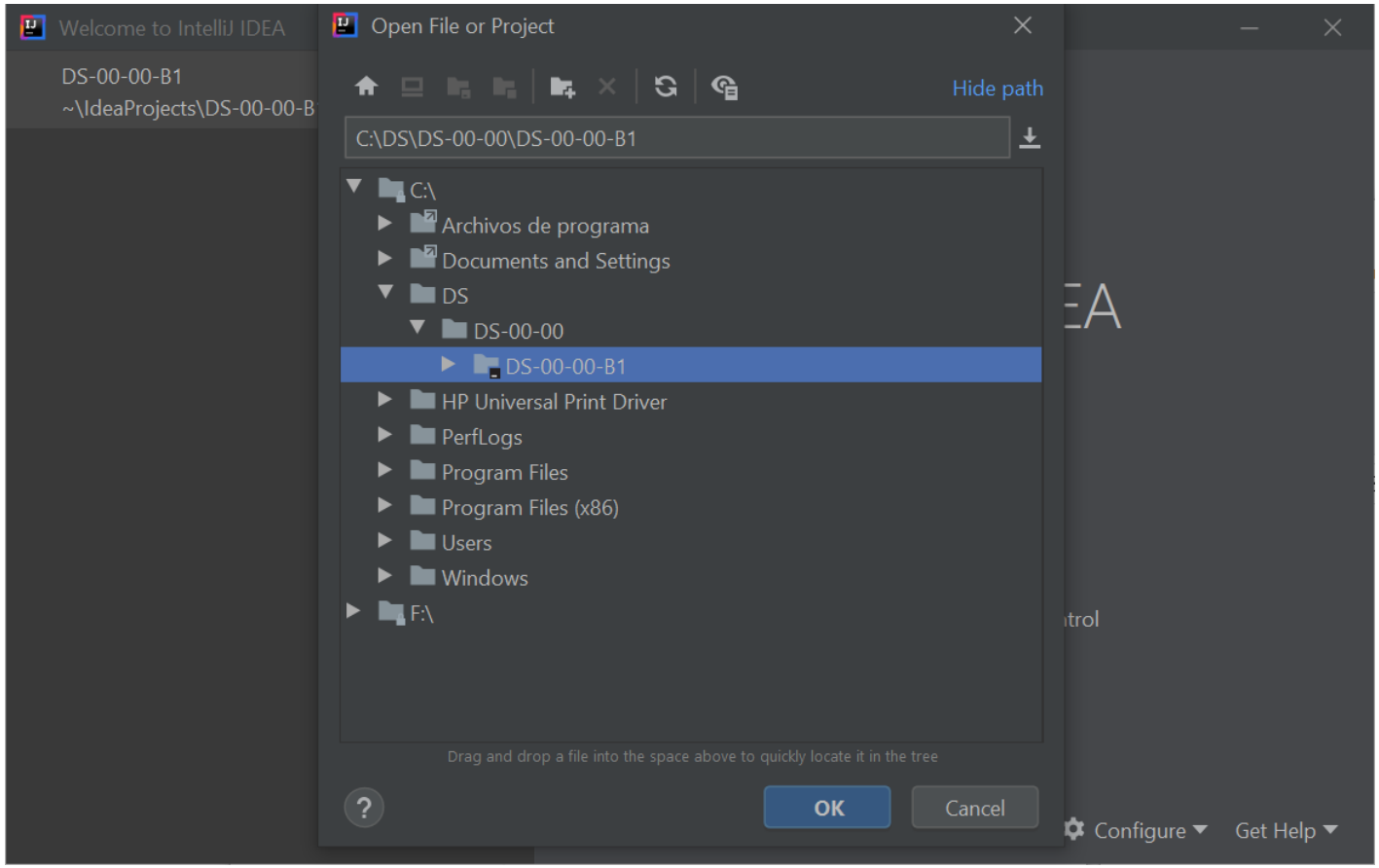
Uploading the project into the Repository with IntelliJ IDEA

- And simply, from the browser, we copy the whole folder into the previously cloned repository.
- We can also see that the target folder has a hidden `.git` folder that should not be touched.



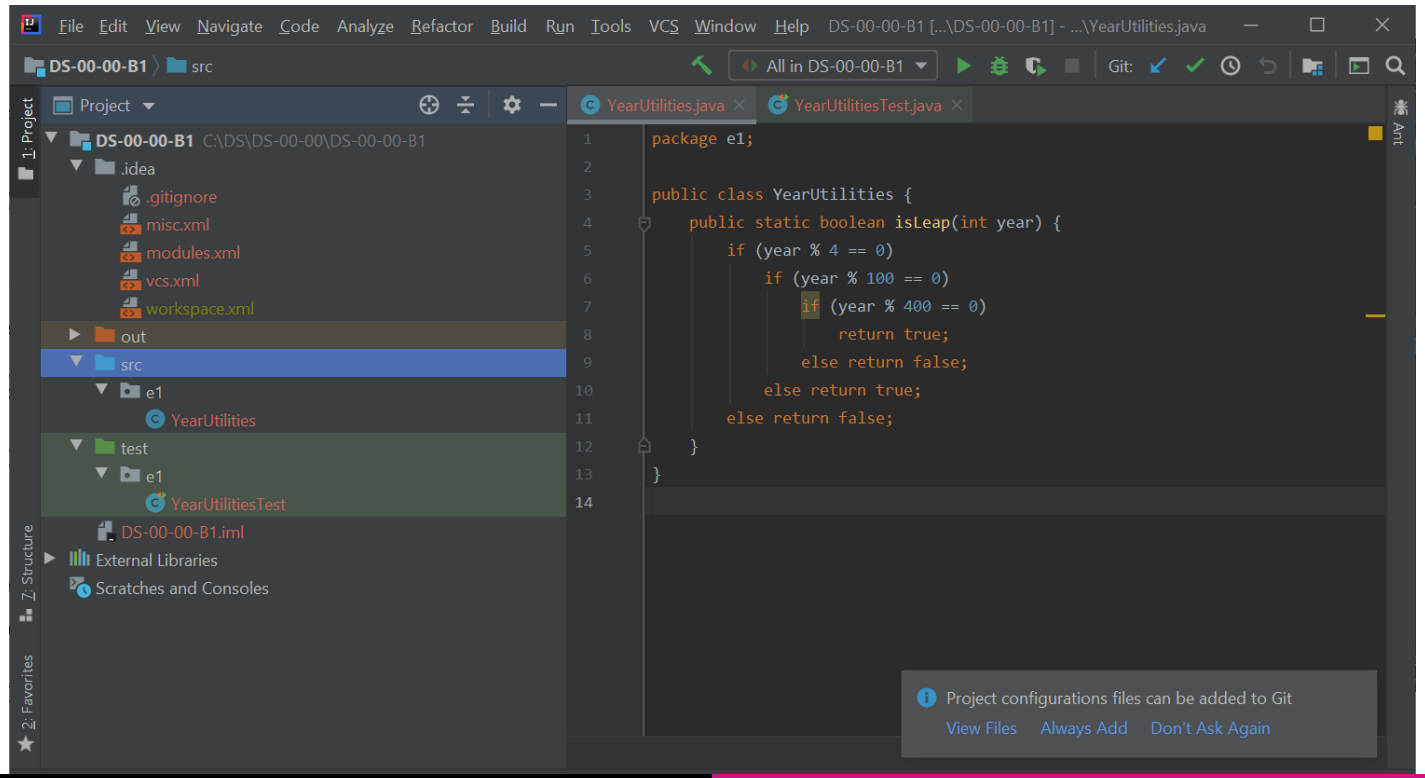
Uploading the project into the Repository with IntelliJ IDEA

- The next step is opening the new project, which is in the Git-controlled location.



Uploading the project into the Repository with IntelliJ IDEA

- The files marked in light red are in the local copy but are not marked for uploading to the repository.
- The file in yellow (`workspace.xml`) will be ignored by Git.
- It will also ask if we want to upload the project's configuration files into the repository.

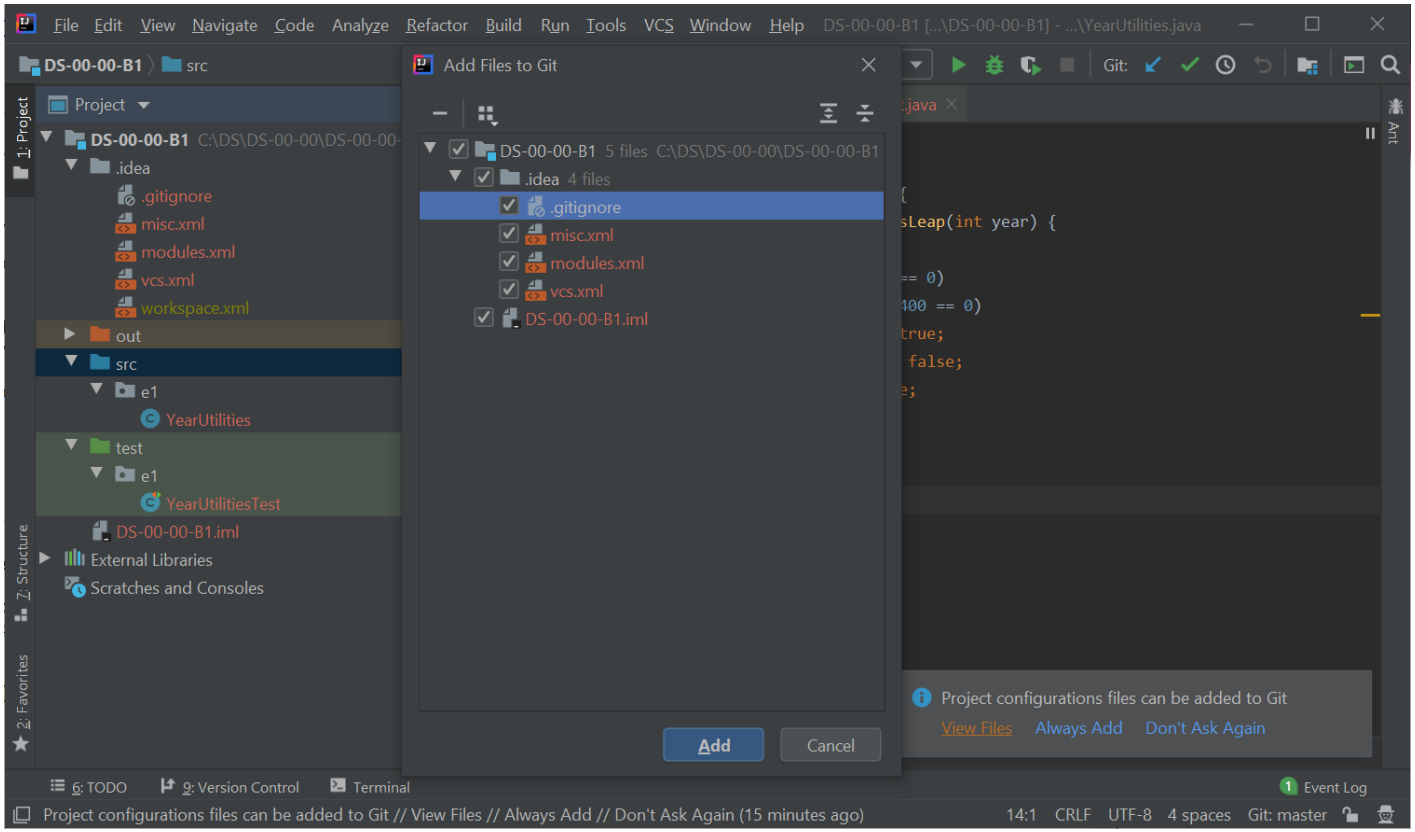


Uploading the project into the Repository with IntelliJ IDEA

- **Which files will we upload?**
  - We must upload all the sources (`src` and `test` folders).
  - We will also upload the project's configuration files (`.idea` folder) and the `.iml` file. This might be unnecessary if we used different IDEs or *build* systems like Maven or Gradle, but that is not the case in this course.
  - The `.idea/workspace.xml` file must not be uploaded as it stores the local configuration of our own IDE (open files, etc.). This has been ignored by adding a `.gitignore` to the folder, which simply contains the name of that configuration file.
  - The compiled files (`out` folder) are generated from the sources, so they must not be uploaded to the repository.

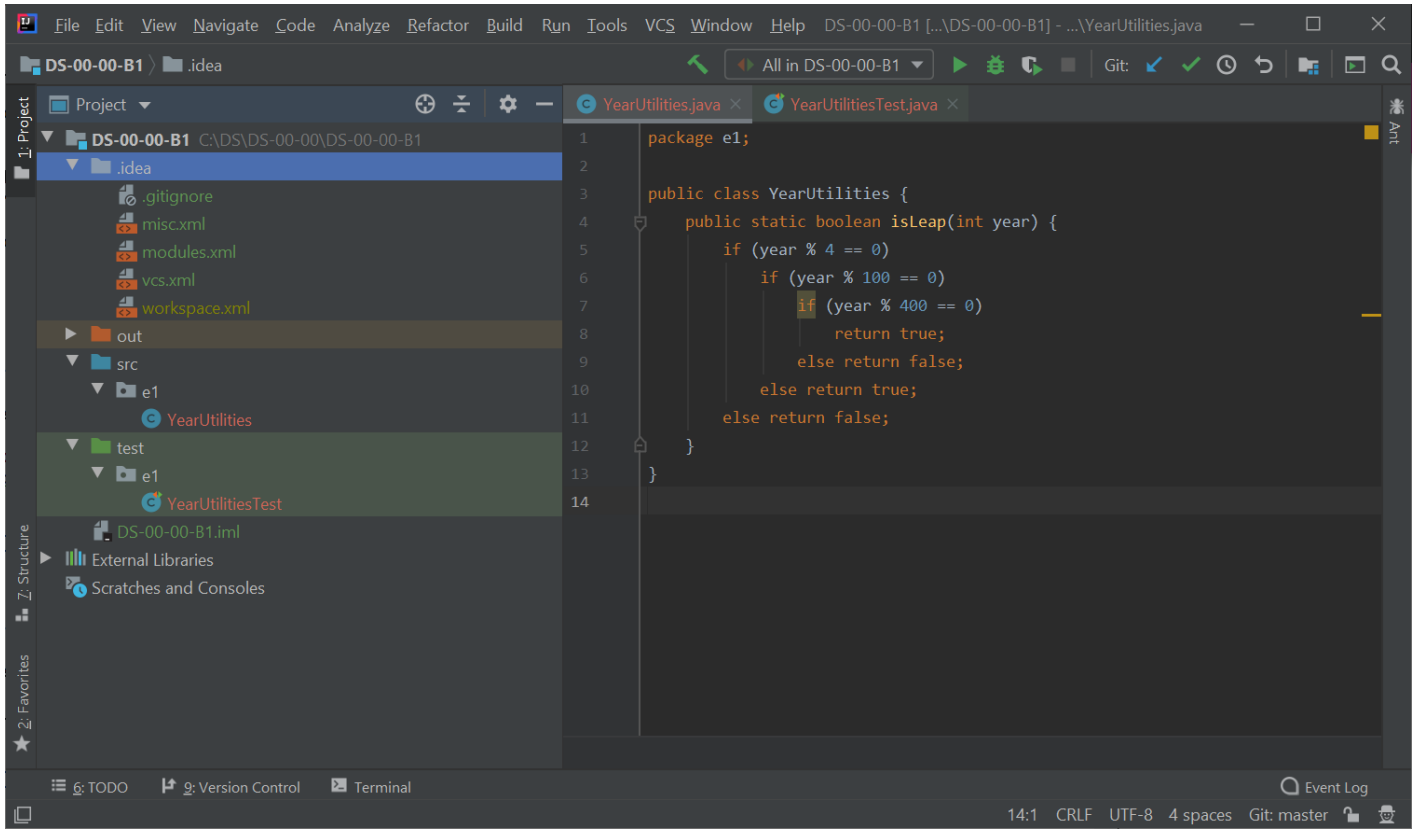
Uploading the project into the Repository with IntelliJ IDEA

- If we click on *View Files* on the configuration *pop-up*, we can state that we do want to upload the project's configuration. We click *Add*.



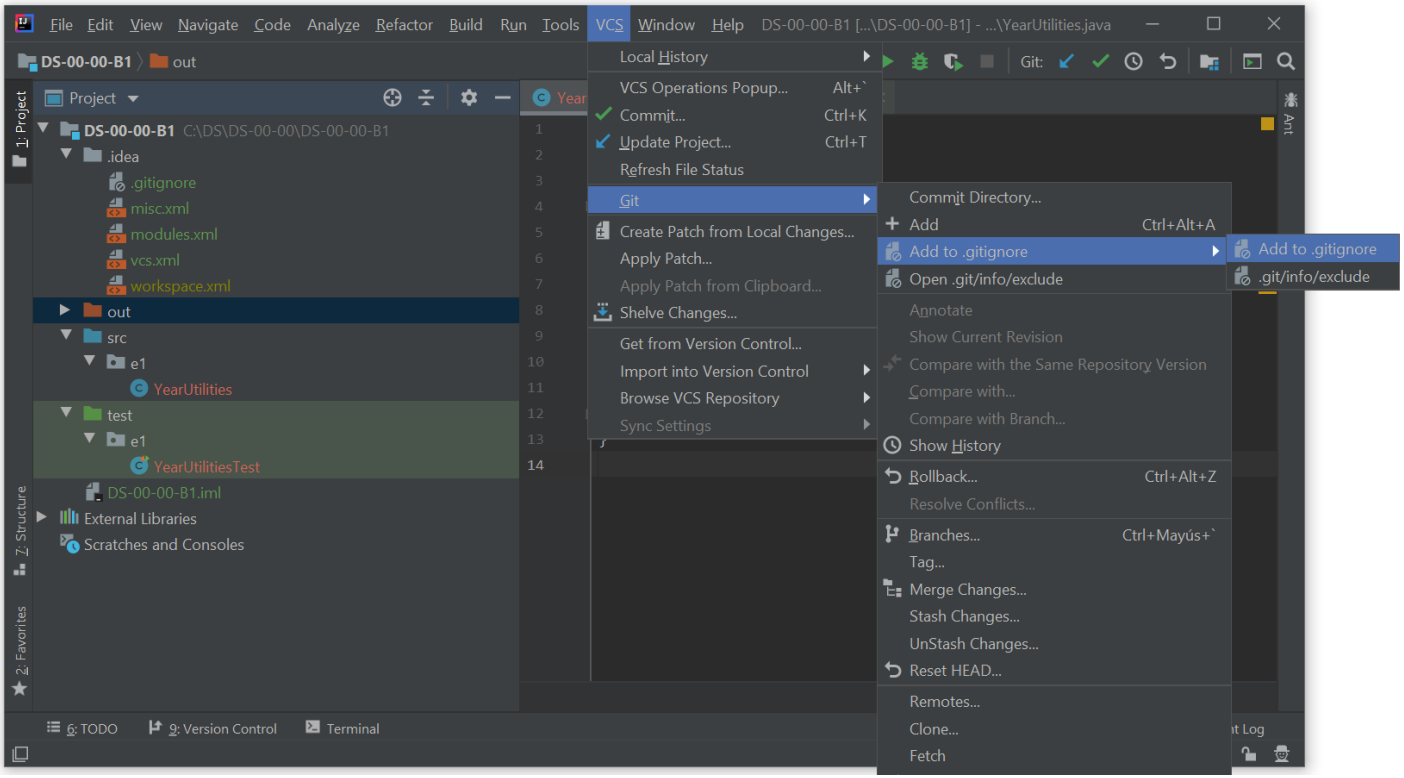
Uploading the project into the Repository with IntelliJ IDEA

- We now see that all the configuration files are marked in green, which means that they are ready for uploading.



