Contents

Please keep in mind that the remaining parts of this documentation are just for help to fulfill the requirements stated above, and they may not contain all the necessary information. You can find further info on the internet (git/stylecop/doxygen official help pages, google). Some links can be found in the various parts of this documentation.

GIT QUICKSTART

Git is a distributed version control/source control system that allows us to centrally manage our source code and the changes to this source code (record everything: which file was modified when and how and by whom). This is achieved by not storing every state (every version) of every file, but instead only the difference/increment between file revisions.

The distributed nature of Git is shown if we compare it with other “traditional” source control systems (SVN, CVS): in Git, the changes of the source code are not only stored in a central storage, but also in a local storage as well. According to this, in Git, we have to differentiate between the *local repository* and the *remote repository*. The third storage is the local current state of the development files, this is the *working copy*.

The basic Git operations are listed in this table:

ADD	Add files from the woking copy to the source control / add modified files into the list of files affected by the next commit
COMMIT	Store the modifications (content changes, ADD, file deletion) of the working copy into the local repository
PUSH	Upload the commits of the local repository into the remote repository
FETCH	Download the commits of the remote repository into the local repository, without changing the working copy
MERGE	Merging branches (e.g. after a FETCH, there are two branches in the local repository: our own development branch and the branch downloaded from the remote repository). If there is contradiction (CONFLICT) between the branches (different modifications to the same file), then we have to fix these conflicts manually. In addition to this, the BRANCH / CHECKOUT commands can be used to create new branches, every branch is a separated version of the code, and also very often every new feature is implemented in its own branch (see "A successful git branching model", nvie.com; THIS IS NOT REQUIRED FOR US)
PULL	FETCH + MERGE

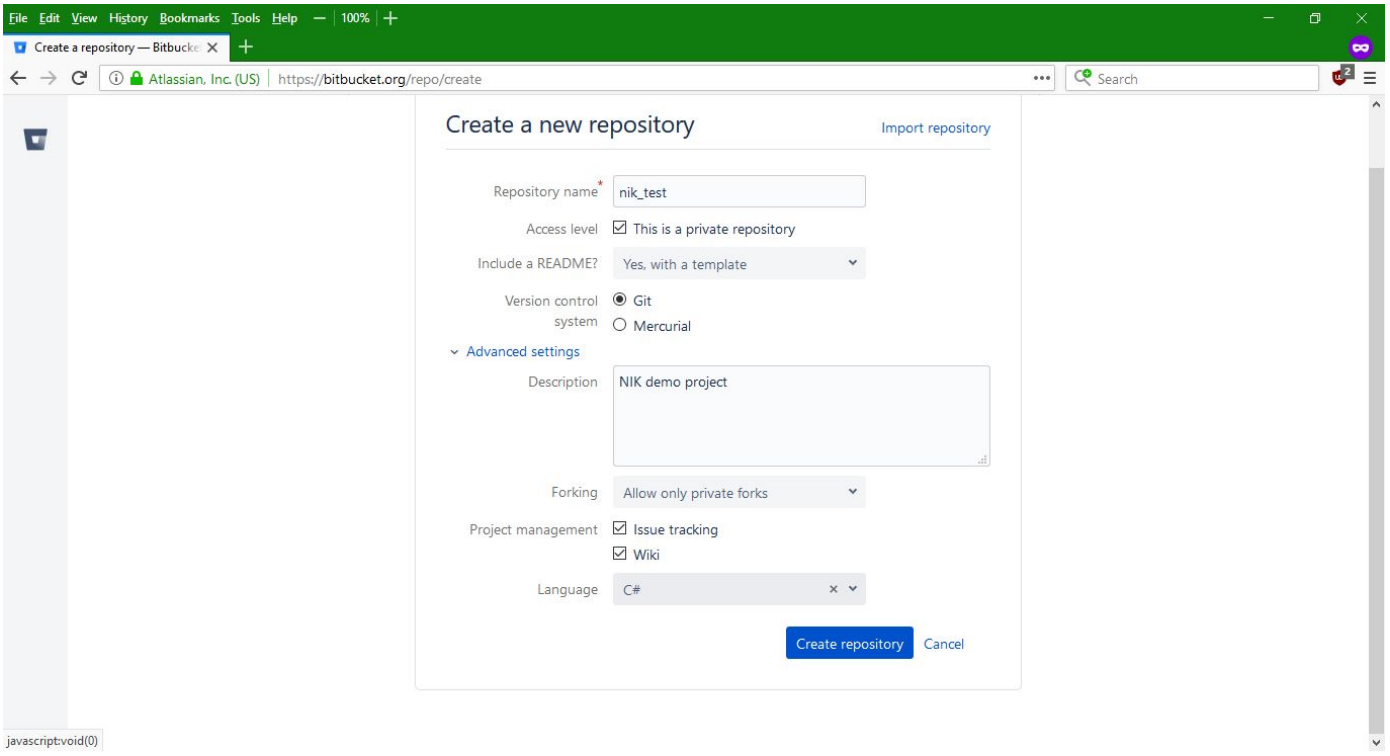
A typical development cycle looks like the following:

