Source Code Management and Code Quality

Source Control

Every developer faces the question of where to store their source code. While day-to-day work happens on the developer's computer, long-term storage requires a different approach.

What if the computer crashes and all the code is lost? This problem and its solutions have existed for as long as software code has existed. The simplest solution is to store the code on a separate computer and manually back it up from time to time. This is what non-IT professionals do: work with their own documents — spreadsheets, text files, or presentations.

This raises the question of 'source of truth.' If more than one copy of a document exists, which one is considered the authoritative one? The most recent version? Or the version edited by a more senior employee? And if so, and the computer hosting the backups crashes, how can the most current versions of the documents be restored?

These are non-trivial issues, and each organization resolves them differently. Source code is also a document, but usually a large number of them. If there is more than one person on a team, in addition to the aforementioned issues, the question of concurrent access to the source code file or of sequential edits arises.

What does this mean? Let's imagine that Alice and Bob are working on their own tasks. And, without consulting each other, they both edit the same file. Then, they copy the changes to a shared folder where the entire team's source code is stored. Bob's changes could overwrite Alice's changes. And vice versa. It all depends on who overwrote the file first and who second.

If this seems like some kind of ancient nonsense, the author has seen a similar approach even today. Developers literally negotiated among themselves who would edit which file.  
Another example is a database. The best approach is to use scripts to update the database structure and data to the latest state. However, this practice is far from universal. Quite often, the source code for tables, stored procedures, and related objects is stored directly in the database. If the source code for a stored procedure needs to be updated from the DEV environment to the PROD environment, a developer manually copies it.

Version control systems (VCS) come to help. There are many different similar programs out there, but they all work roughly the same way. These are programs that have both a server and a client side.

One of the oldest programs still in use is Apache Subversion (SVN).

The server maintains a central source code repository. Each developer creates a local repository. The program copies the source code from the central repository to this local copy and tracks the differences.

Multiple developers can work on the code simultaneously and then merge it.

Some systems, such as Microsoft Visual SourceSafe, did not allow multiple people to work on a file simultaneously. The file was locked. While exclusive access to a resource is common in information systems, in VCS it is the exception, not the rule.

Git and GitHub

GitHub is widely used for storing and sharing source code. Initially, this project was developed as open source but is currently owned by Microsoft. GitHub integration is available in all modern IDEs, and many utilities make it easier to work with code. In this chapter, we will discuss the general concept of storing source code in GitHub and standard practices.

Note that "Git" and "GitHub" refer to different things. Git is a program for managing source code, while GitHub is a platform for hosting and storing source code.

Source code is stored in repositories. A repository can be thought of as the root folder of a project. Repositories can be public and private. You can define users who will have access to the repository, as well as the permissions of each of them. Usually, a repository has one or more administrators.

For a relatively simple project, a single repository is sufficient. However, for complex projects, especially those involving multiple teams, several repositories are often created.

**Creating a repository**

In this chapter, we will guide you through the process of creating a repository. You must have an active GitHub account.

Go to GitHub (working with a corporate account of Unicorn University is demonstrated here).

[pic1]

In the picture, the section with repositories is outlined in red. Let's go there.

[pic2]

Click the "New Repository" button on the right side of the page.

[pic3]

On this page, please specify the parameters for the repository being created. The repository must have a unique name within the account. Decide whether the repository should be private or public. If GitHub is used internally by an organization, then "public" means "visible within the organization." Such a repository is not available to all Internet users. However, if the repository is created outside the organization, then the public repository and its contents will be accessible to anyone.

The readme.md file is an optional but highly desirable part of the repository. This is a good practice. This file supports a special syntax for markup and contains information about the essence of the project, including the assembly and launch order. I recommend always having such a file in the project.

The .gitignore file is a must-have for every project. In this file, you can mark parts of the source code (individual files, paths, rules) that should not be included in the repository. If the technology stack is known in advance, you can select it from the drop-down list. However, this is not necessary; you can later manually add specific files and paths to .gitignore.

The license is an optional parameter. If you intend to create an open-source project, you can select the type of relationship with the user.