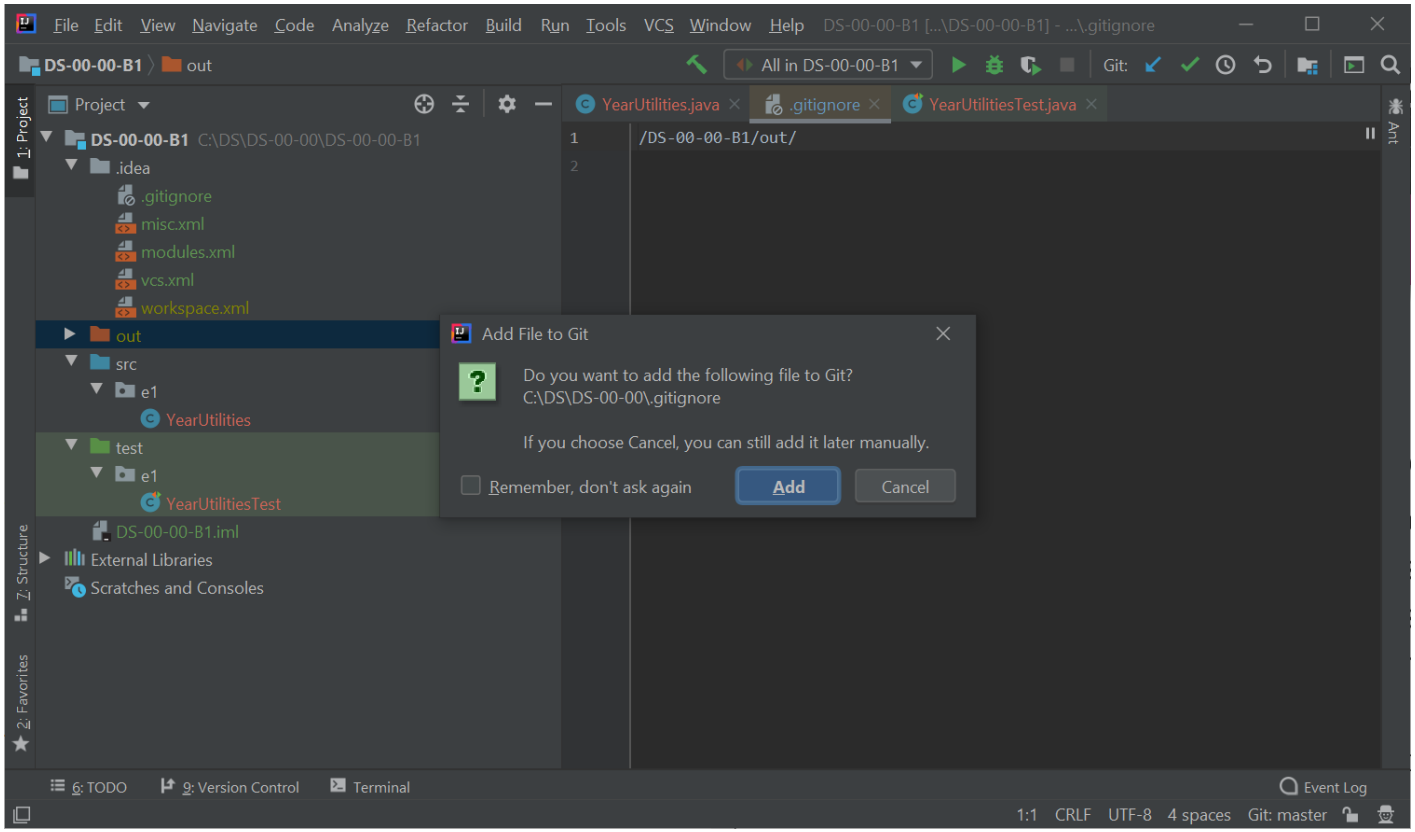
Uploading the project into the Repository with IntelliJ IDEA

- In order to avoid being asked again about the `out` folder, we click on it and from the *VCS - Git* menu we create a `.gitignore` file (it will ask for permission).



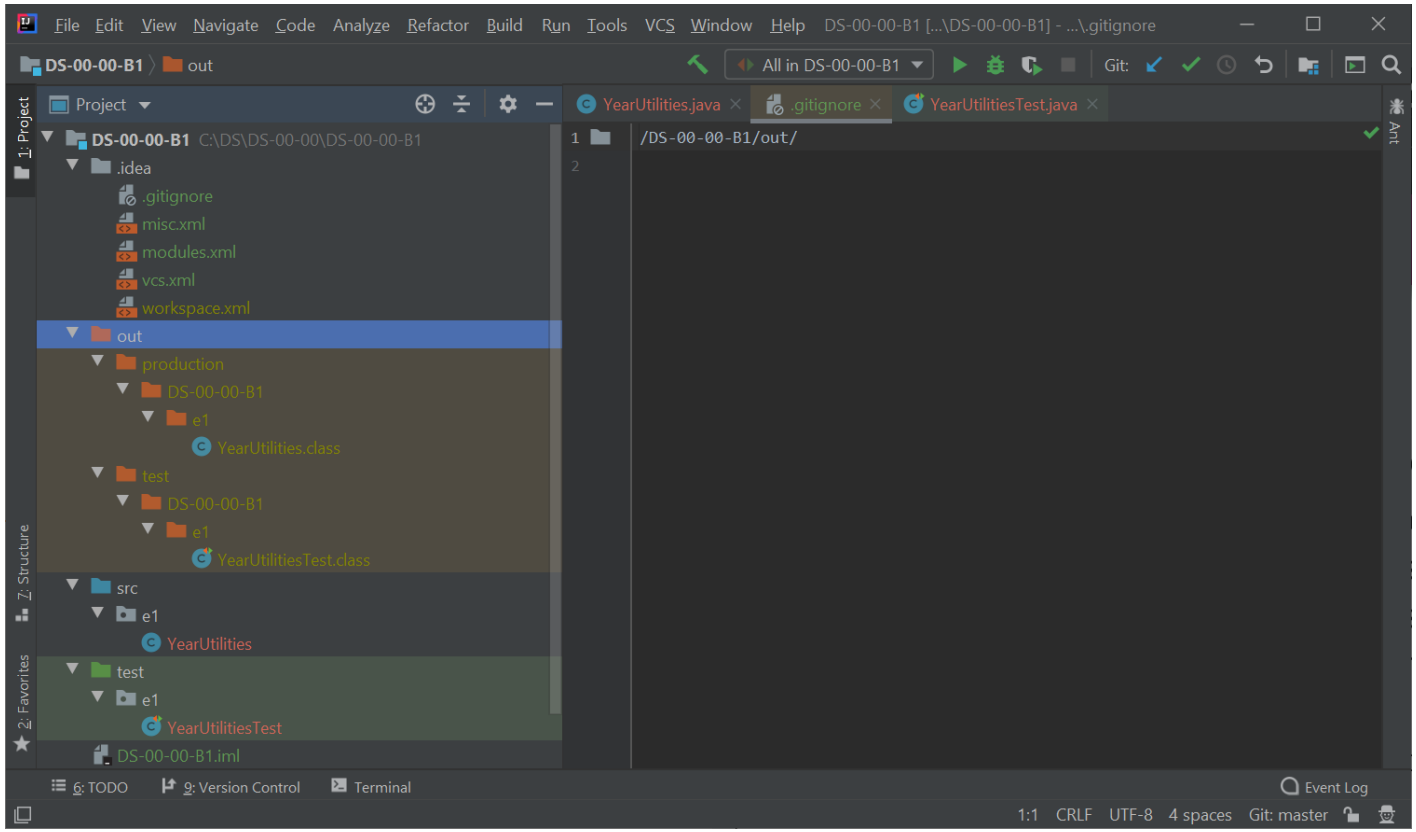
Uploading the project into the Repository with IntelliJ IDEA

- The created `.gitignore` file must be also uploaded to the repository.



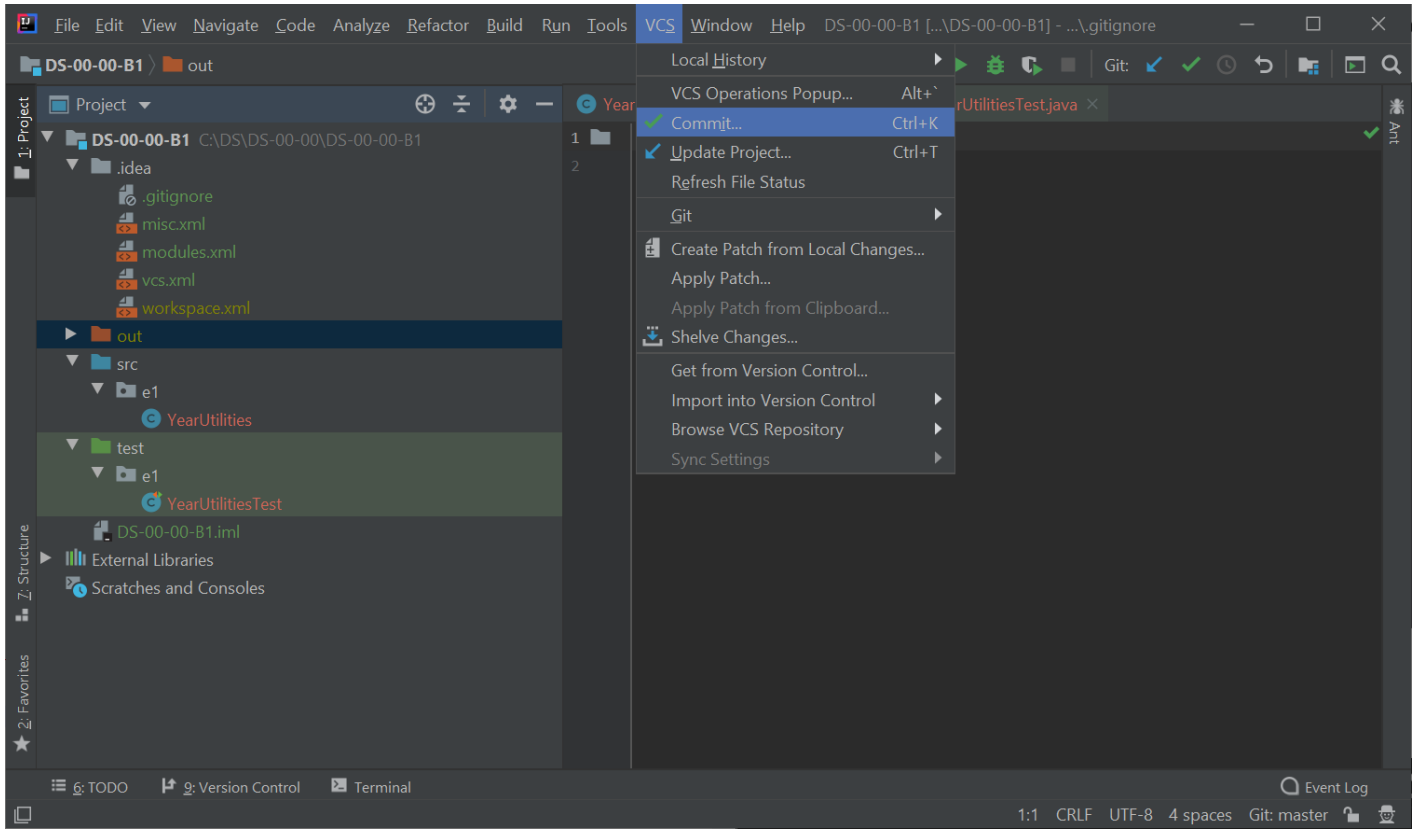
Uploading the project into the Repository with IntelliJ IDEA

- Now the `out` folder and all its contents are marked in yellow, so they will be ignored by Git.



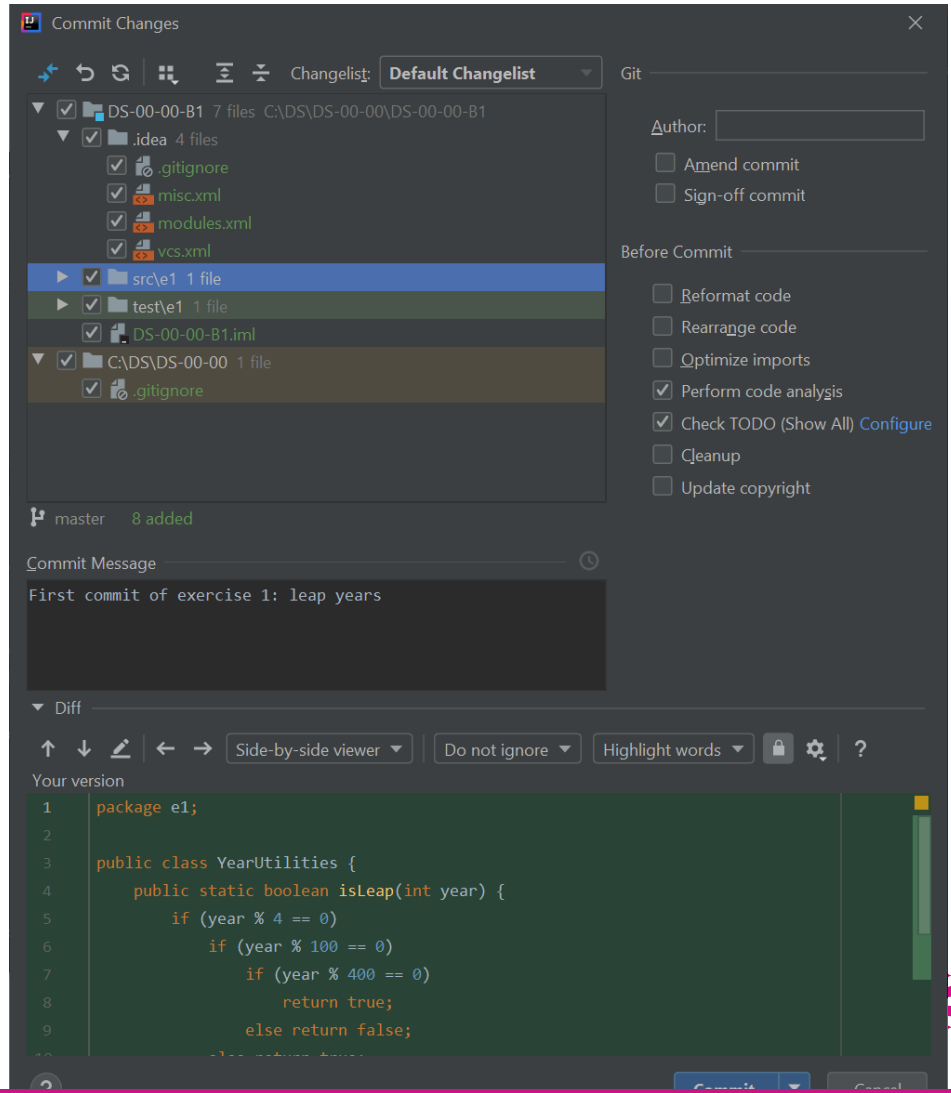
Uploading the project into the Repository with IntelliJ IDEA

- Now we *commit* the changes to upload the versioned files.



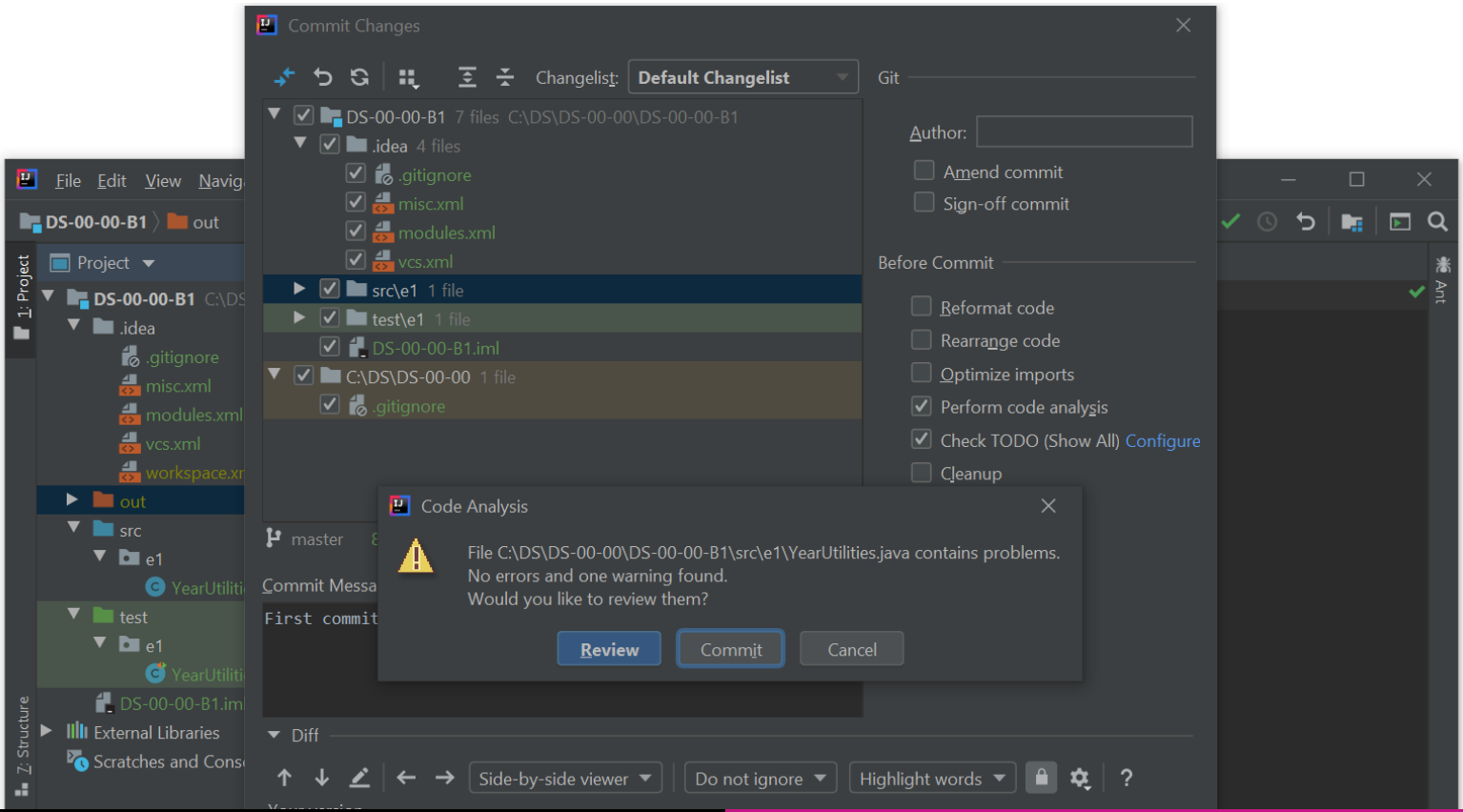
Uploading the project into the Repository with IntelliJ IDEA

- All that is marked in green will be uploaded.
- All that is ignored will not appear.
- The local files that are not versioned (from the `src` and `test` folders) will appear and we must mark them for uploading.
- A small *diff* section will appear, which lets us see the changes being made.
- Messages are mandatory and should be self-explanatory.



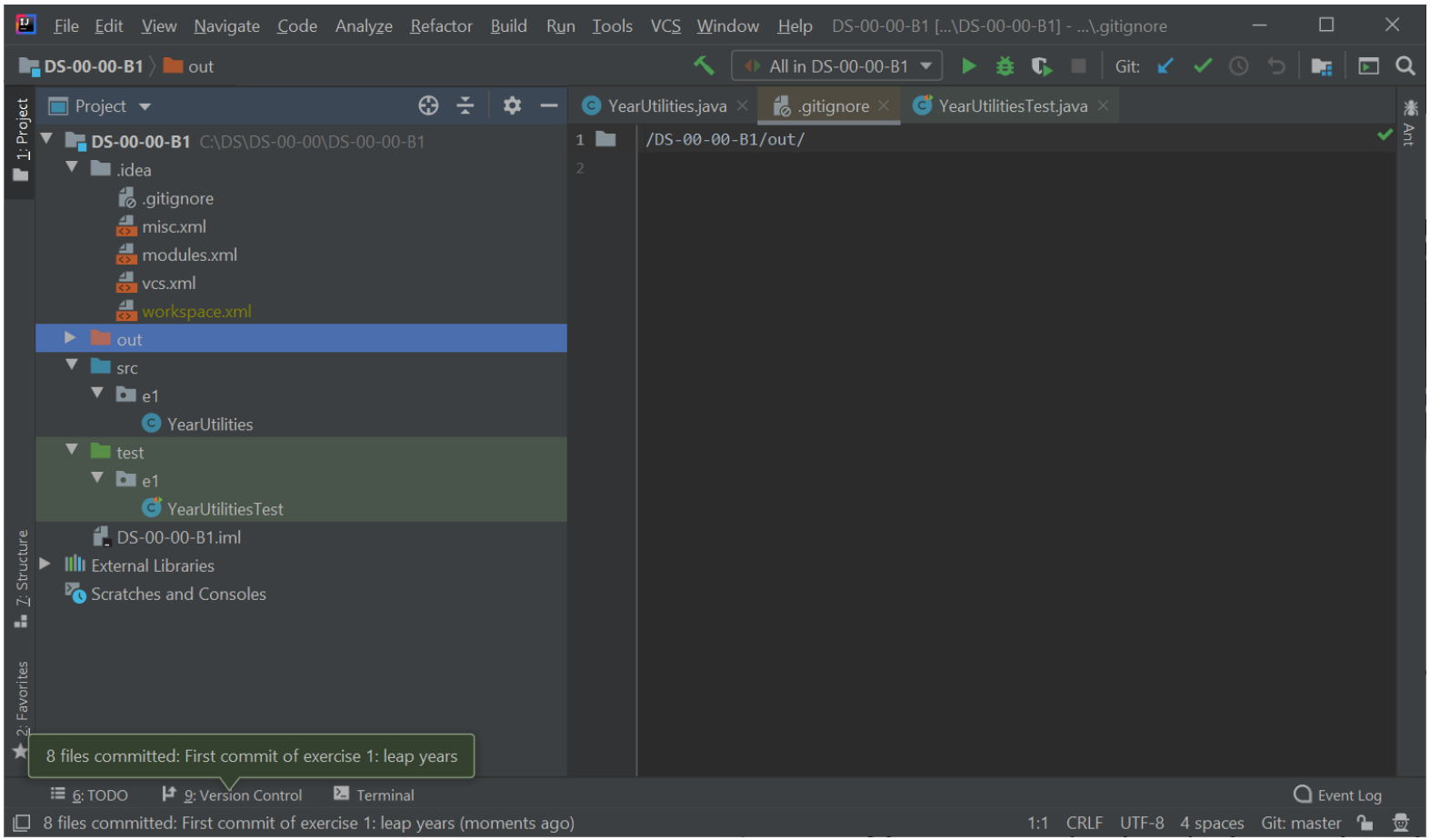
Uploading the project into the Repository with IntelliJ IDEA

- If *Perform code analysis* is enabled, it may tell us to revise the code before we *commit*.
- We can ignore it and *commit* directly, or follow the IDE's recommendations and *commit* later.