1. At the start of the day, the developer downloads the earlier modifications of the remote repository using a PULL command; into the local repository
2. The occasional conflicts have to be solved manually

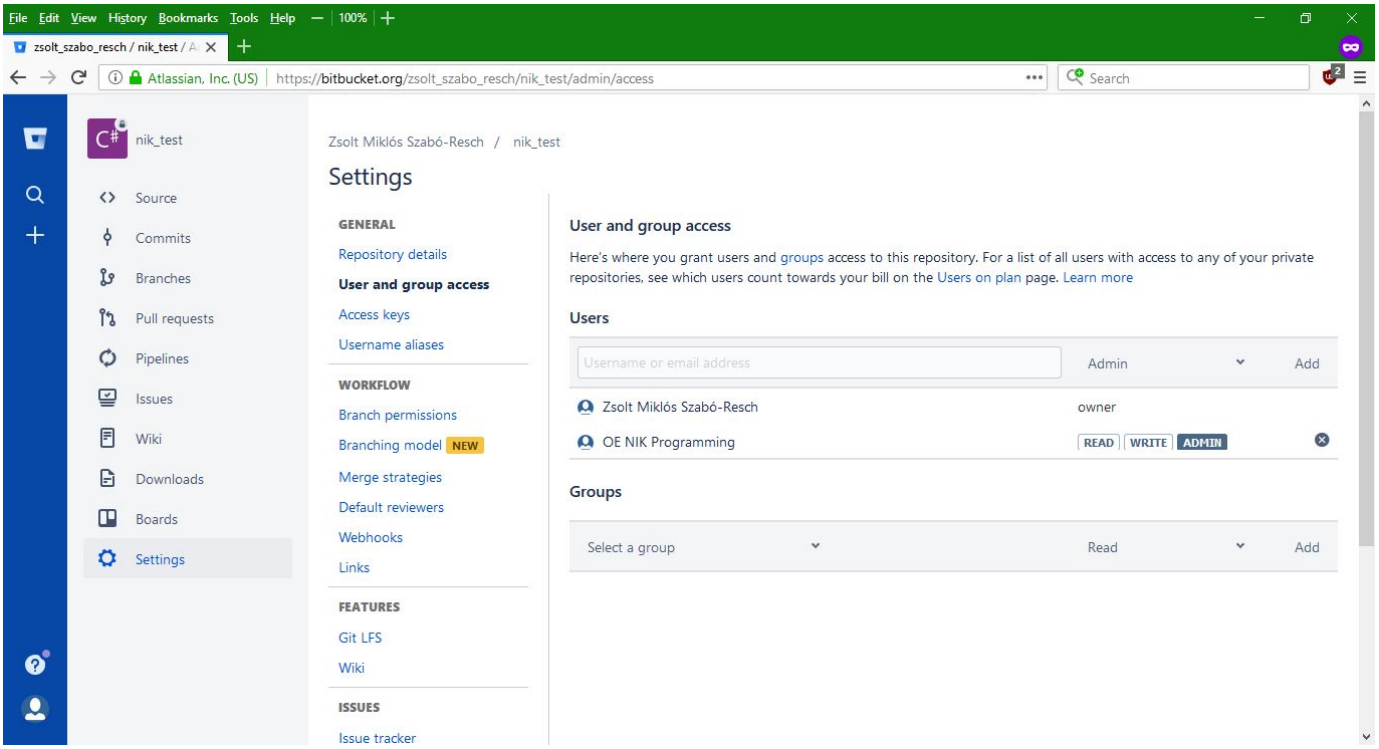
3. Work starts on a new feature / bugfix (the new feature usually goes with the creation of a new branch)
4. At the end of a single work-process the modifications are stored with a COMMIT command
   (this can even be one commit per file, but it can be 10 files, depending on the definition of „one work-item". Basically the principle is to work with multiple small commits, and descriptive commit messages. The basic rule is that one commit fixes only one thing / adds only one feature to the code: „If you have to put the word "and" or "also" in your summary, you need to split it up", this is the golden rule).
5. At the end of the day / at the end of the feature or bugfix, once again PULL, and the conflicts have to be solved manually
6. PUSH, and we can move on to the next ticket/work-item/bugfix/feature ...