When all the parameters are set, click the "Create repository" button.

**Repository page**

Now, let's examine the elements present in each repository. For example, we will use the repository of a small game, Molehole, created in the Garmoska account.

[pic4]

Group 1 consists of the following functions:

* Code. This is the home page of the repository, an overview of the source code.
* Issues, Projects. If the tasks are stored in GitHub, a list of them will be available here. It is rarely used because other solutions are used to manage tasks.
* Pull Requests. Requests for changes to the code in the repository. We will talk about them below.
* Actions. Control center for automating actions: build, test, and deployment.
* Wiki. Encyclopedia about the project.
* Security. Security rules for this project.
* Insights. Statistics of changes in the code.
* Settings. Repository settings. This is where you can set the permissions of project participants and define the rules for working with branches as pull requests.

The most frequently used functions are Code, Pull Requests, and Settings.

Group 2 is used to manage code branches. We will talk about the branches below.

Group 3 displays the state of the source code for the selected branch in Group 2. In the picture, the “main” branch is active.

Below Group 3 is the contents of the readme.md file. This information greets a new user, so it is worth explaining in the first lines what kind of project this is, how to build and run it.

**How to work with source code**

GitHub stores source code in so-called branches. They can be thought of as isolated pockets. Changes in one pocket do not affect the contents of another pocket.

A repository always has at least one branch. By default, it is called "main". Developers create their branches from this branch, make changes to the code, and then merge their branch with the parent. While work is in progress in a separate branch, the contents of the parent branch remain unchanged.

Branches can be created from any branch, not necessarily only from "main". The number of branches is not limited. However, in practice, they try to keep it smaller; otherwise, it is difficult to merge the branches back later.

After merging a child branch with its parent, the child branch is often deleted to conserve space and prevent confusion, as it is no longer needed.

You can often see the following scheme for organizing branches in a repository. The "main" branch contains the latest stable version of the code. Let it not be the newest but the one that has been checked and approved. In short, at any time, the code from the "main" branch can be collected and given to the customer. The "develop" branch is created from the "main" branch. It is the root branch (or trunk) for all developers. It is from this, and not from the "main" branch, that all other branches for features and bug fixes are generated, and they are merged into it. If the project has automated testing and deployment tools (CI/CD), they are also aimed at the "develop" branch.

At the appointed time, changes from the "develop" branch are merged into the main branch. Moreover, the "develop" branch is not deleted but continues to serve as the trunk of the tree. Through this, the project acquires a new public version to show to the customer.

To switch branches and create new ones, the function shown below is used.

[pic5]

Item 1. Filter by branch name. If a branch with this name does not exist, then GitHub offers to create a new branch with the specified name. In this case, the code from the branch that is active will be copied there.

Item 2. List of branches with the default branch.

Item 3. This function can be used to create a local copy of the repository, which will be discussed below.

Item 4. If there are a lot of branches, you can see their full list by clicking the “View all branches” button. On this page, you can see the state of each branch: how far ahead or behind it is from the parent. Each branch can be deleted or restored.

Let's create a new branch. In the text field in Item 1, enter the name “demo-branch” and click “Create demo-branch from main”. The branch names in this button depend on the name of the new branch and the name of the parent branch.

The new branch has become active, as shown in the figure.

[pic6]

**Cloning a repository**

To make changes to the source code, you first need to create a copy of the repository on your computer. This operation is called “cloning”. Its name reflects the essence of the process: a local copy of the branch is created, which is located on the developer's computer, and then the online version of the branch and the local version are synchronized with each other.

For all the following operations, we will use the official GitHub Desktop program. Additionally, there are numerous free and commercial programs with similar functionality. Plus, Git supports operations through the command line.

GitHub Desktop looks like this:

[pic8]

First, you need to establish a connection between the program and your GitHub account. Click File> Options> Account and follow the instructions.

Then click File> Clone repository. A menu for selecting a repository will open. If it belongs to you, you can find it on the “GitHub.com” tab. Or go to the “URL” tab. This is a place where you need to specify a special path to the repository and click the button to initiate the cloning process.