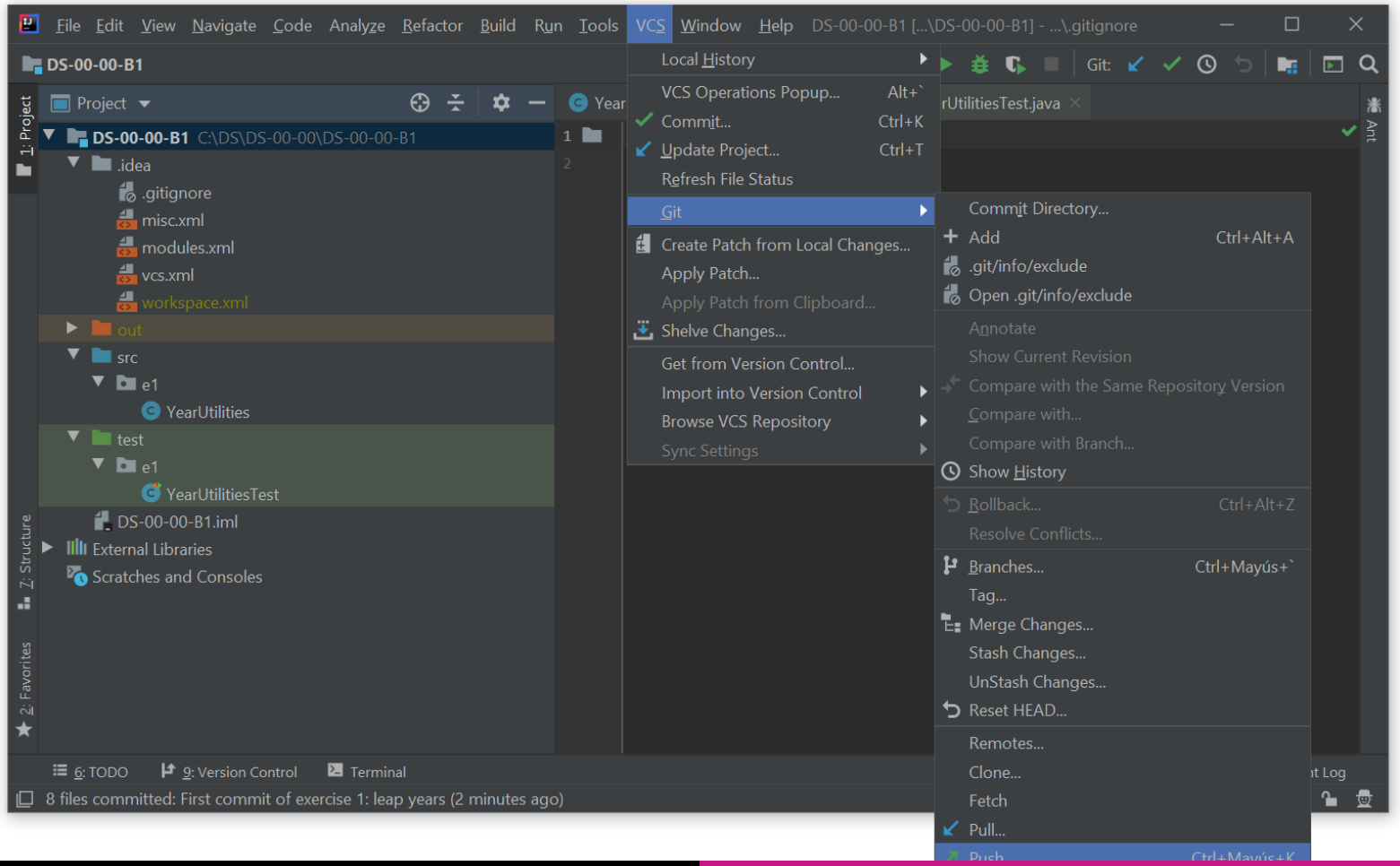
Uploading the project into the Repository with IntelliJ IDEA

- IDEA confirms that changes have been committed. However, they are so far only in our local repository.



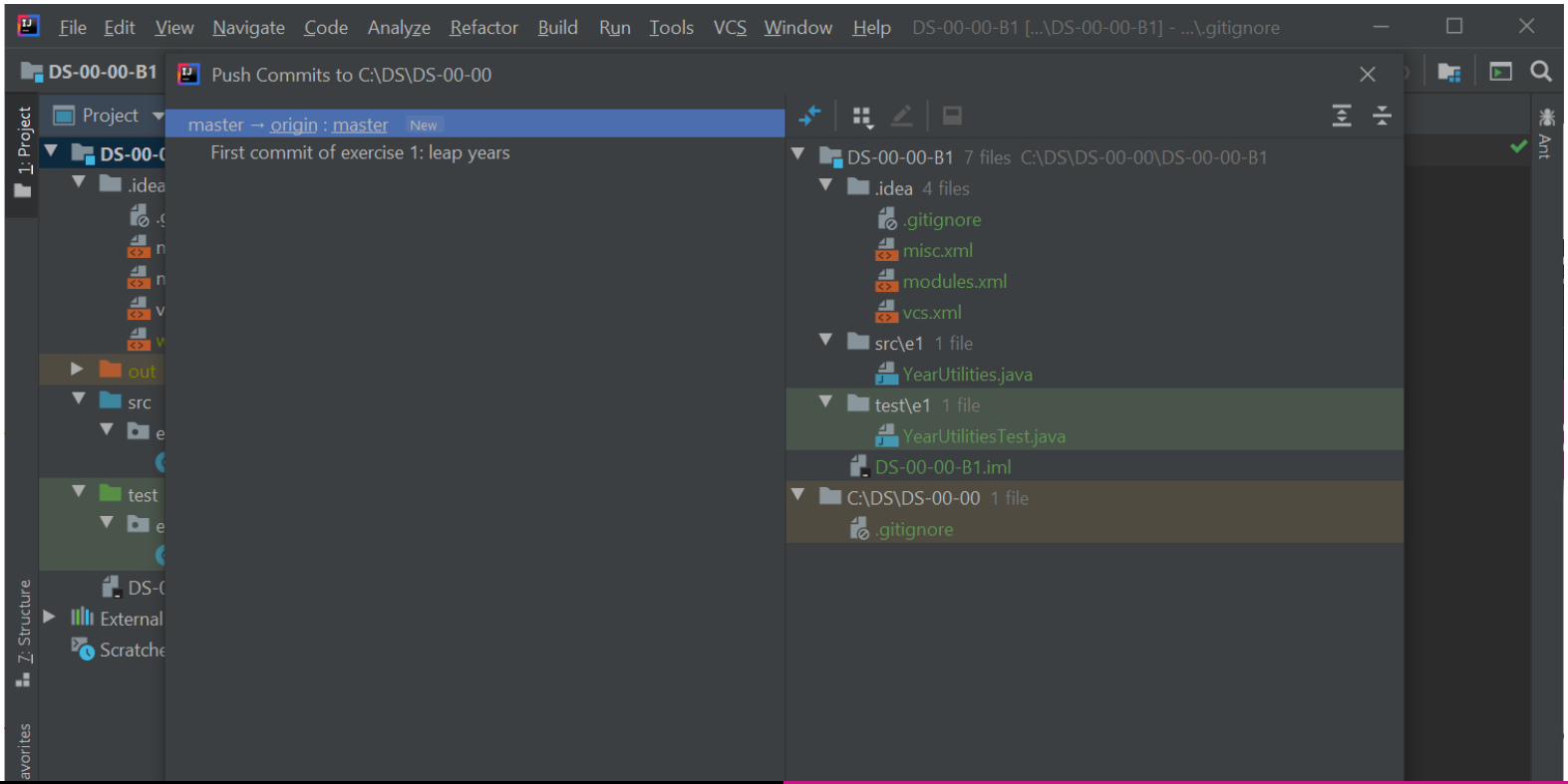
Uploading the project into the Repository with IntelliJ IDEA

- To upload the files into the remote repository, the action is to *push* the changes.



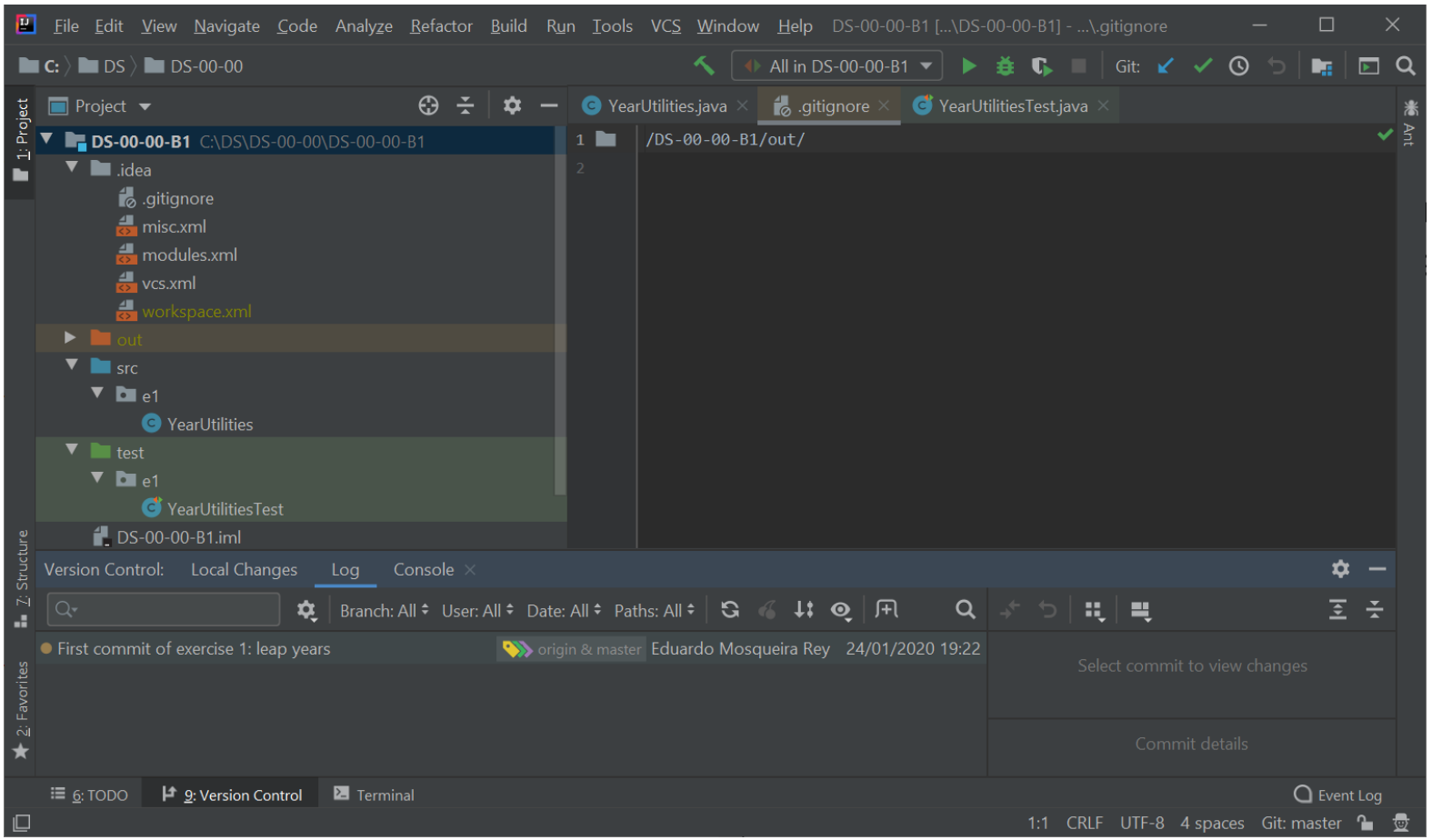
Uploading the project into the Repository with IntelliJ IDEA

- It will ask us which remote branch will be associated with our local branch.
- As we can see, it associates by default `master` to `origin/master`, so no further action is required. We simply click *Push*.



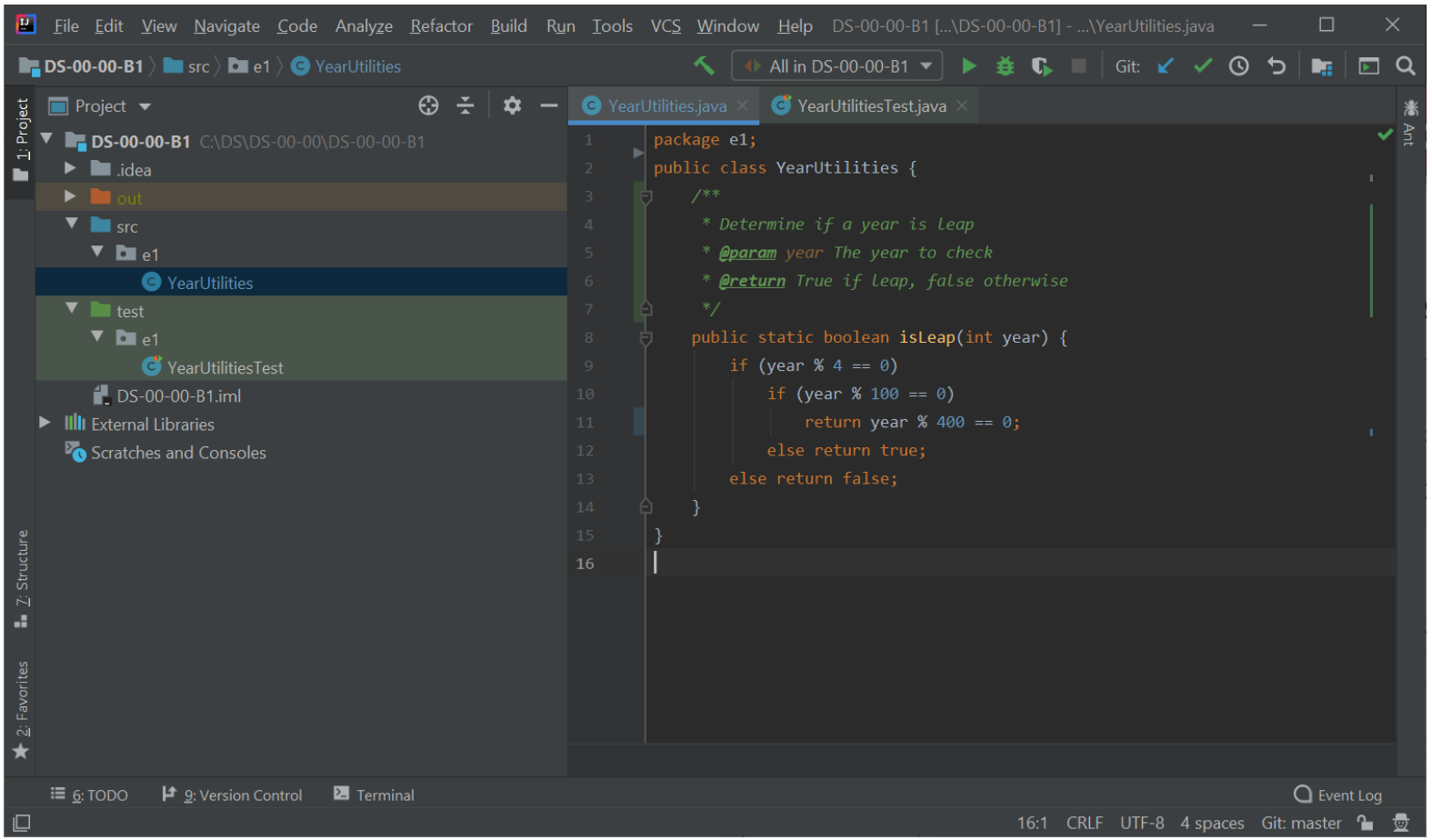
Uploading the project into the Repository with IntelliJ IDEA

- If pushing succeeds, a notification *pop-up* will appear.
- We can see the result in *Version Control - Log*.



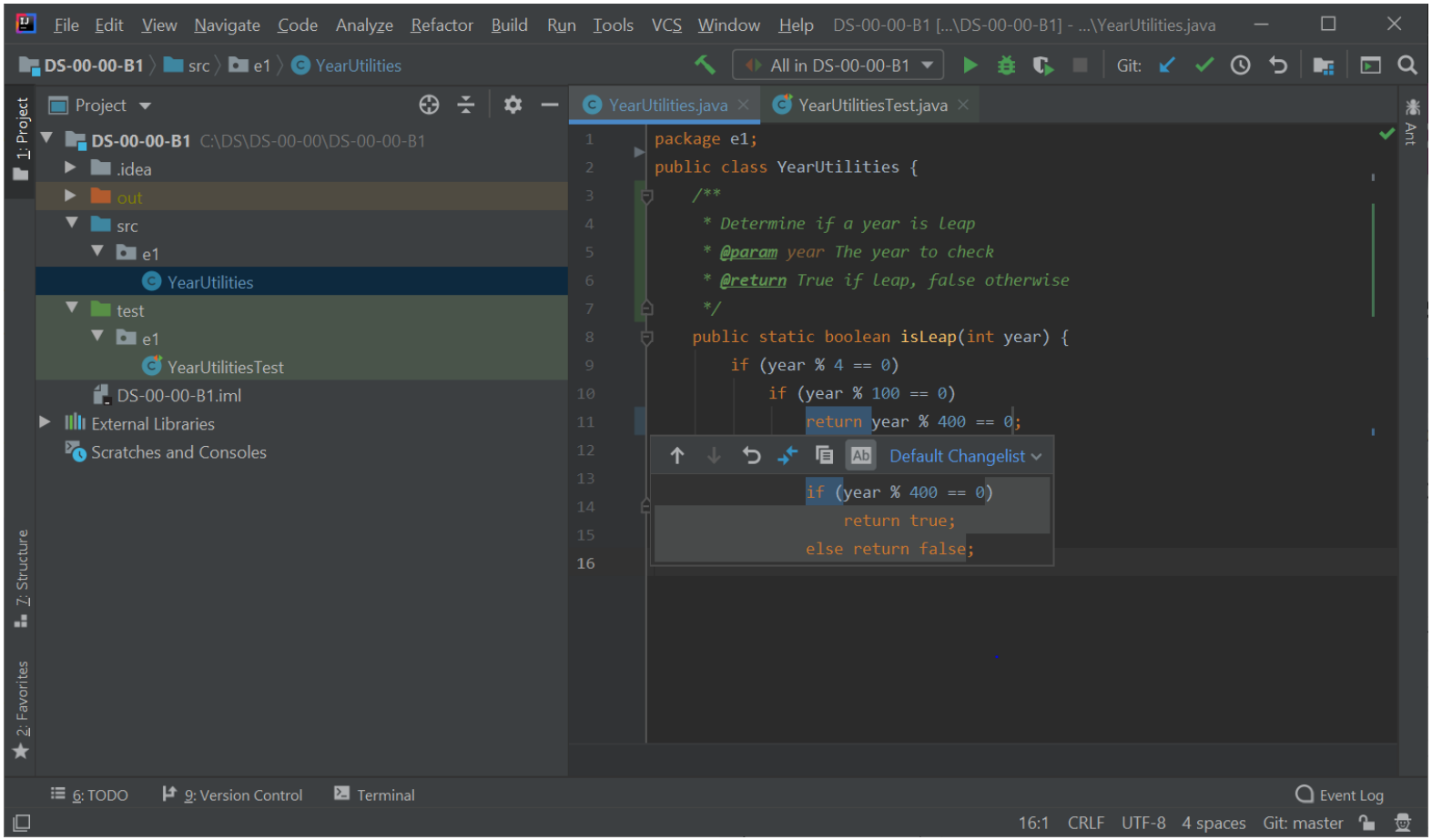
Editing code

- We can now make changes to our code.
- The new lines are highlighted in green on the side, the modified lines are in blue and the deleted lines have a triangle.