Creation of the bitbucket repo

1. The GIT serverside system, that we will use during the semester is: bitbucket.org. For our aims, the github.com would be good as well, but there all projects are public, only the paying plans allow private projects. Also, any private GitLab installation could be usable as well, but for the univeristy project works, you have to use the bitbucket website.
2. Bitbucket registration, activation, login, then we can create the new repository. Issue tracking is advised, but not obligatory. Include a Readme: YES, so that the GIT repository itself is also created.

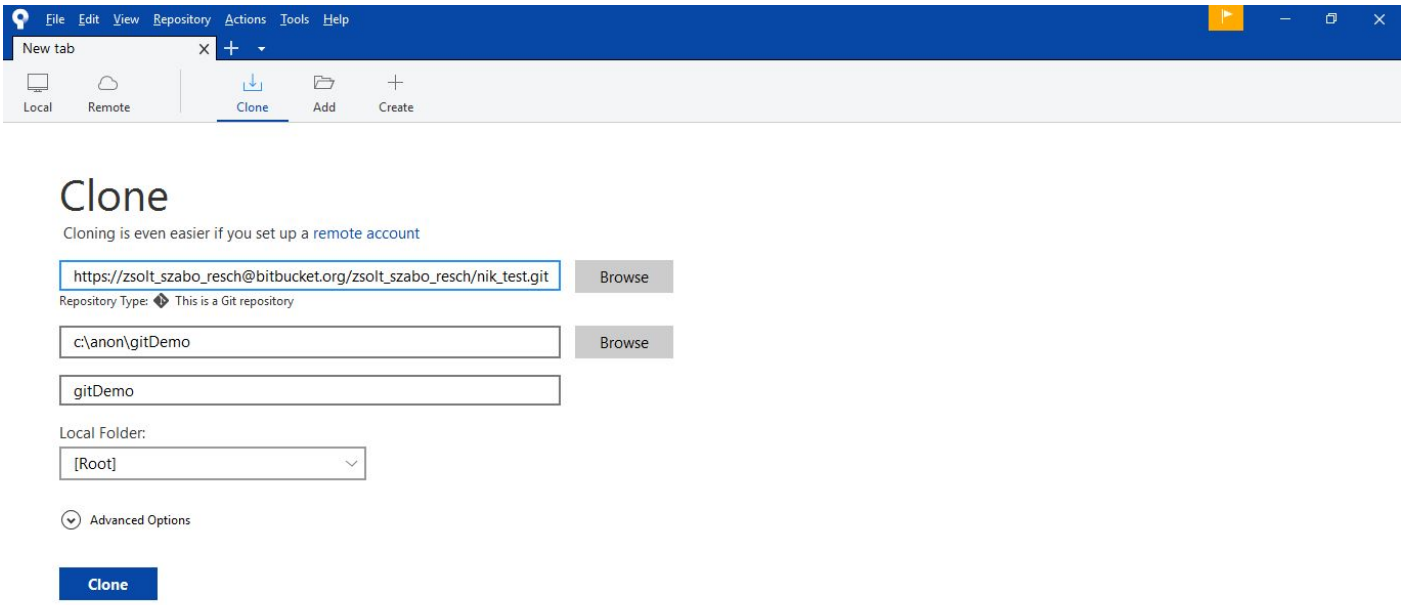


3. After creation, we get to the „Source” part of the project site, in the top right corner there is a “Clone” button that shows the repo’s git address: https://zsolt_szabo_resch@bitbucket.org/zsolt_szabo_resch/nik_test.git
4. In the left-hand side, click on the Settings menuitem, then „User and Group Access” should be used to add permissions to other users.
   It is obligatory to give ADMIN persmissions for the **oe_nik_prog** user (e-mail address: **nikprog@iar.nik.uni-obuda.hu**).