[pic9]

When cloning is successful, GitHub Desktop displays the repository name and the active branch.

[pic10]

The list of files on the left is empty, not because there are no files, but because there are no local changes in them yet. The files are located in the directory on the disk that you specified in the cloning settings.

The picture above shows the menu for selecting the active branch. When the repository is cloned, its active branch is the default branch. We need to switch it to “demo-branch”. Find it in the list and click on it. If the branch is not displayed, then click the “Fetch origin” button. This button starts a check for changes in the online repository. If there are any, the program will offer to download them. It is a good practice to click this button every time before starting work in case someone else has updated the files.

I added a new file to demonstrate the changes.

[pic11]

It is displayed in the list of modified files. In the right panel, you can see its contents. These changes are only on the local computer and are not visible in the online repository.

To move files between the online repository and the local computer, use the pull (online > local) and push (local > online) commands. Each push operation must have a comment. GitHub Desktop provides a text field for this. You need to select the files that should be transferred to the online repository. The remaining changes will only be applied to the local computer. You can also add a file or an entire directory to .gitignore so that GitHub Desktop does not display them and skips them when transferring to the online repository.

Once the list of files to transfer is defined and the comment is written, click the “Commit” button. The program will create a local commit, that is, a package with changes. It has not yet been transferred to the server.

In the upper right corner, the inscription on the button has changed to “Push origin (1)”. This means that there is one commit that can be sent to the online repository. You can accumulate several commits and then send them online.

Once the push is complete, we can see the changes in the online repository.

[pic12]

This means that the changes made on the local computer have been synchronized with the online branch. If the goal of the work for which the branch was created has been achieved, then you need to prepare a pull request. Usually, developers make commits more often to record their progress in the online repository. Please note that changes in a specific branch do not affect "develop" and not "main" until they are merged into these branches.

To make a pull request, you need to click the “Compare & pull request” button. Alternatively, you can go to the list of all branches (see above) and click the “New pull request” button, as shown in the picture.

[pic13]

Then, the pull request settings page will appear. Be especially careful when specifying the recipient branch. You can accidentally merge changes into the wrong branch. Keep in mind that in the repository Settings, you can define rules to protect branches from unintentional damage.

[pic14]

**Validation of Pull requests**

The organization may have special rules regarding merging pull requests. Typically, they are concerned:

* Passing automated tests.
* Correct code style. This can be checked manually or automatically.
* Comments from reviewers. Experienced team members review each other's code for potential errors and provide constructive comments.

The repository can be configured not to merge branches until all the above-mentioned comments are fixed. Often, fixing them takes a significant amount of time. On the one hand, this creates an additional burden on team members. On the other hand, it helps to avoid introducing problems and spreading them across different branches.

[pic15]

When there are no more comments on the pull request, the code of the child and parent branches is merged. Sometimes, the parent branch already contains changes that came there, for example, from its parent branch. And such changes can conflict. If this happens, then the code is manually or semi-automatically brought to an equilibrium state step by step. If the project has automatic tests, they are re-run. To reduce overhead, it's best to merge code into child branches as often as possible. This is covered in the final part of this section.

Useful Practices

* Establish clear rules within the team so that all participants share a common understanding of source code standards and know what is expected from their pull requests.
* It is better to configure the rules in Settings so that changes can only be uploaded to the “main” branch from the “develop” branch. For the “develop” branch, enable a rule that prevents direct code changes, allowing only pull requests.
* Code review for each pull request is good practice. Let it require additional team effort. The review requires tact from the reviewer to convey their comments in a polite manner. Remember that the goal of the review is to improve the quality of the product and share experiences, not to determine who the most knowledgeable specialist is.
* Create a separate branch for each feature. Do as many commits & pushes as necessary. Remember that local changes to the source code can be lost if the computer crashes.
* Do not share source code files in the chat or otherwise. Invite a colleague to clone your code branch and collaborate on it.
* Check for changes in “develop” as often as possible. Pull them into your working branch to reduce future issues. If you don't update the branch for a long time, the code will diverge significantly, and merging it will be a difficult task.
* Read the documentation and learn more about Git & GitHub.