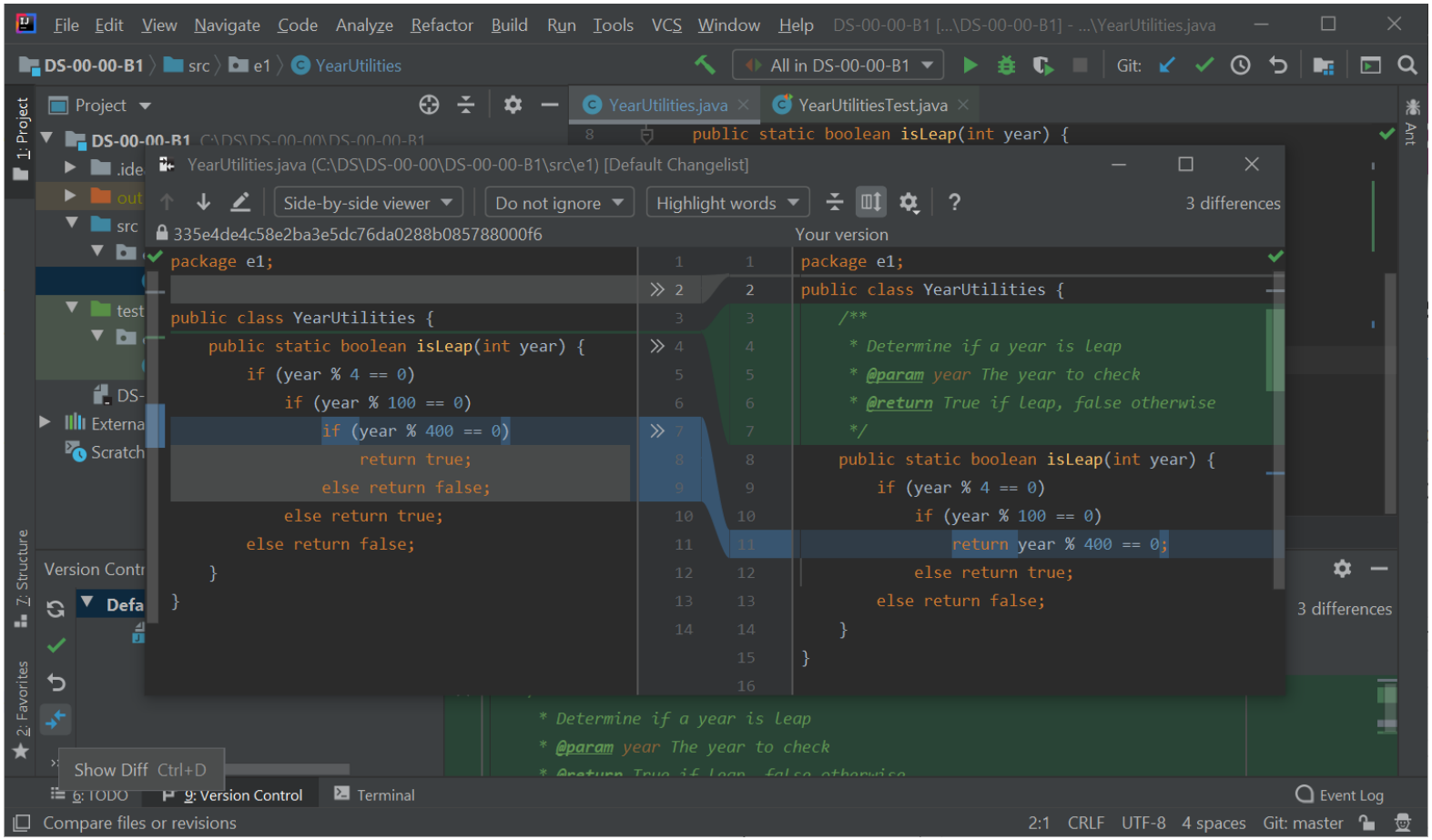
Editing code

- Clicking on the side we can see the changes in more detail.



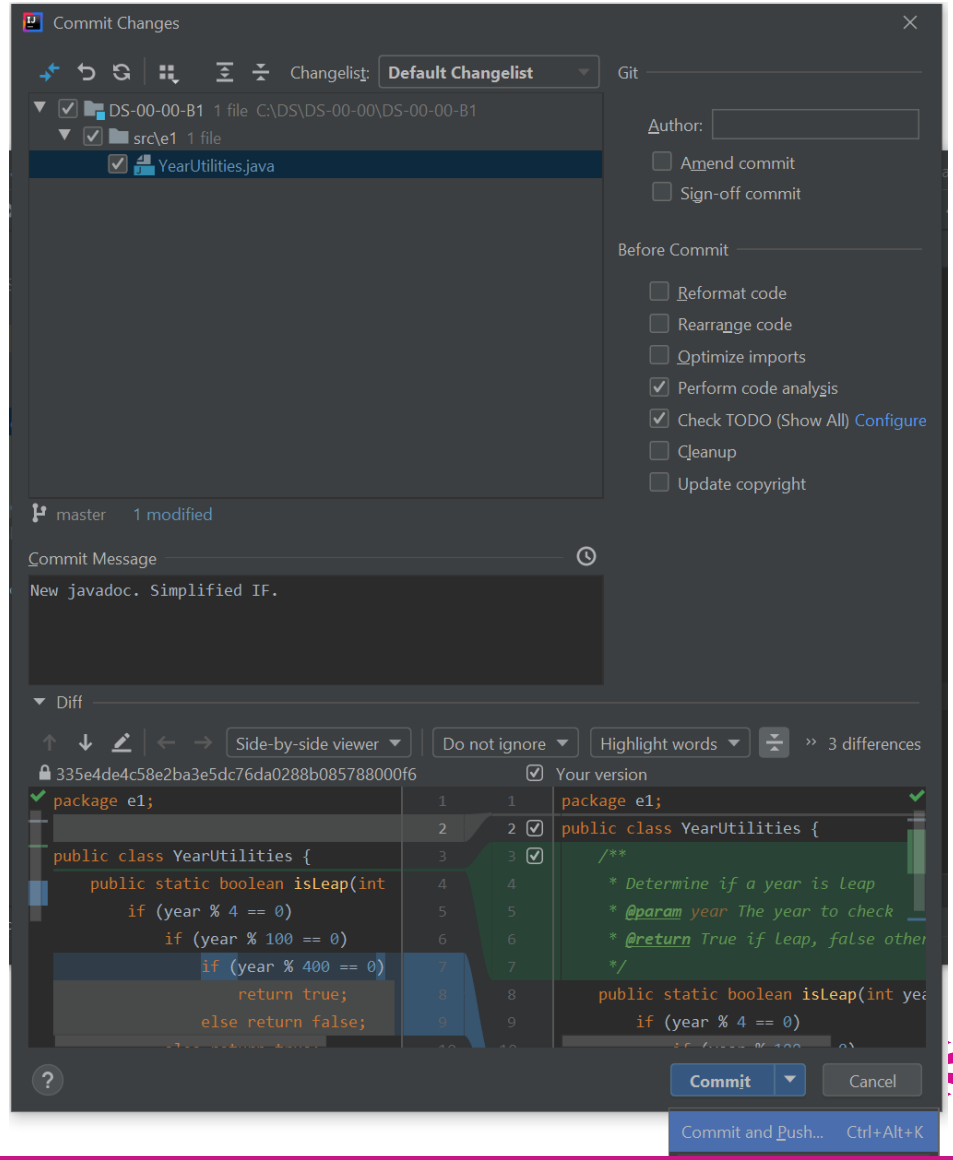
Editing code

- We can see the changes for the entire file selecting *diff* from the *Version Control* window.



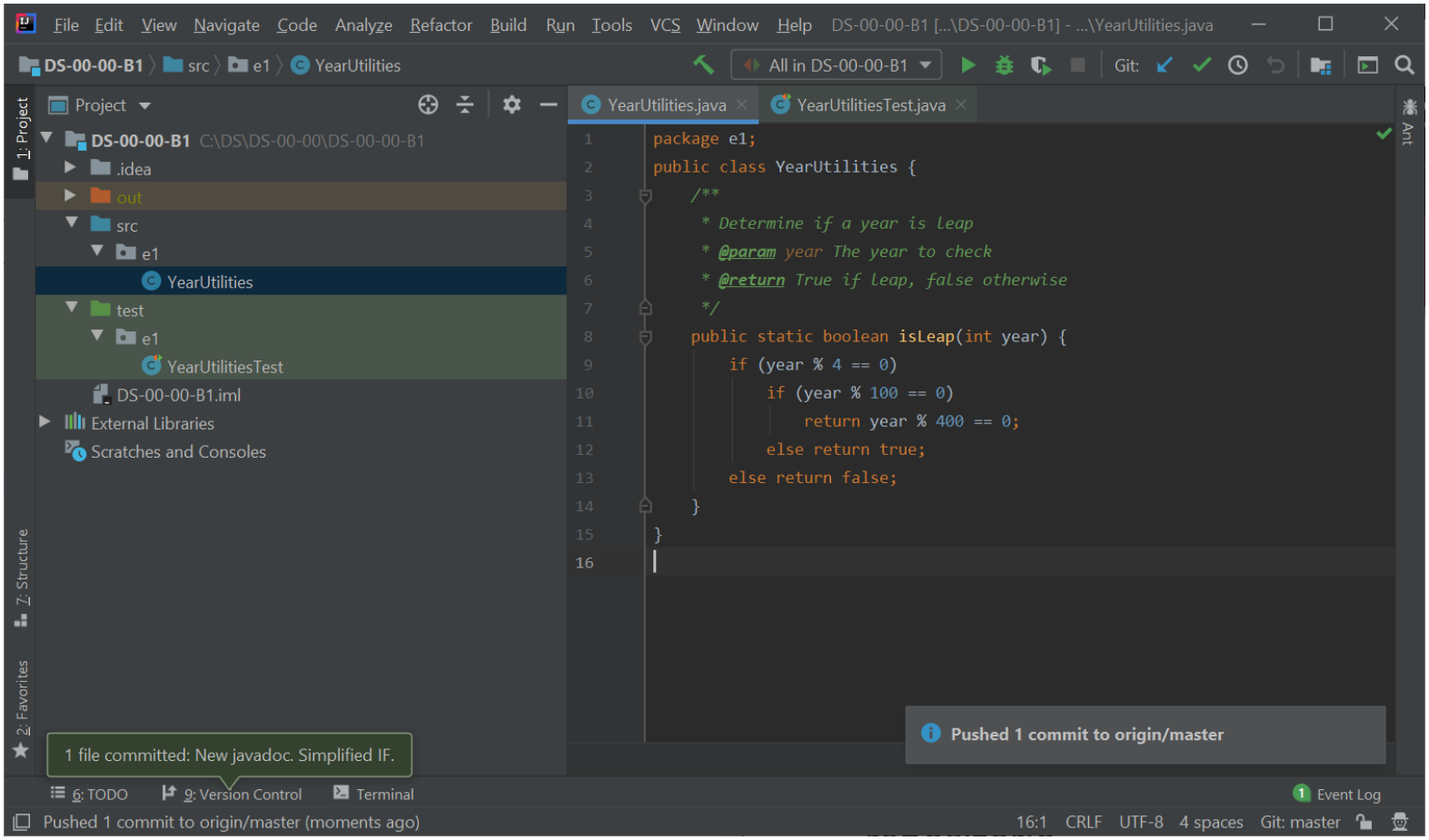
Sending changes

- Performing a new *commit* will submit the changes to the repository.
- To make it faster, we can *commit* and *push* at the same time.



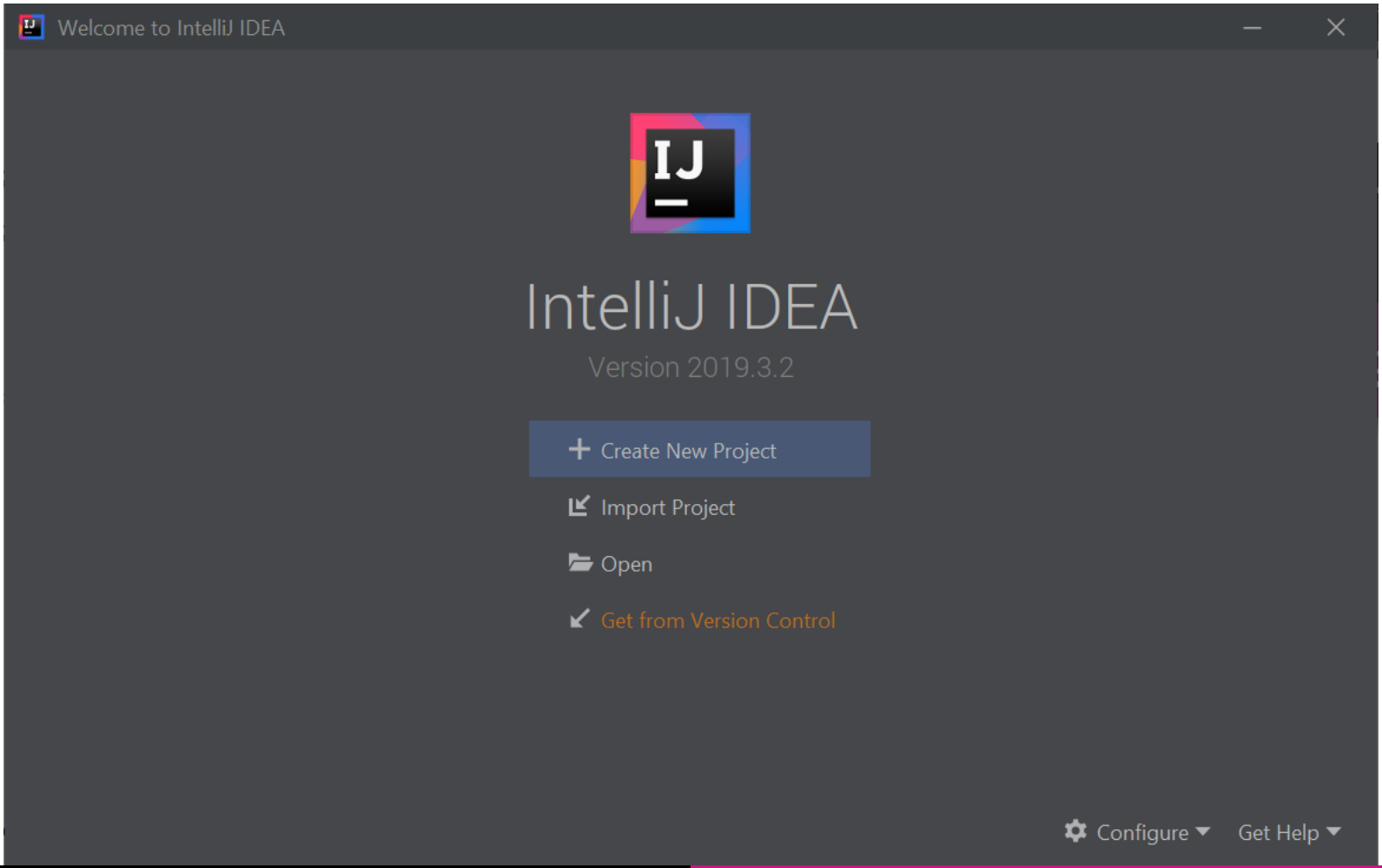
Sending changes

- If everything went as expected, two *pop-ups* will appear, one for the *commit* and another for the *push*.



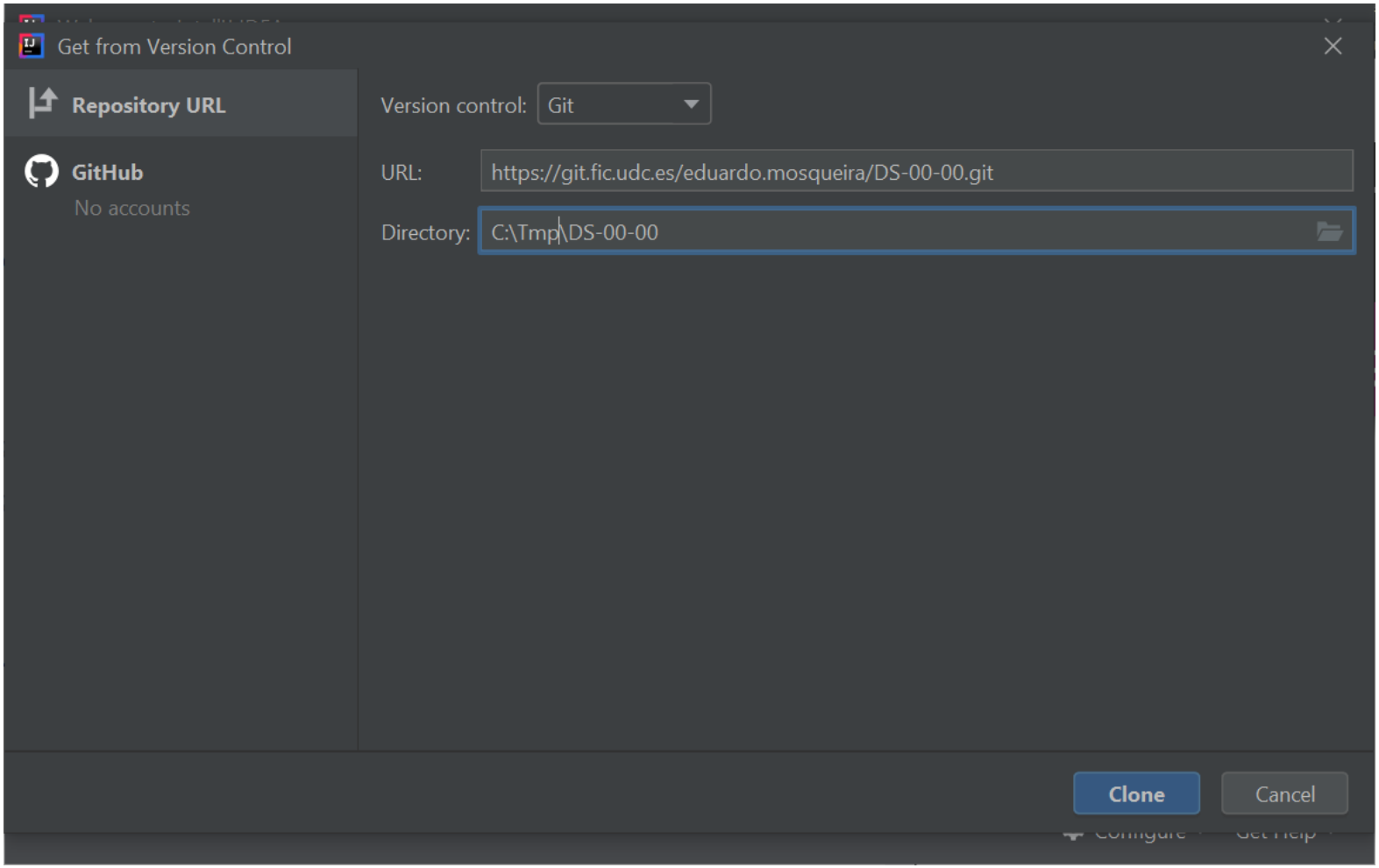
Downloading a fresh copy of the project

- In order to download a fresh copy of the project into our computer, we can select *Get from Version Control* when launching IntelliJ IDEA.



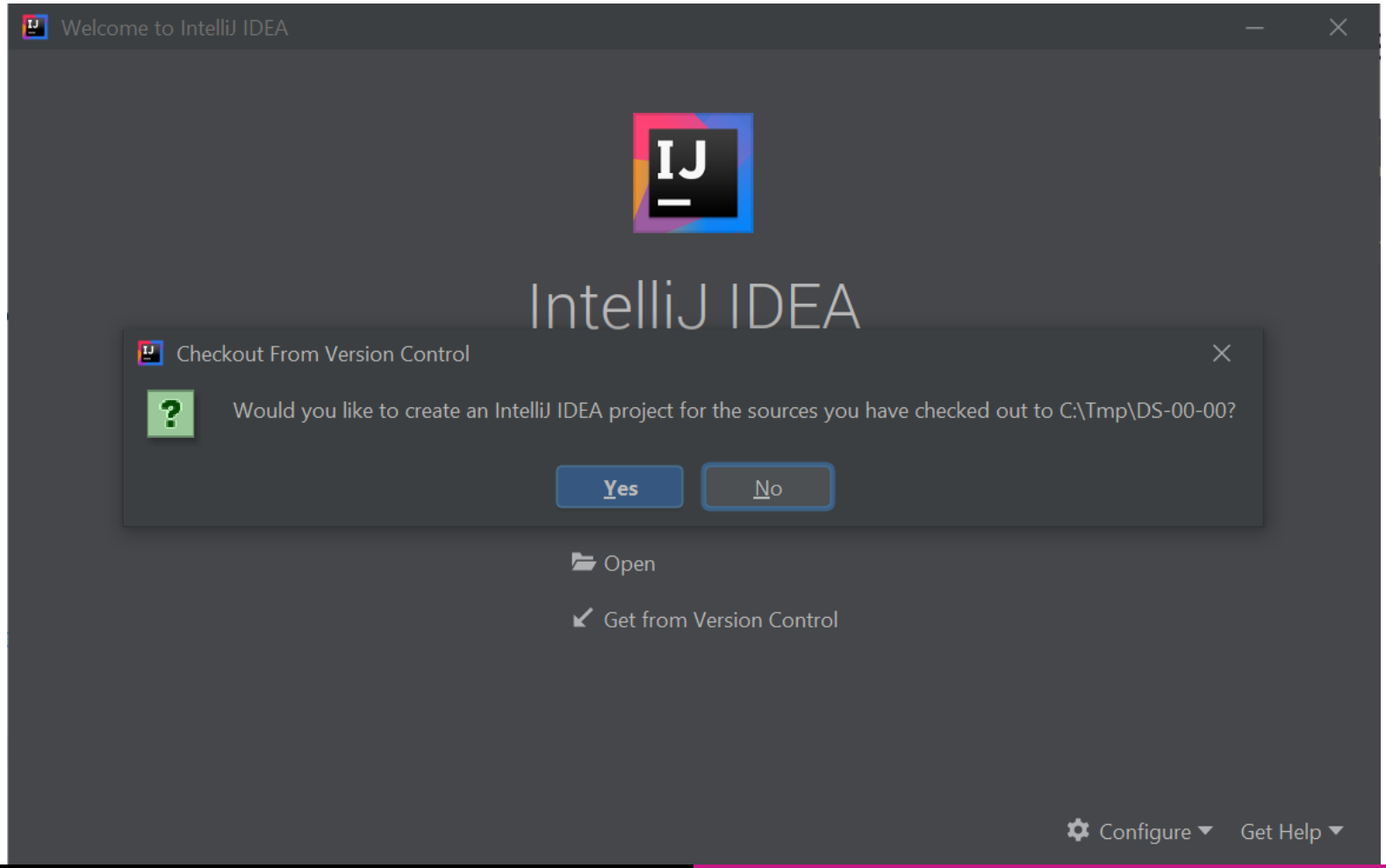
Downloading a fresh copy of the project

- We choose Git, enter the remote repository's location and the local folder where it will be downloaded. Then we click *Clone*.