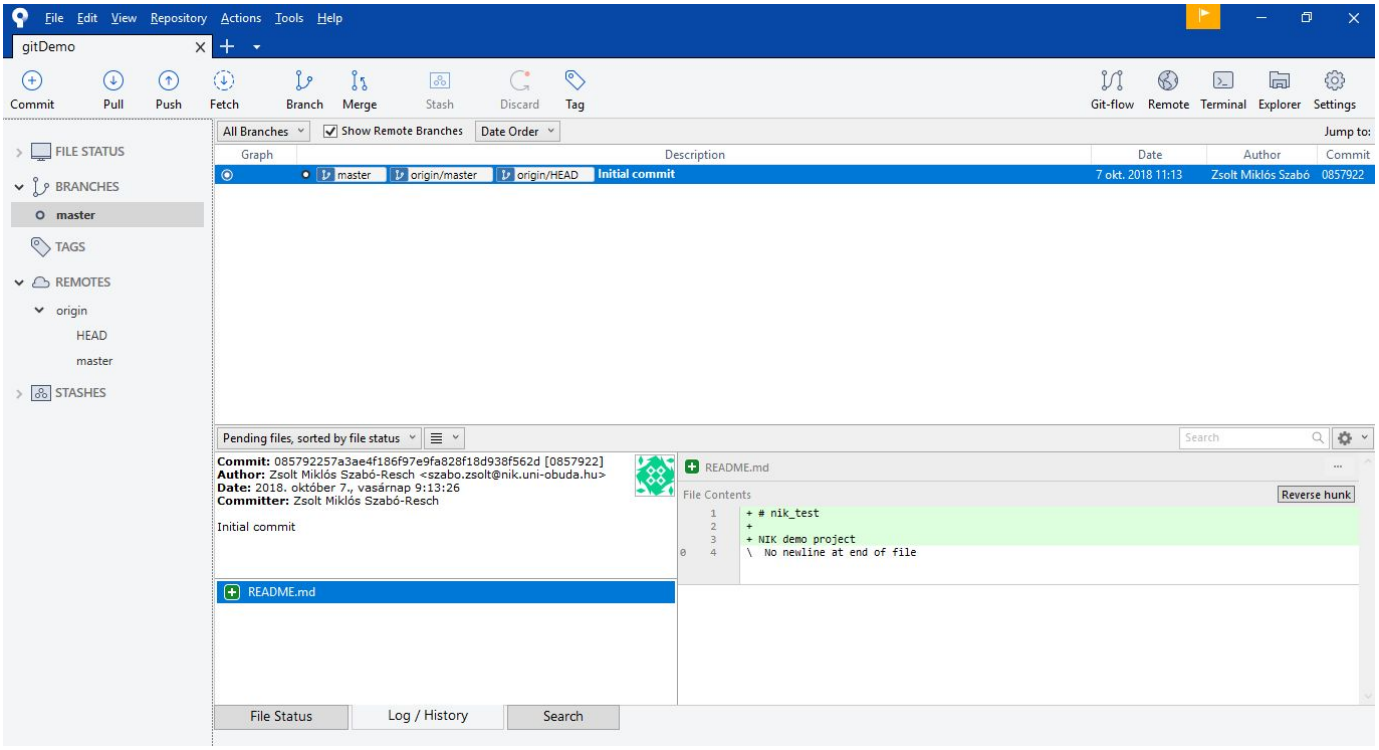


Using Git

1. This section shows the GIT usage using the SourceTree GUI client; naturally other GIT clients are allowed too (TortoiseGit, Git commandline). GIT clients integrated into the IDE (NetBeans, Visual Studio) is not advised, because with the Prog3 subject, the git repository will contain TWO sub-directories, and each IDE will only know about one of them. Of course both could be configured to handle this, but rather it is suggested that an external (and better) GIT client is used instead.
2. Download SourceTree (https://www.sourcetreeapp.com/), install (it will ask for the BitBucket user), then using the Clone button, copy the git repository's address. In the pop-up, enter your BitBucket login data, so the text "This is a git repository" is shown, and the lower CLONE button is enabled.