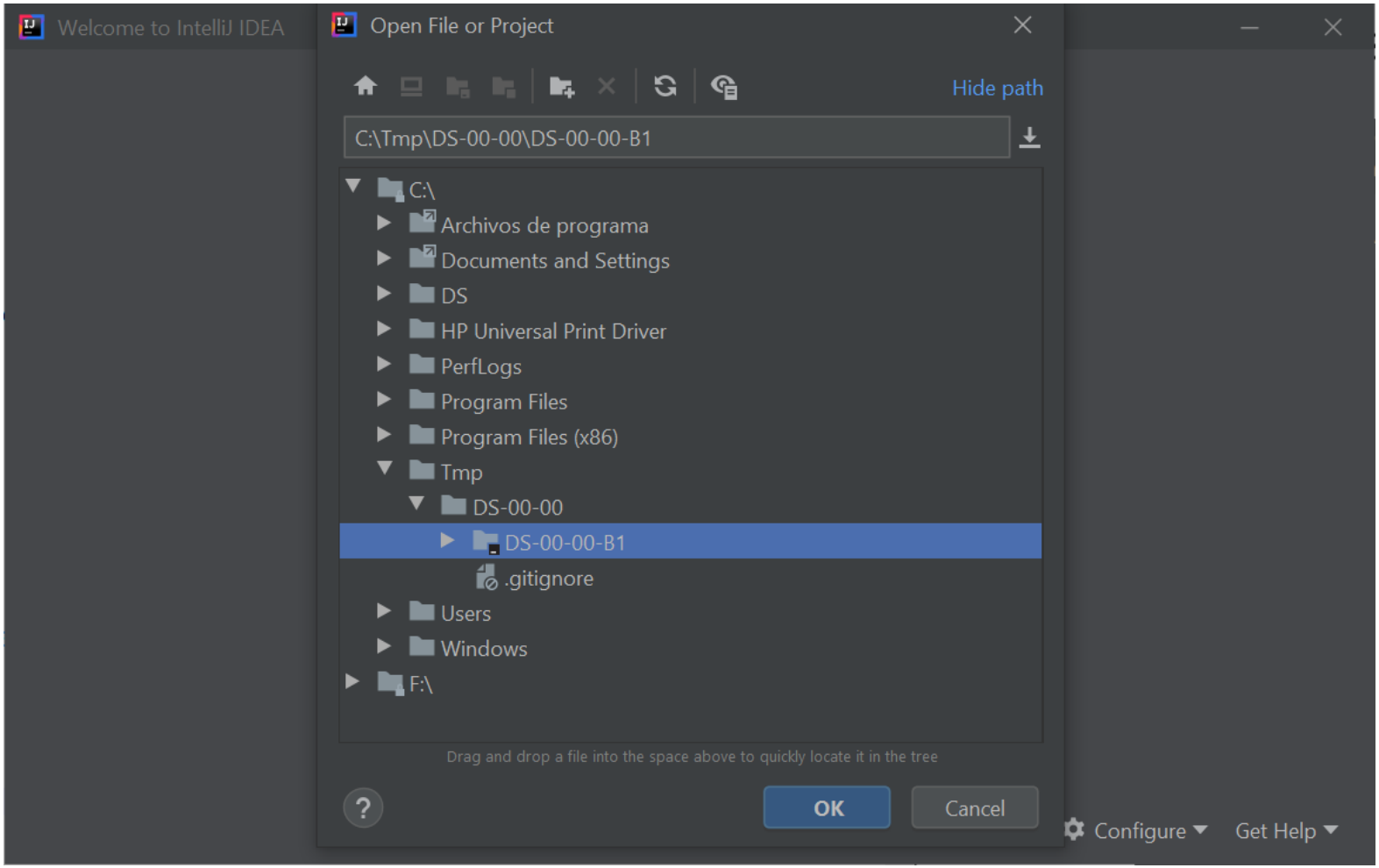
Downloading a fresh copy of the project

- It is not necessary to state whether you want to create a new IntelliJ IDEA project for those sources, since we have already created one. So we choose no.



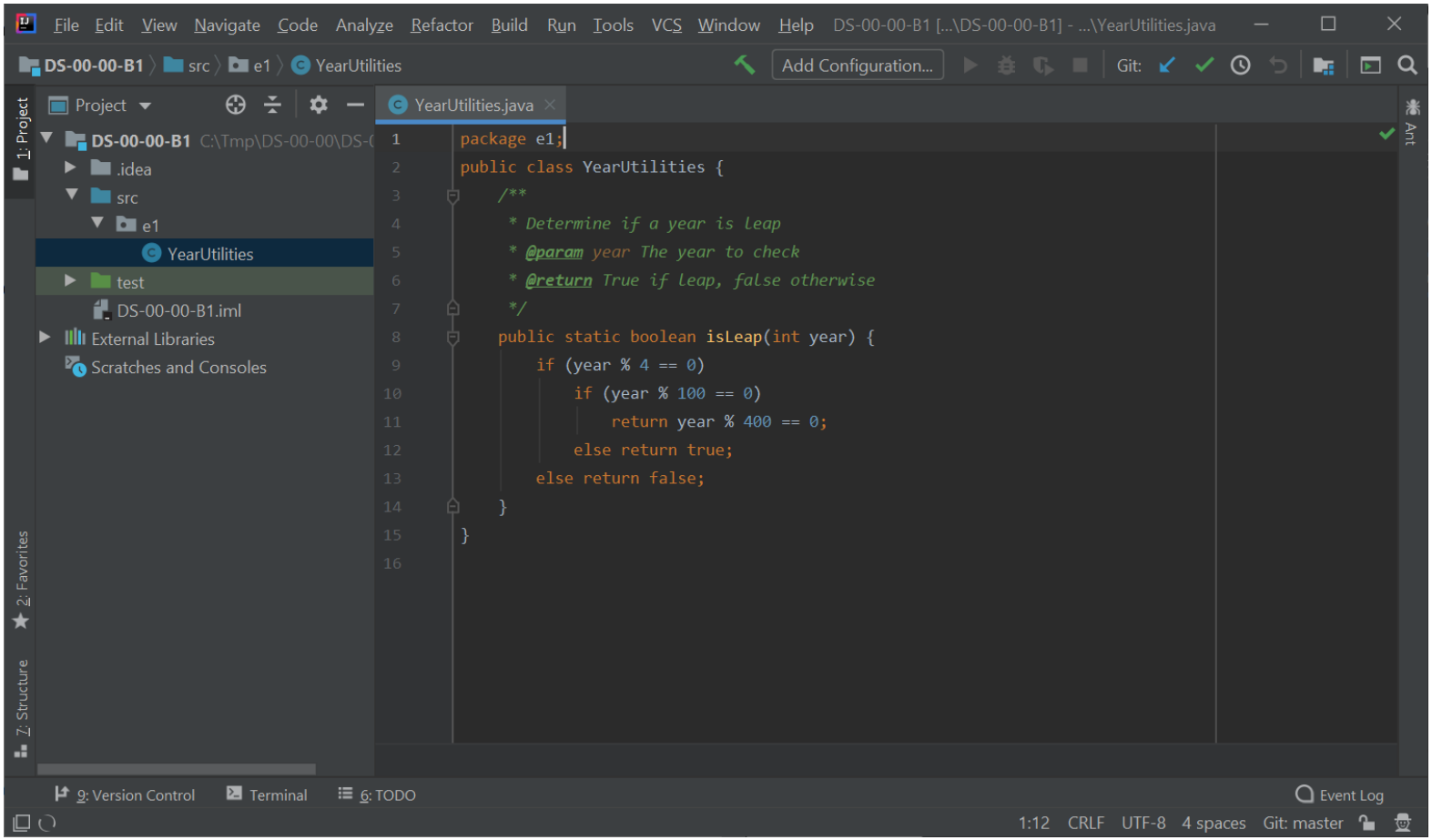
Downloading a fresh copy of the project

- The next step is to open the recently downloaded project.



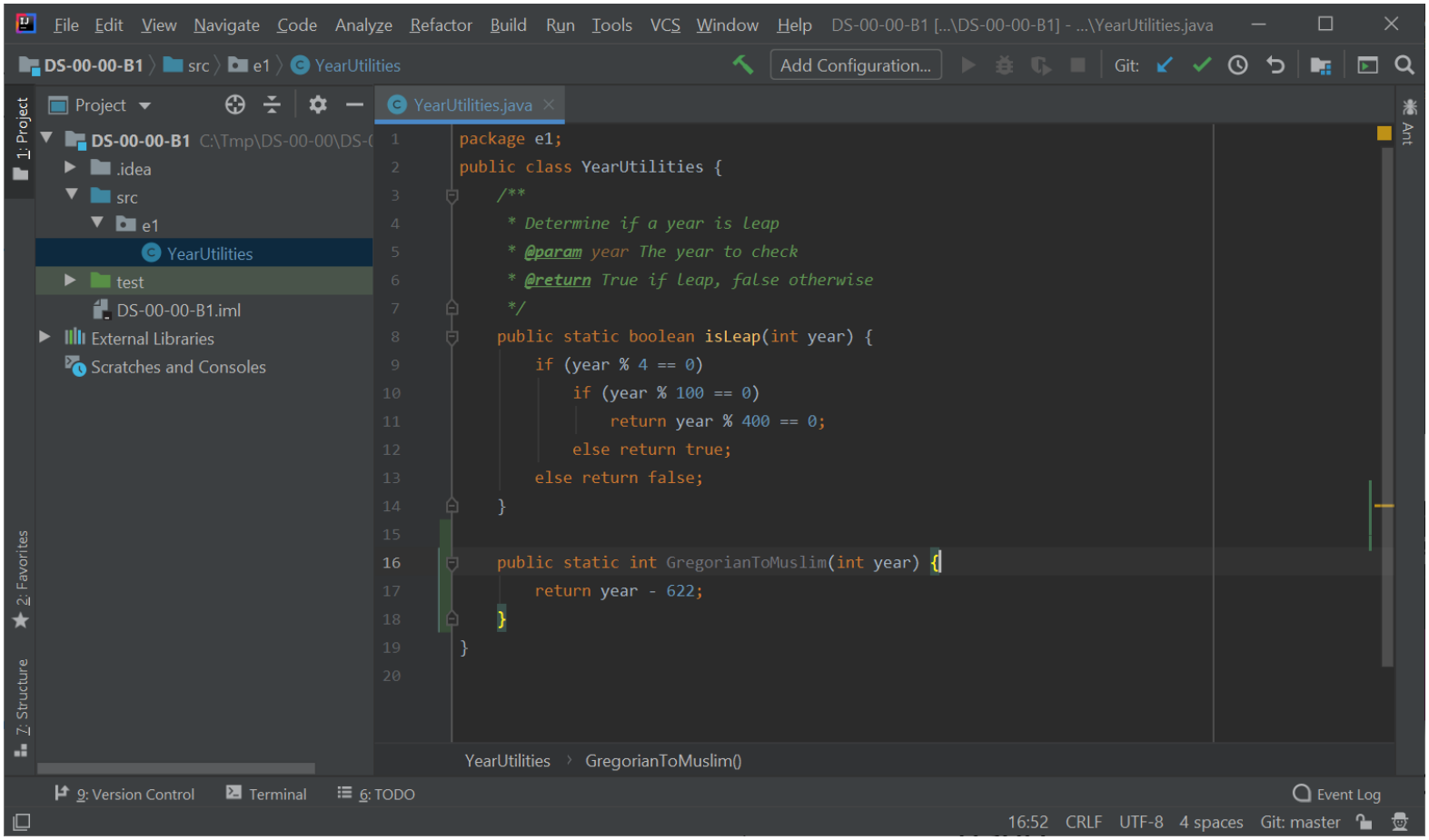
Downloading a fresh copy of the project

- We finally obtain a new copy of our project in another computer.



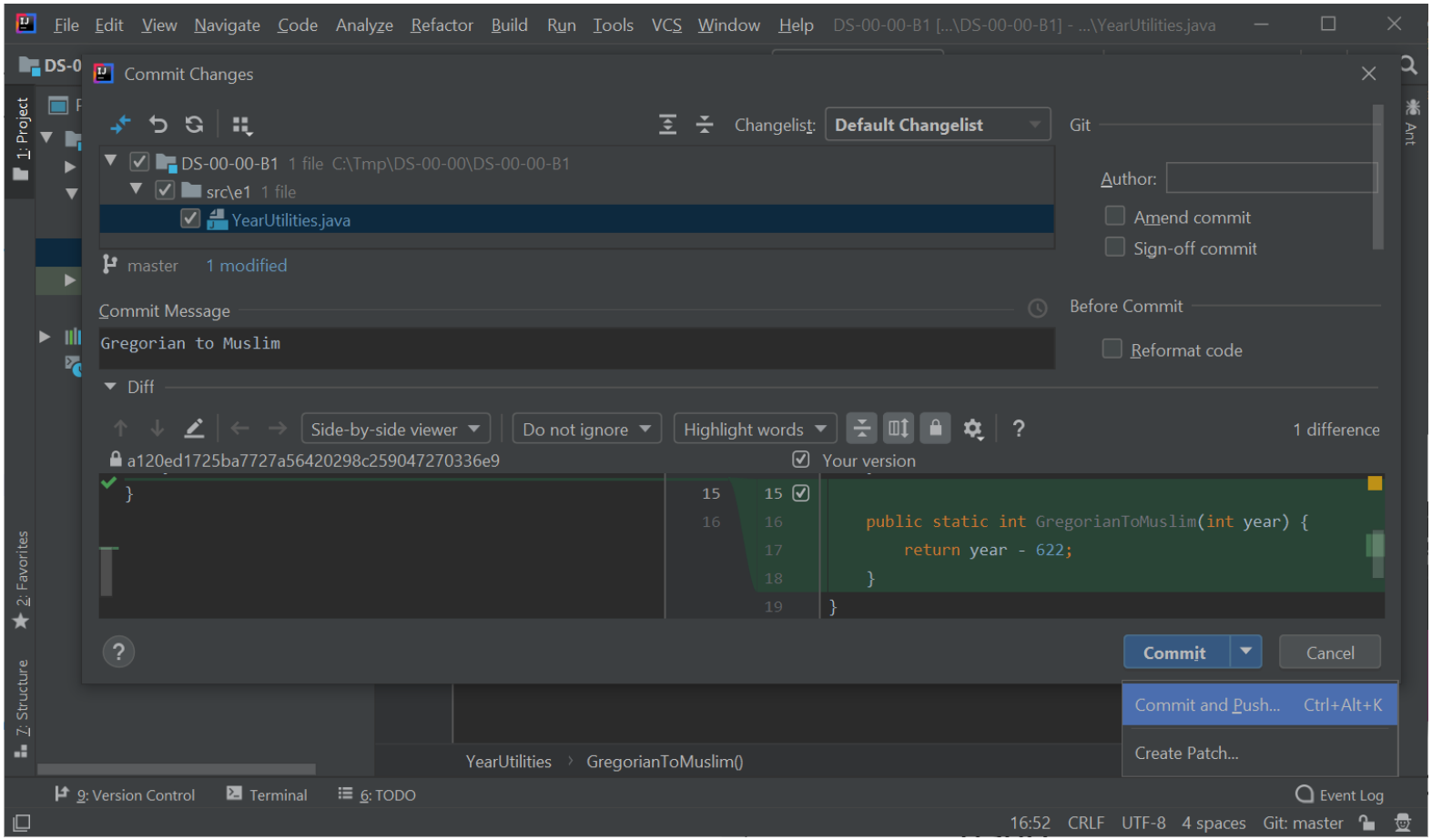
Resolving conflicts

- From this new copy we add a new conversion method, from Gregorian years to Muslim years...



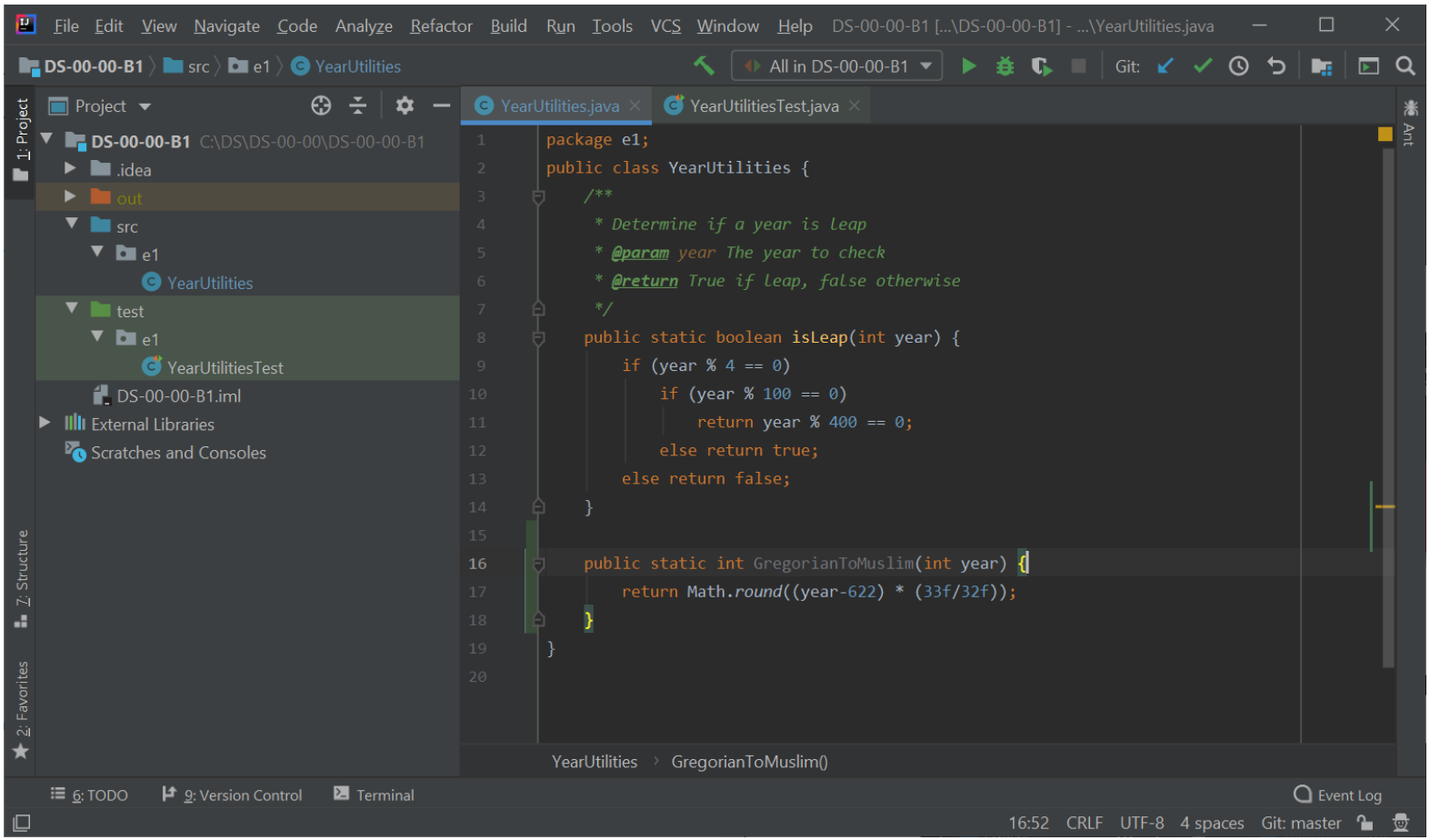
Resolving conflicts

- ...and *commit* and *push*.



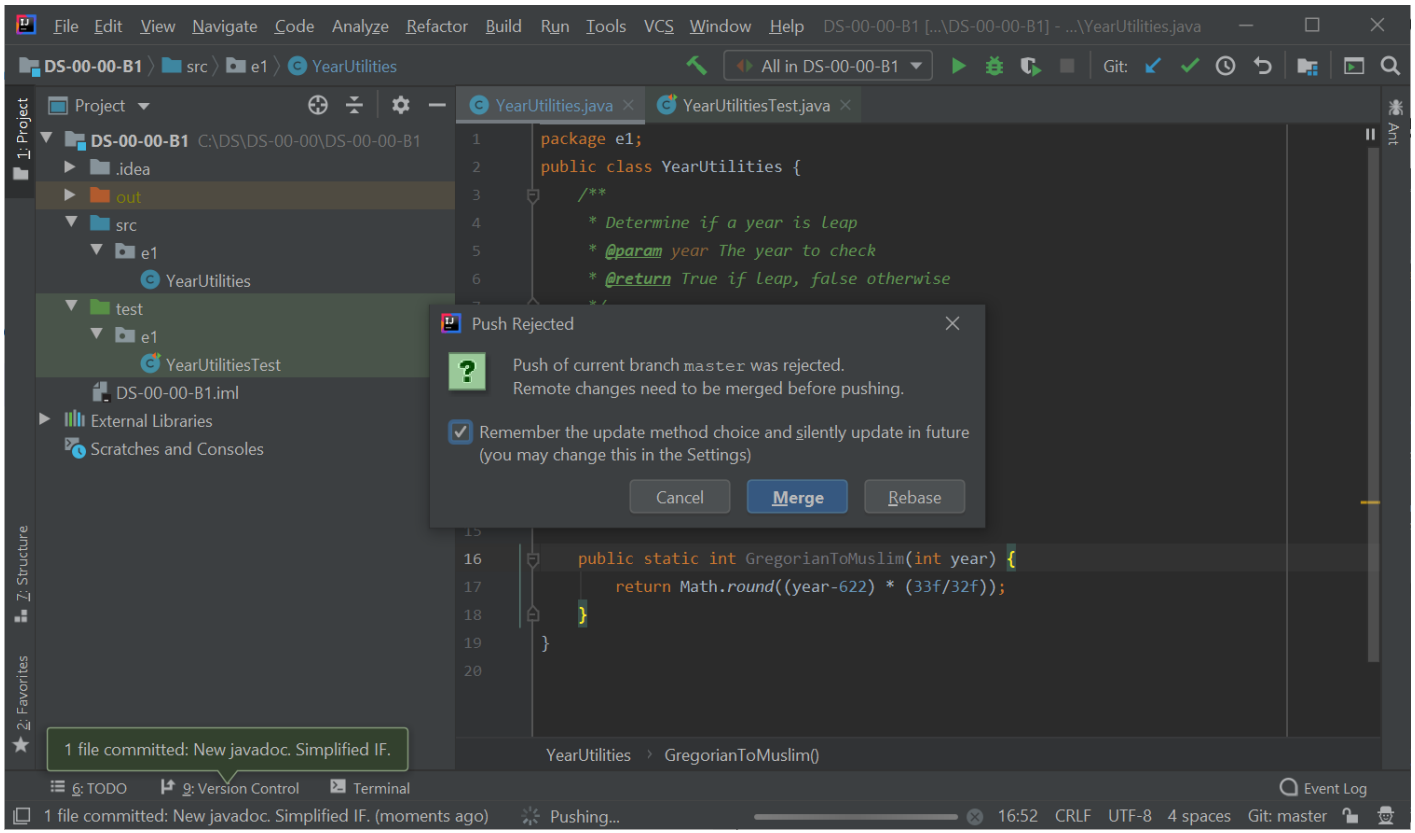
Resolving conflicts

- Due to a coordination error, our partner had simultaneously created a different, more succint version of the `GregorianToMuslim` method.



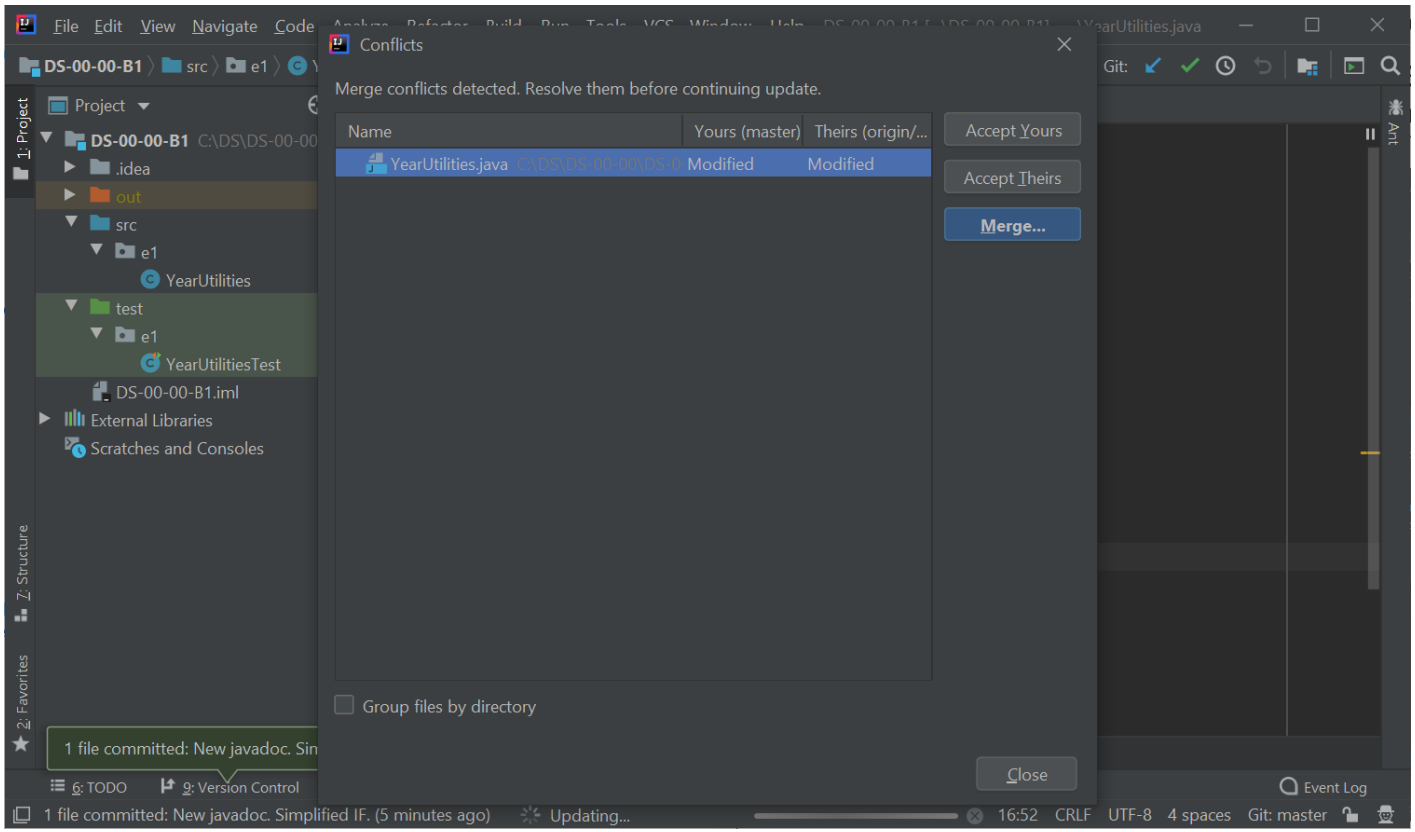
Resolving conflicts

- If we try to *commit & push*, we'll see that the first command succeeds (it is a local operation) but the second one fails because the repository has another version which is newer than the current one.
- It tells us to ***Merge*** the contents of the repository with the local copy. **Do not choose *Rebase***, always *Merge*.



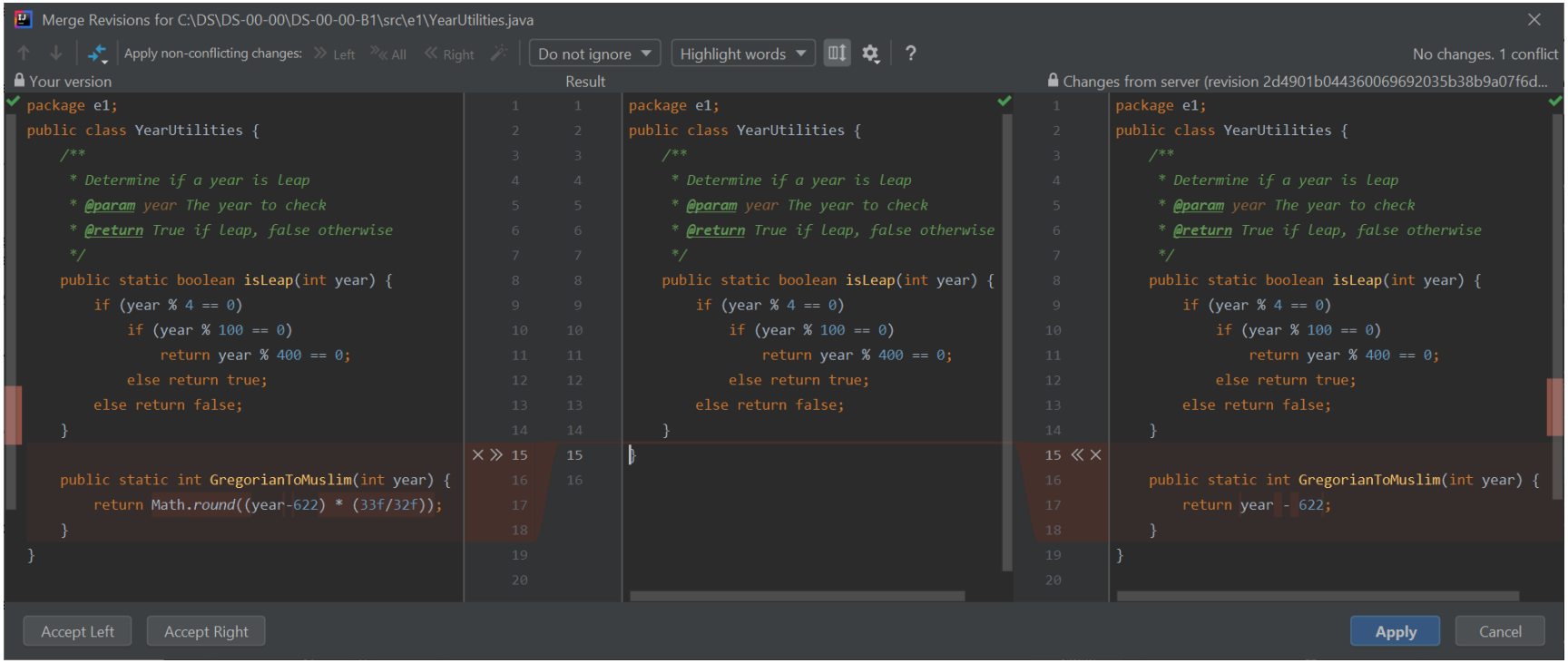
Resolving conflicts

- When we try to merge *merge*, a conflict is created, since we have changed the same lines of code in two different local copies.
- A conflict resolution window appears, in which we can resolve it easily by choosing *Accept Yours* or *Accept Theirs*...



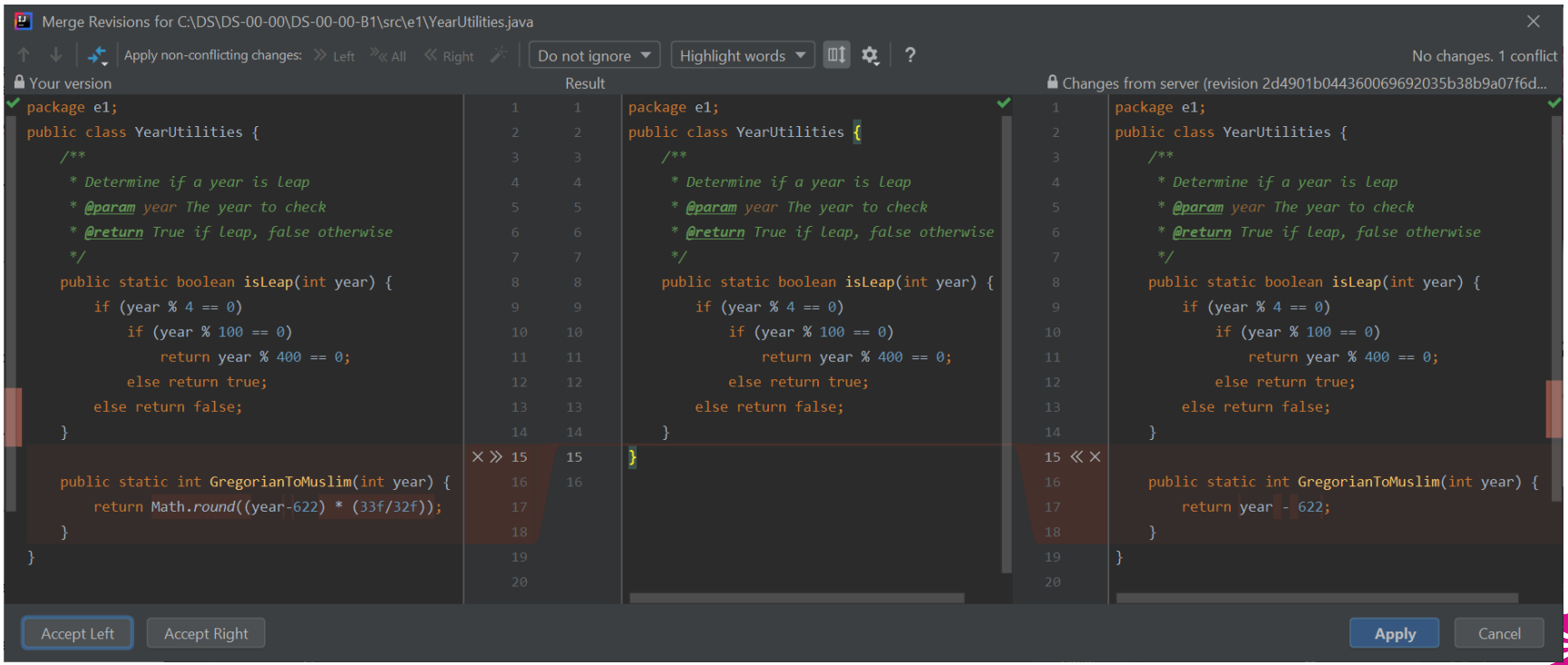
Resolving conflicts

- ...but if we are unsure, we can click on *Merge* to see which lines are in conflict.



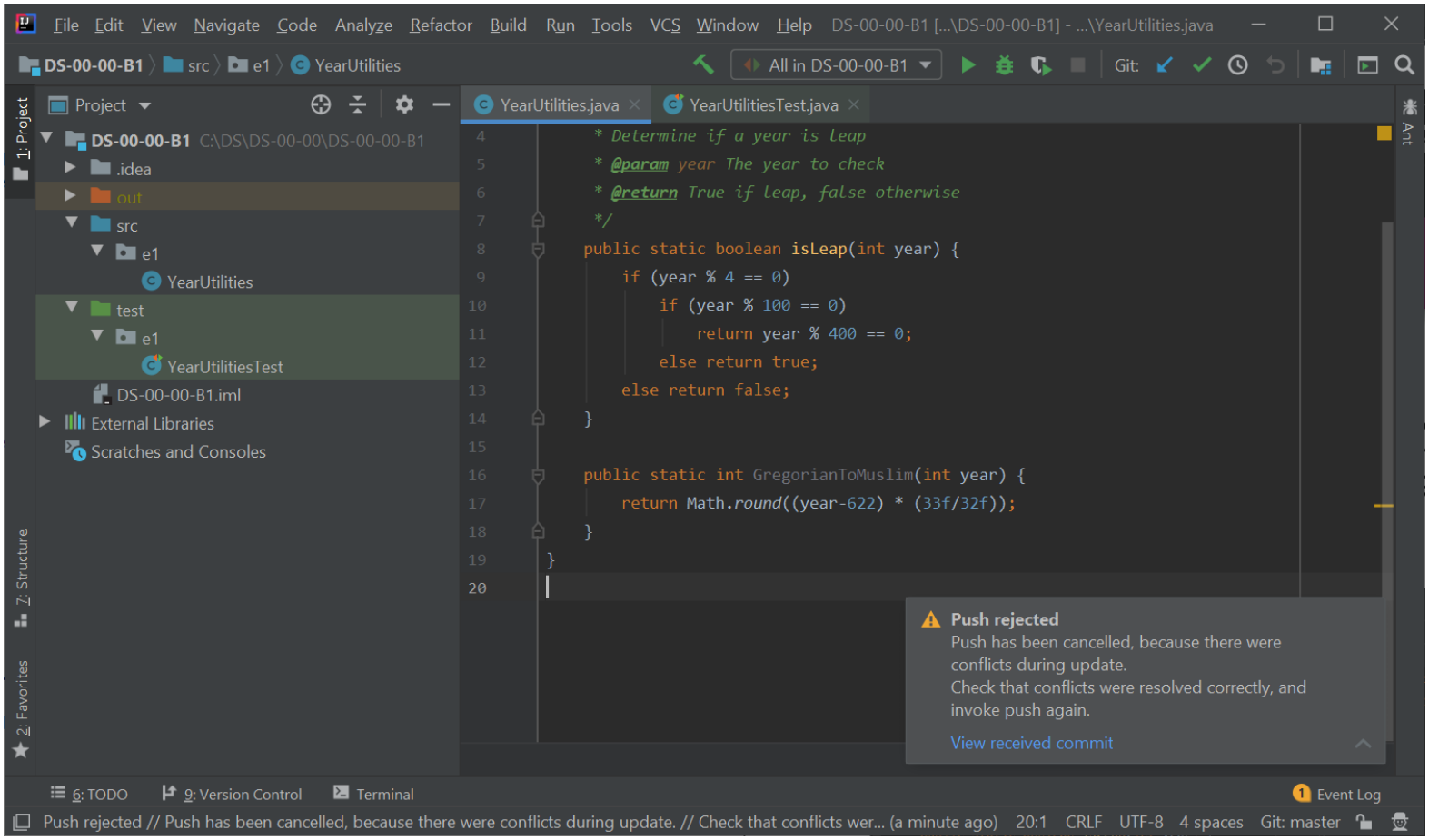
Resolving conflicts

- The *Merge* windows shows that our version (on the left) is more complete, so we select *Accept Left*.
- If we click on the >> or << arrows, we can accept changes from eithe side (assuming there are multiple conflicts).



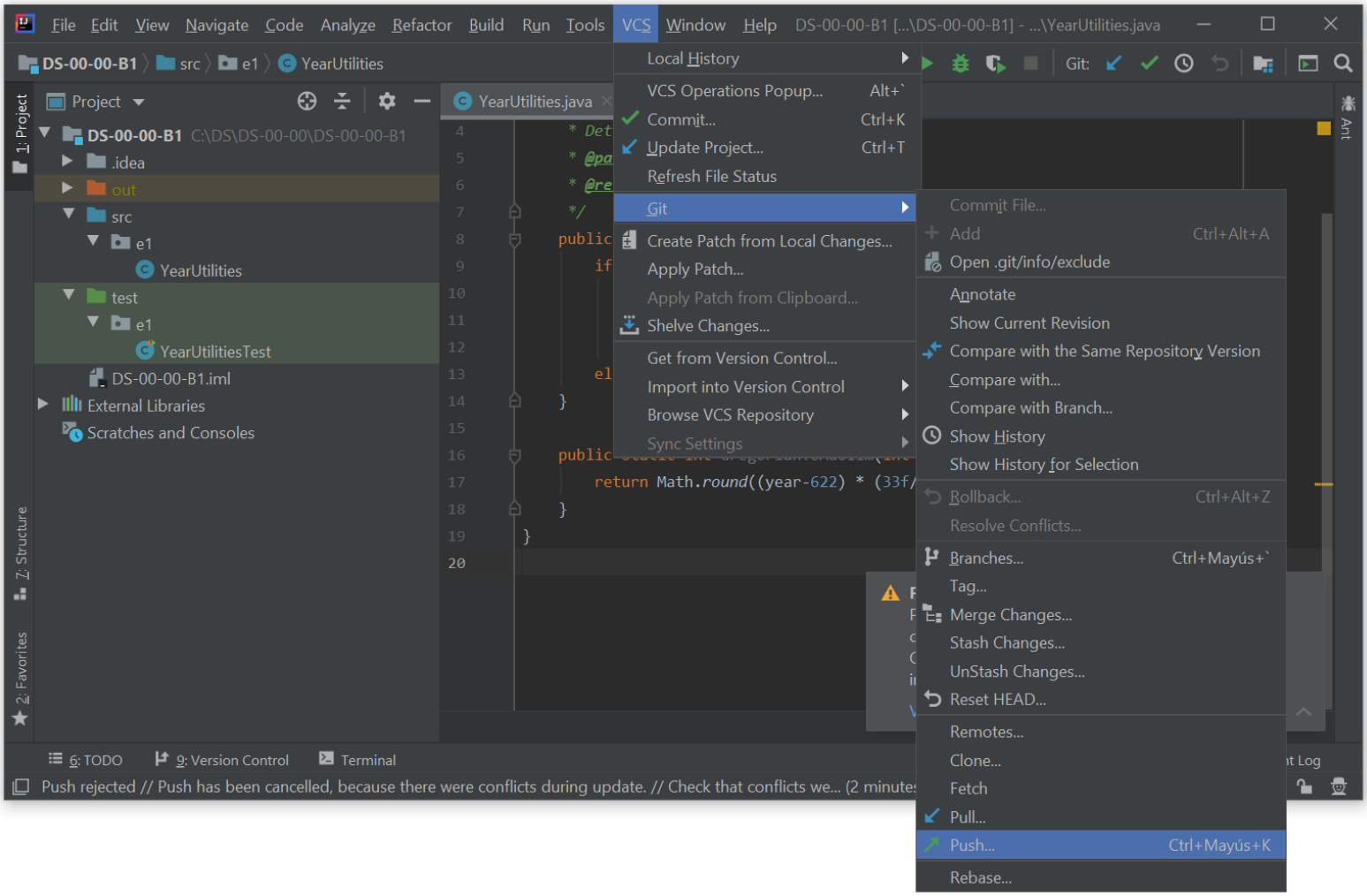
Resolving conflicts

- When it finishes, it reminds us that the *Push* failed. So, once the conflicts have been resolved, we must try it again...



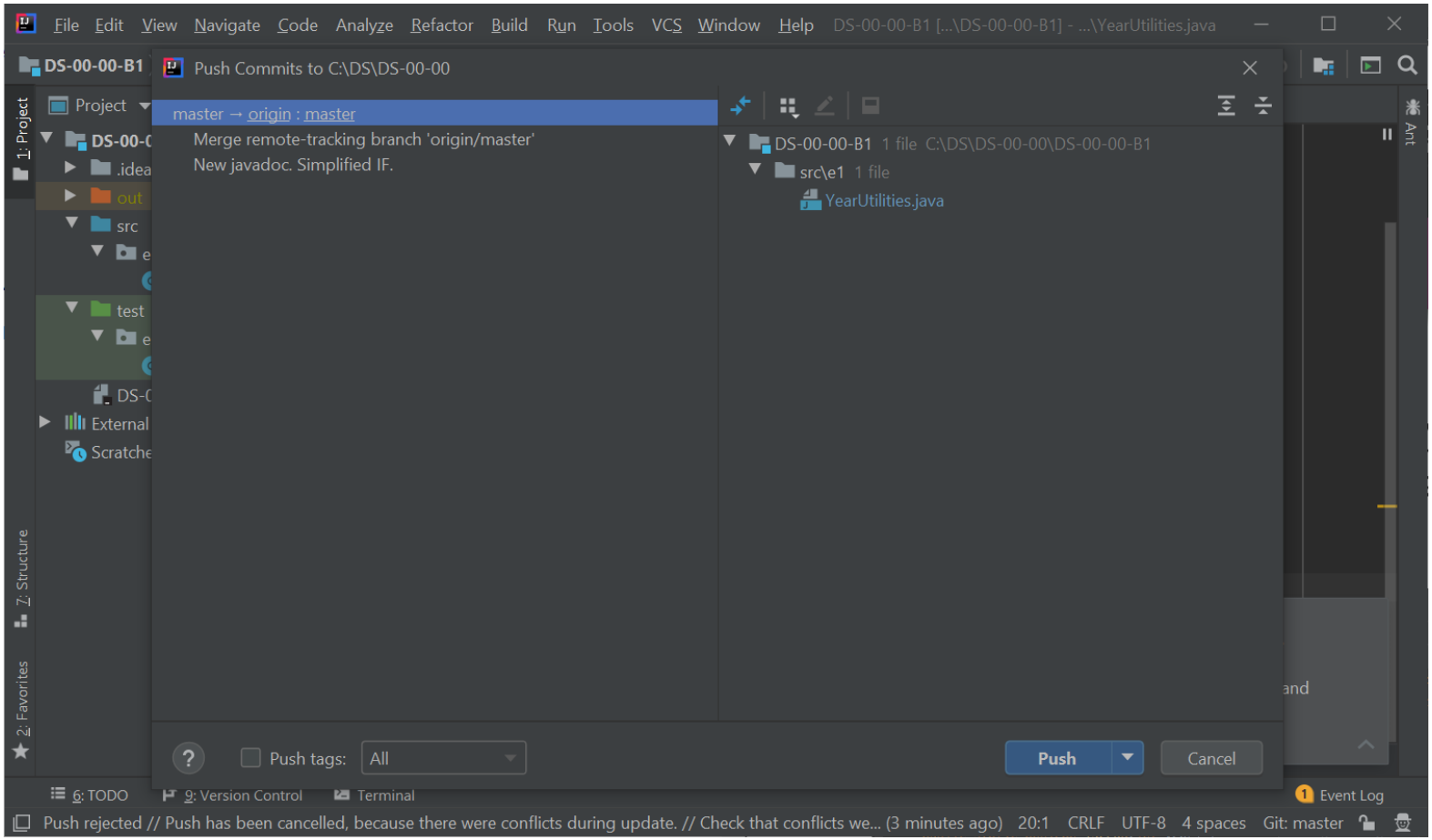
Resolving conflicts

- ...So we *Push* again...



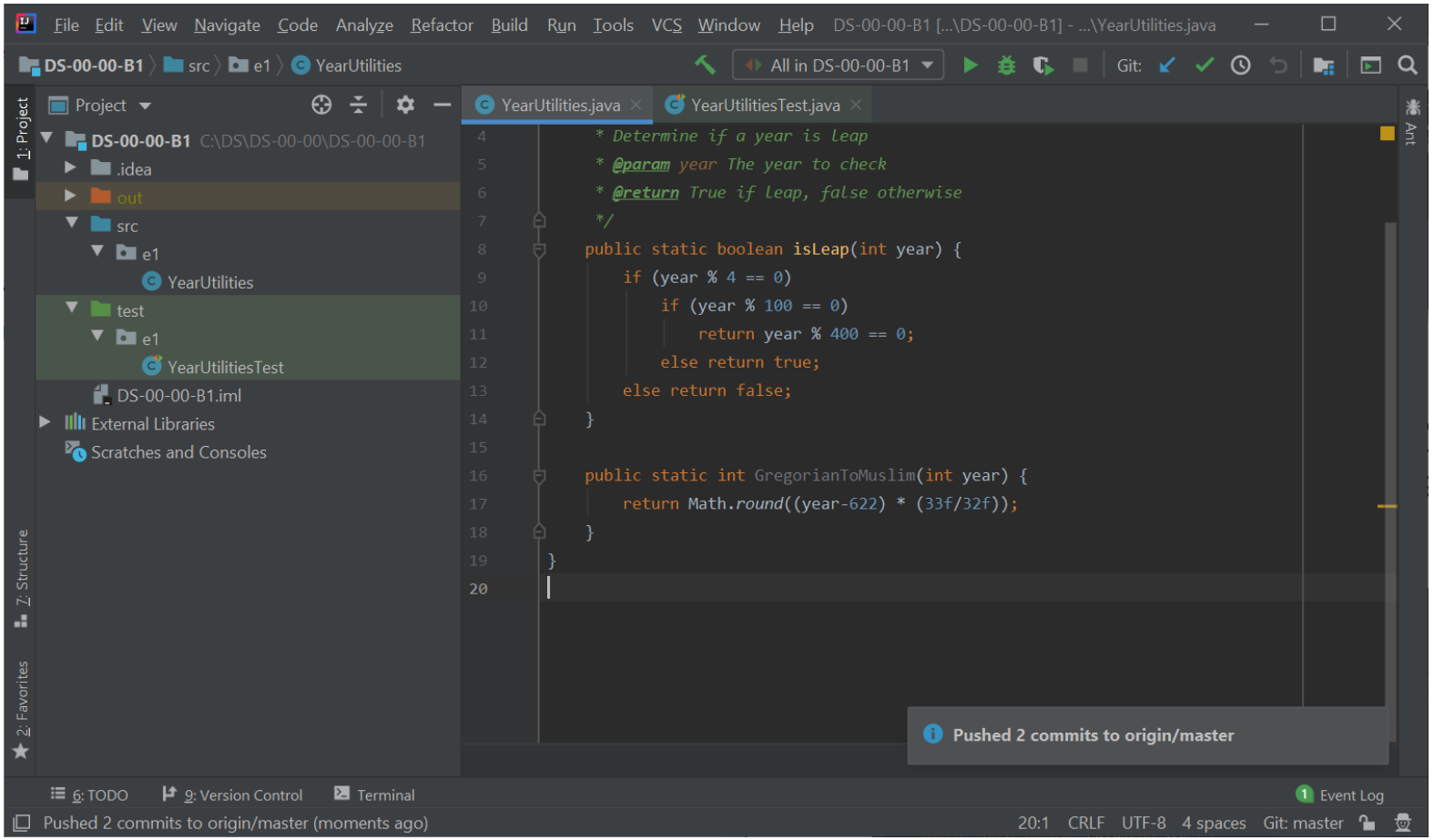
Resolving conflicts

- ...and see that the *commit* message is *Merge remote-tracking branch...*



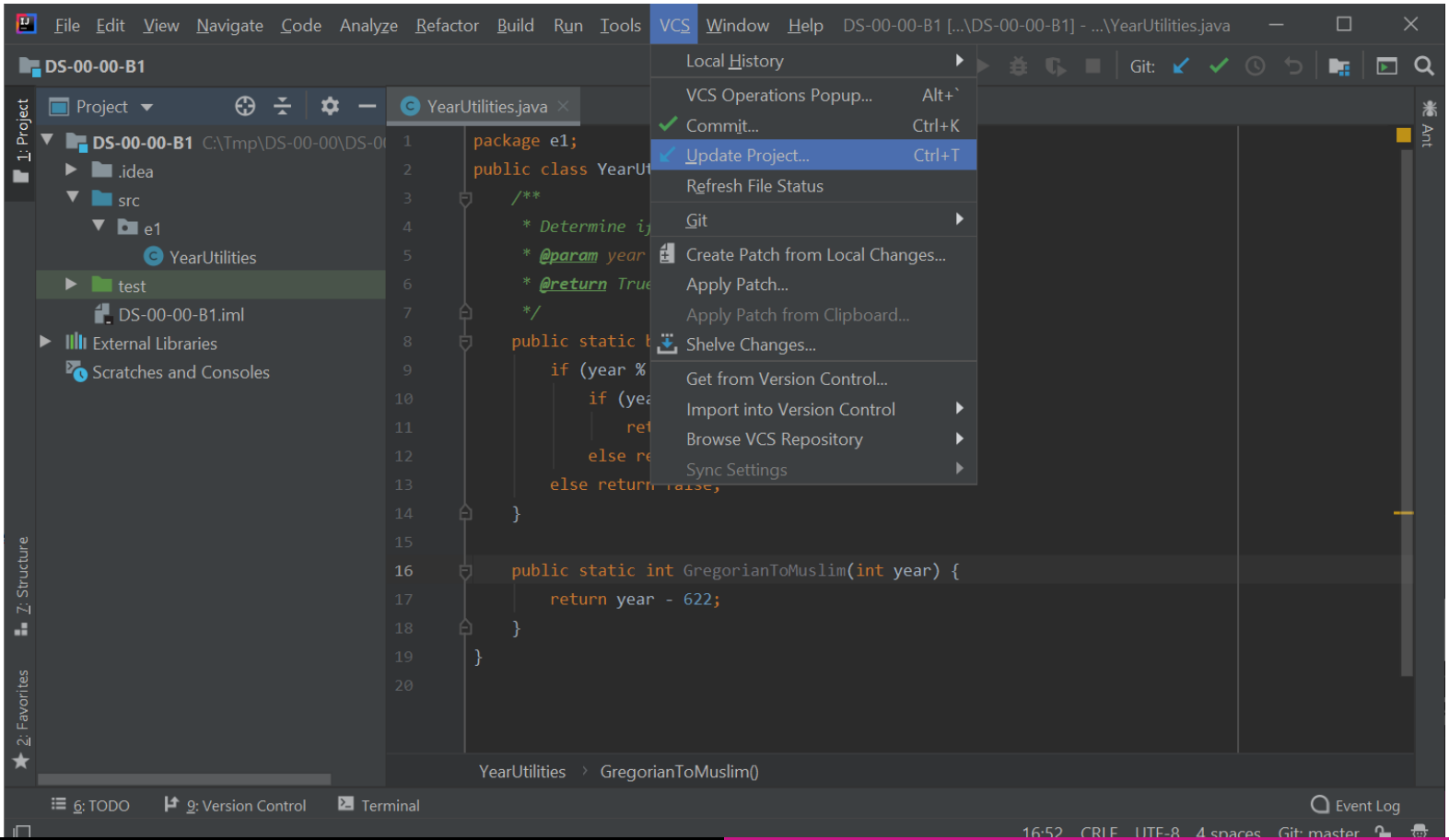
Resolving conflicts

- When it's done, a *pop-up* appears, showing that two *commit*s have been made – the original that we intended and the result of the *merge*.



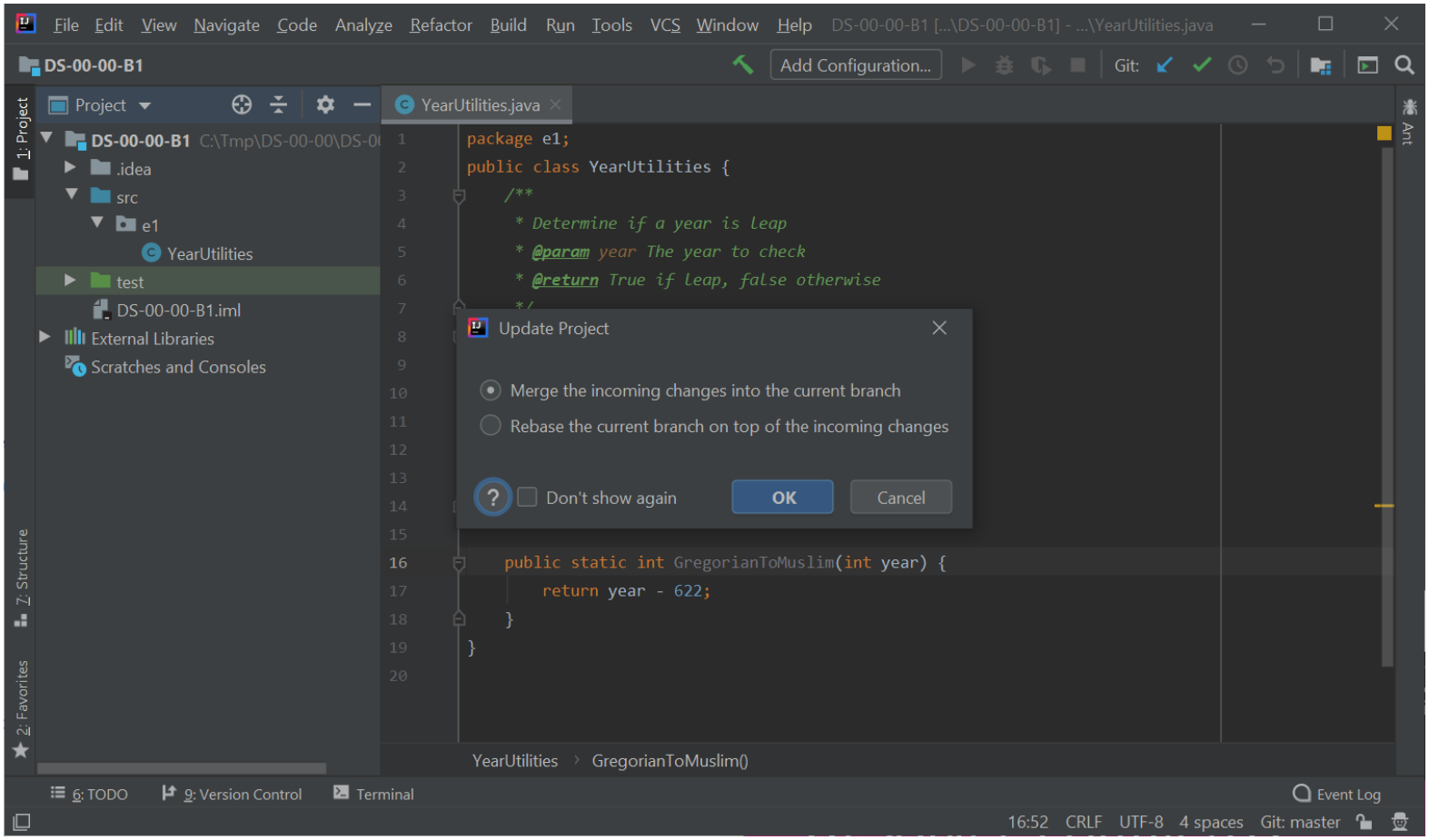
Resolving conflicts

- Now, the other member of the team, the one who had uploaded a simplified version of `GregorianToMuslim` is not up-to-date with the final version...
- ...and has to *Update Project* to download the remote changes.



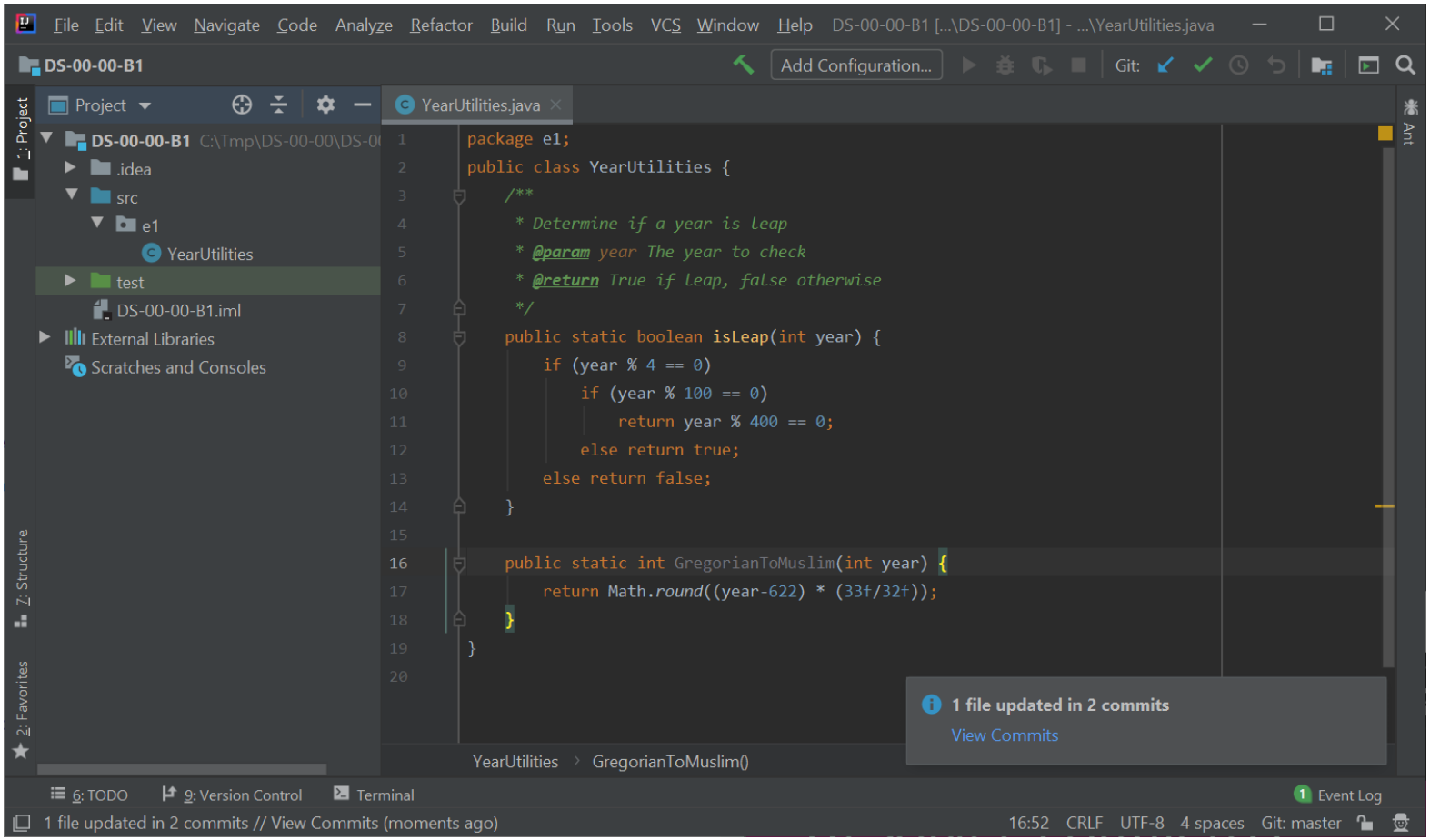
Resolving conflicts

- Again, it asks whether we want to *Merge* or *Rebase*. We choose *Merge*.



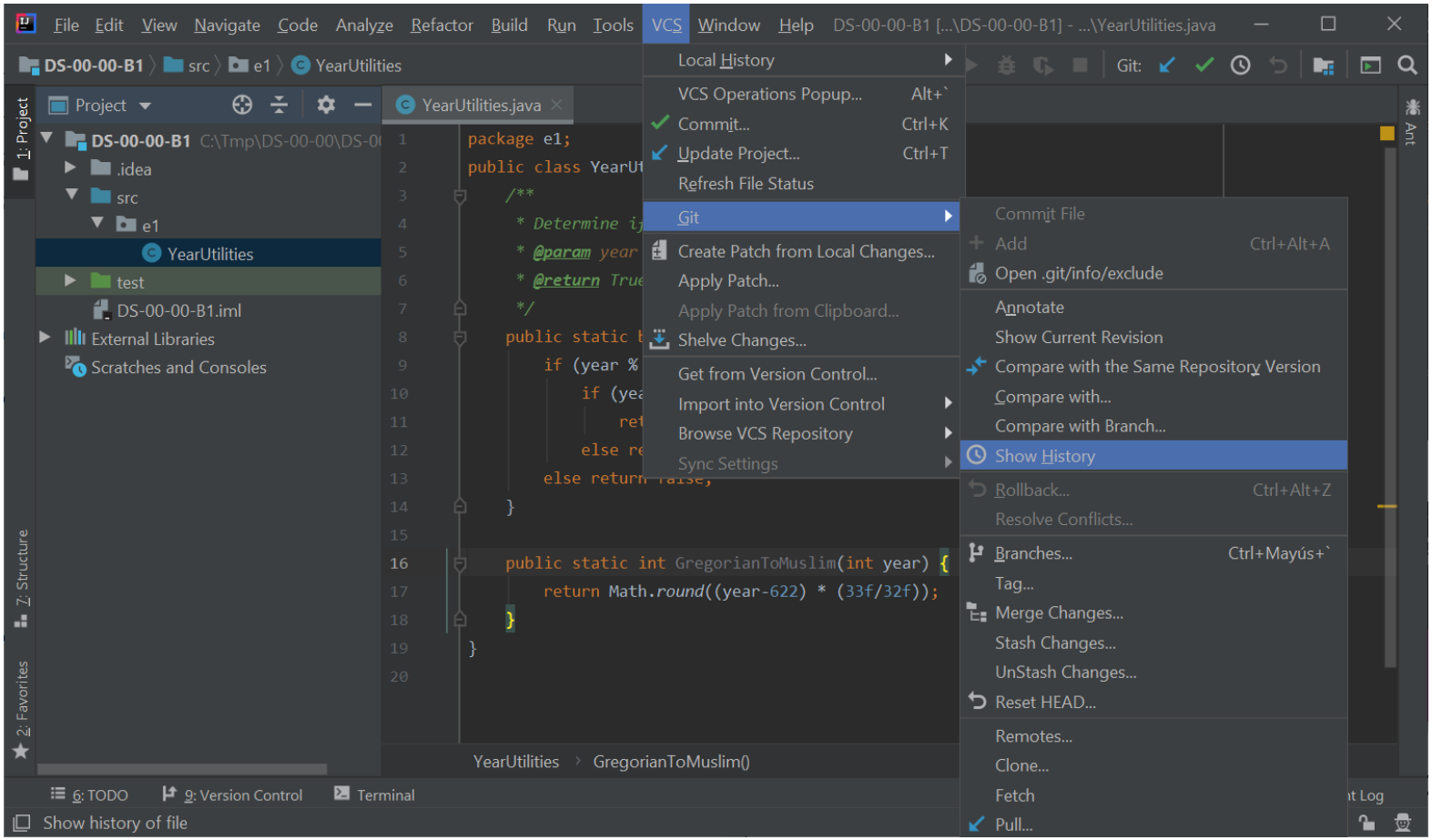
Resolving conflicts

- In this case there are no conflicts, and the remote and local versions are correctly synchronized.



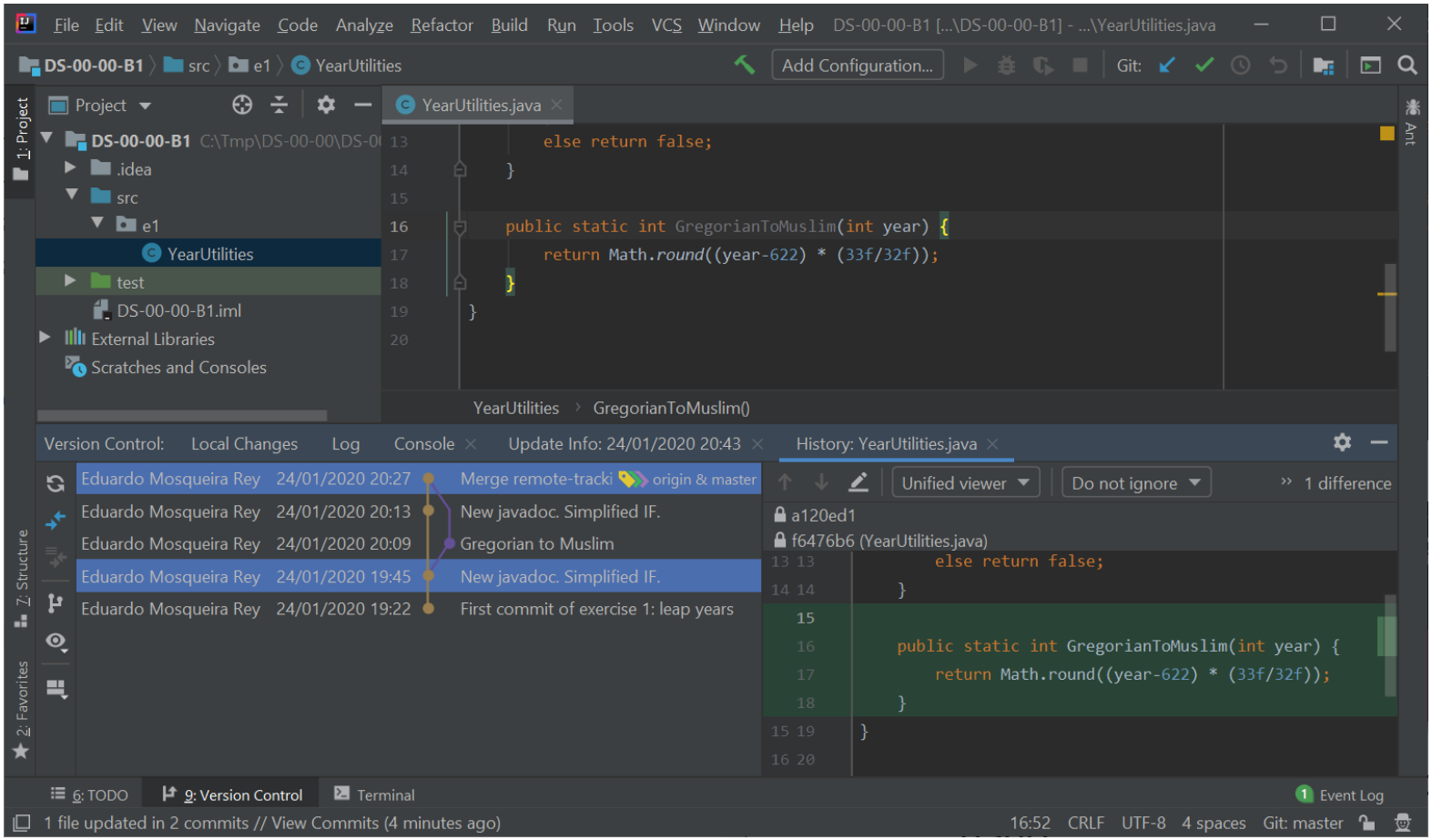
History

- At any point, we can review the history of any file...



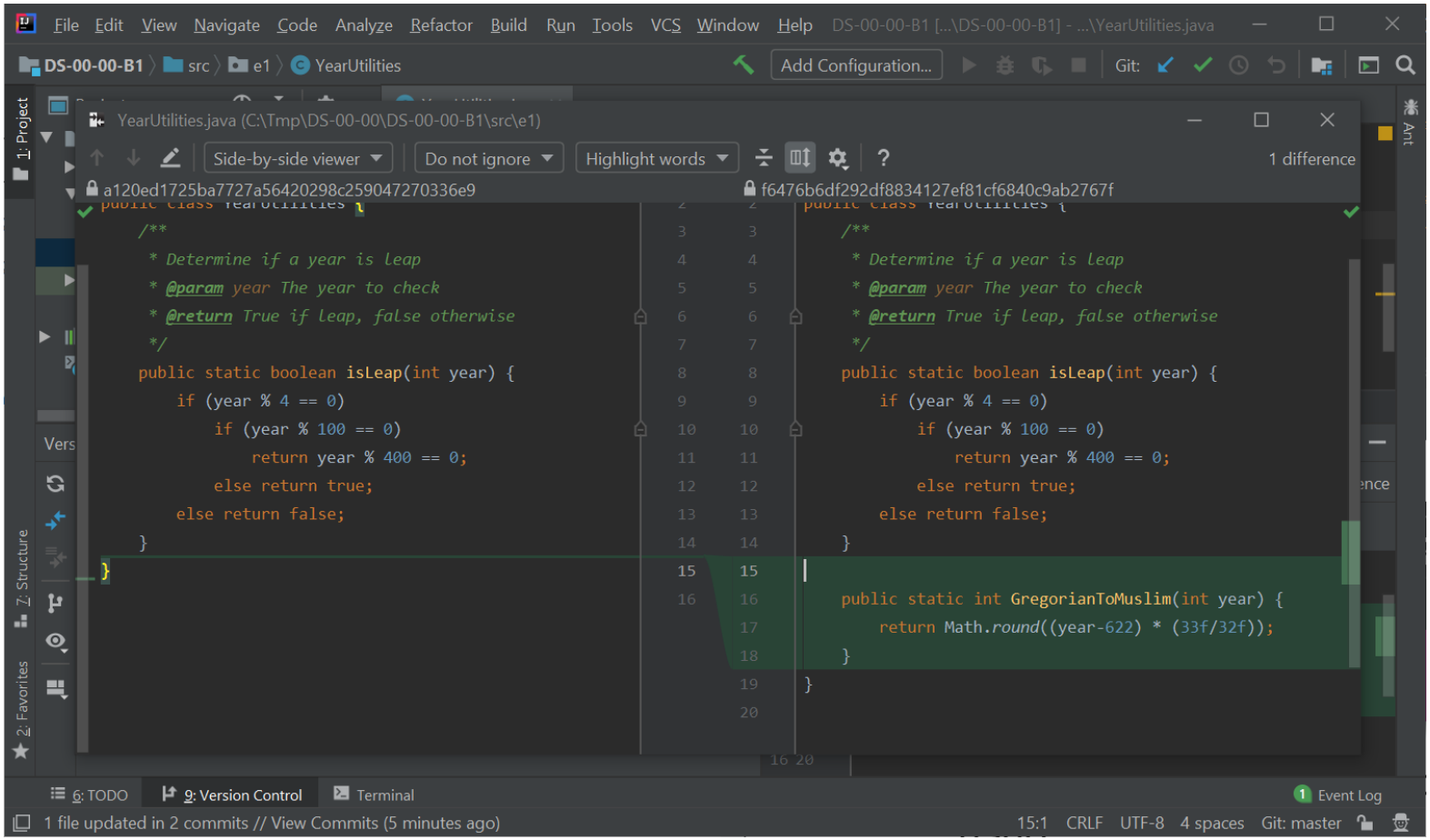
History

- ...see a list of all the changes and mark two version for comparison...



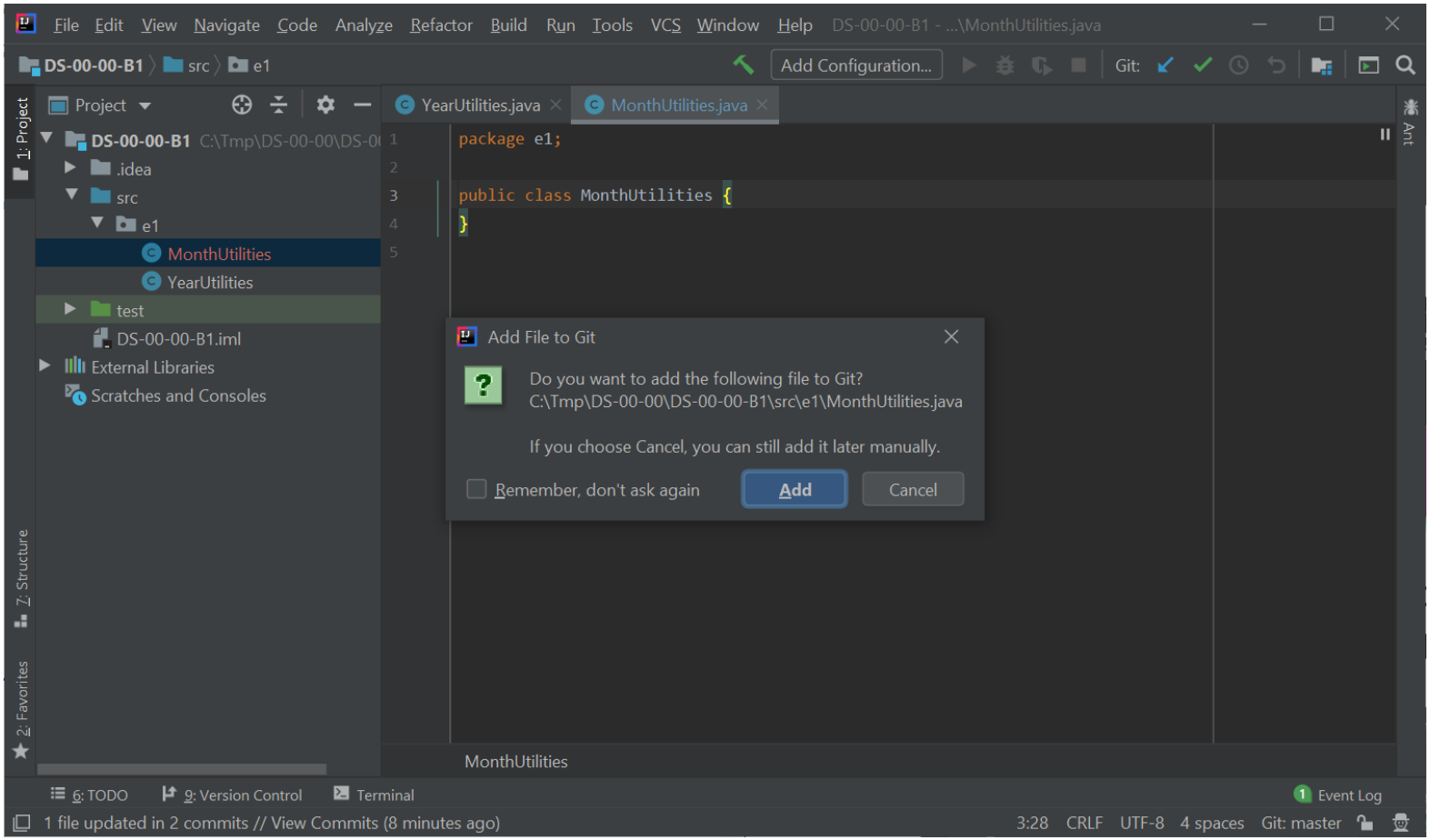
History

- ...and see their differences in a *diff* window (*Ctrl+D*).



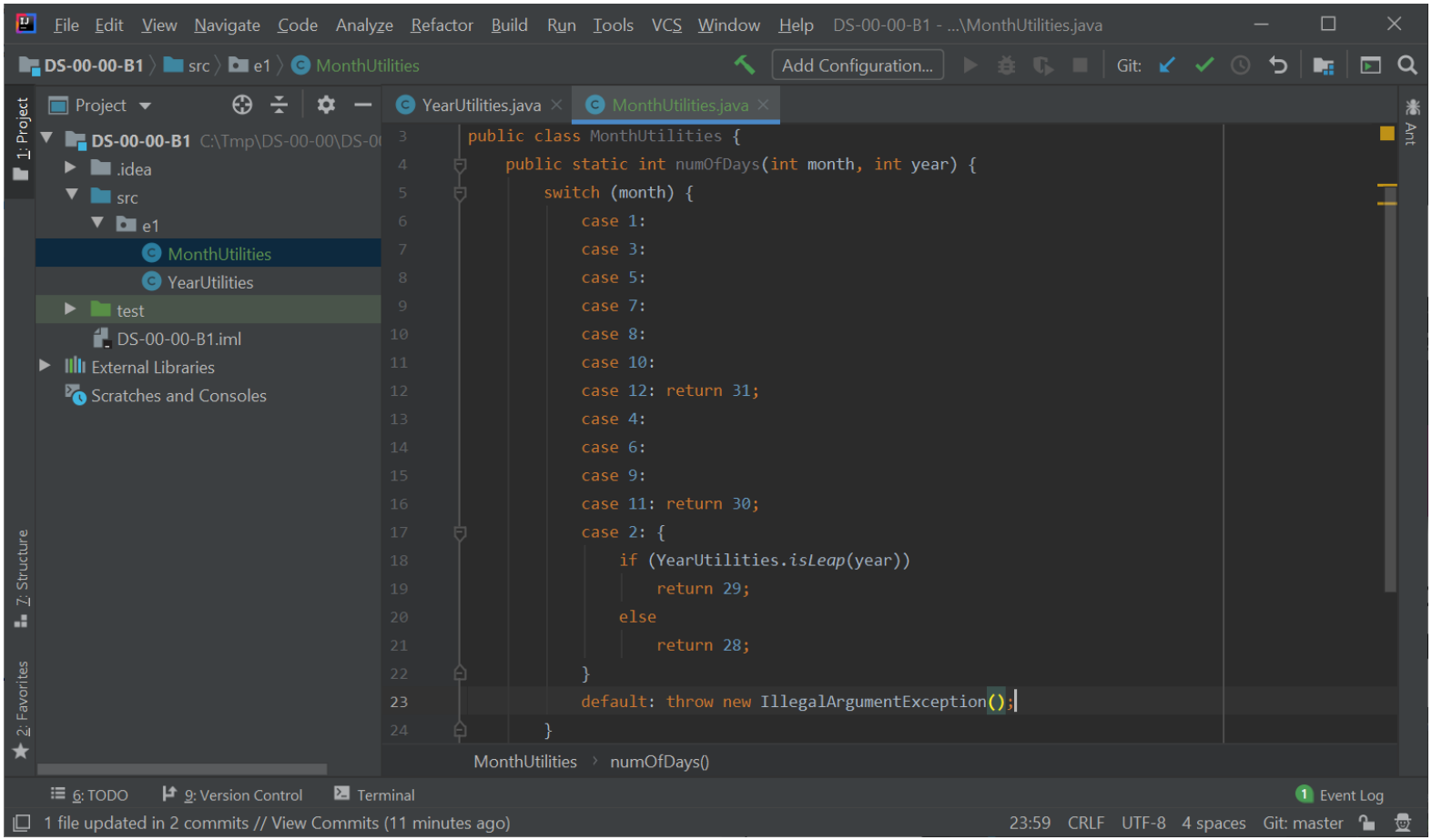
Adding new files to the repository

- Finally, every time a new class is created in our project, the version control system will ask if we want to include it.
- We must choose *Add* in order to include the new file.



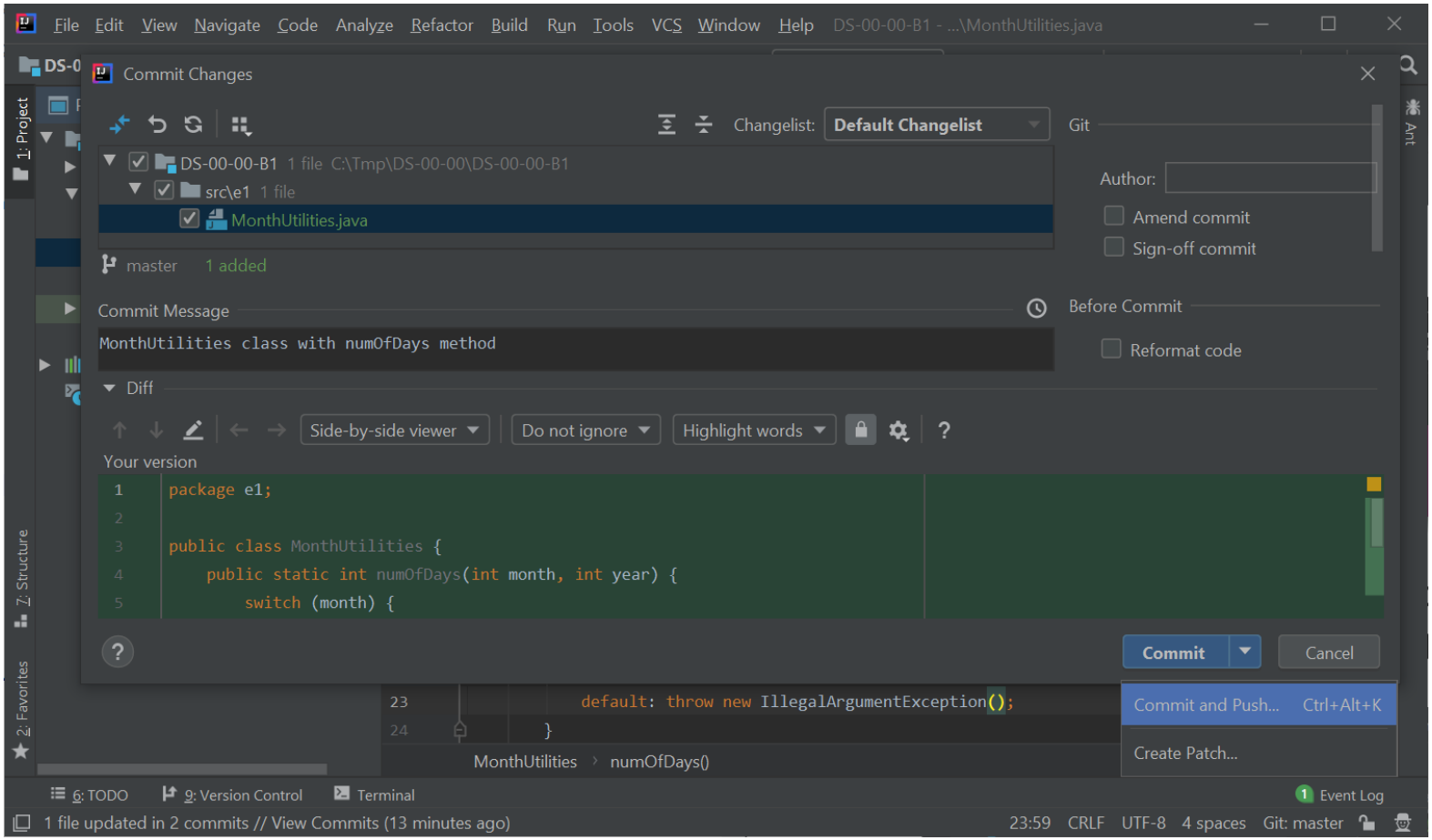
Adding new files to the repository

- When it is included, it will show up in green...



Adding new files to the repository

- ...and the code will be uploaded in the next *commit & push*.



Final recommendations

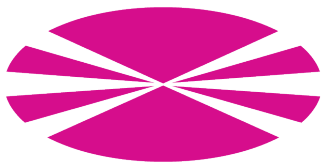
- **Do *commit & push* frequently**, do not wait for changes to pile up.
- **Do also *pull* frequently** to be always up-to-date.
- Try to give a purpose to the ***commit*s**. Do not *commit* random changes to multiple unrelated classes.
- **The *commit* message must be clear** and state the goal unambiguously. Long and detailed messages are preferable to simple "some changes" messages.
- **Arrange the responsibilities in order to minimize conflicts** when *merging*. If you do *pair programming*, no conflicts will occur, since both of you will be working on the same local copy.
- **Do not leave commits for the last moment**, computers sometimes fail at the worst possible moment.
- **Always check that changes have been successfully uploaded on the web page**. Even better, download a fresh copy to check it thoroughly.

Practice 4: Git

Diseño Software (614G01015)

Eduardo Mosqueira Rey (Coordinador)

Departamento de Ciencias de la Computación y Tecnologías de la Información
Facultad de Informática



UNIVERSIDADE DA CORUÑA