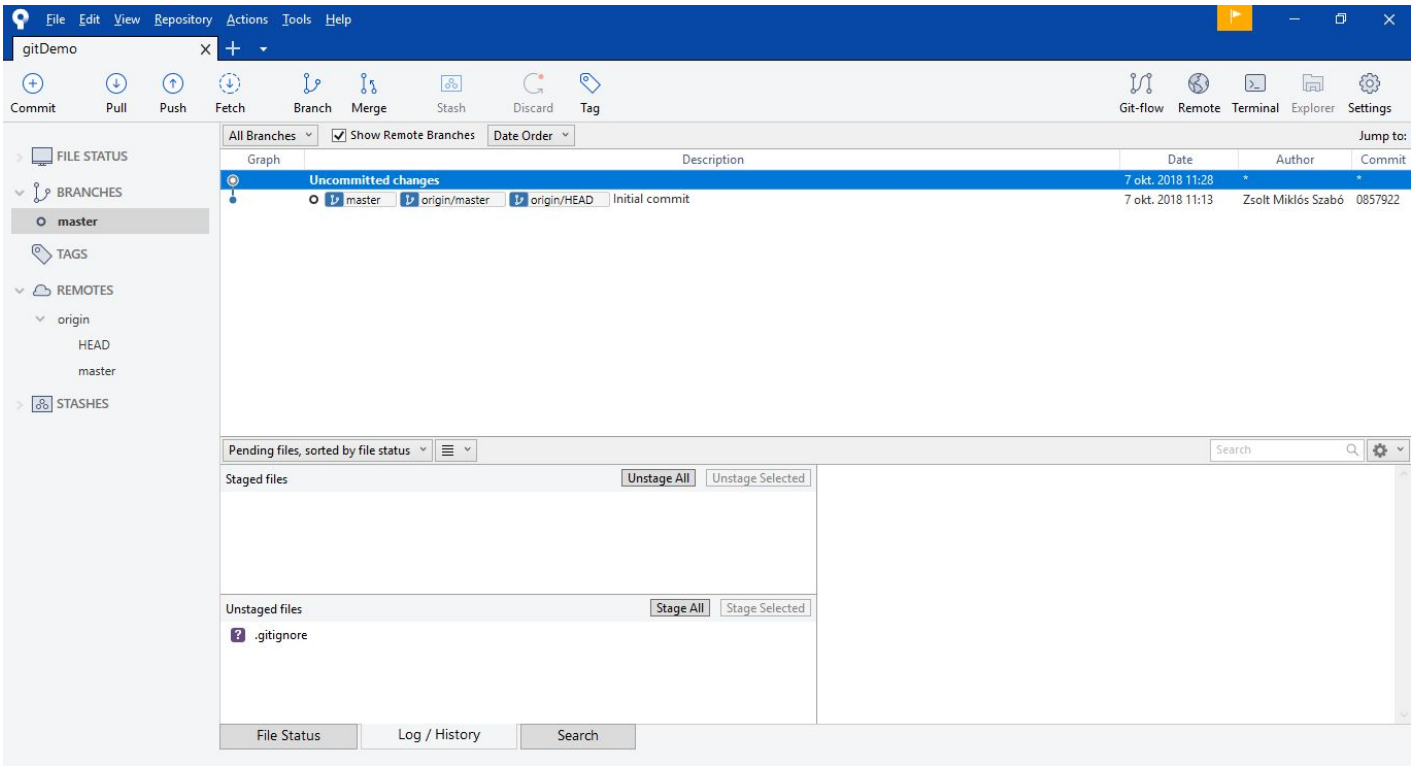


3. In the following screen, on the left side we can see the local and the remote BRANCHes (initially there is a remote origin/master branch, and a locel master branch associated with it). In the middle we can see the list of modifications (COMMITS), down there is the staging area (list of modifications inside a commit), and the button bar above shows the possible operations:
    a. Commit = Staging management (selecting the active modifications that will go into the next commit), and actual commit (saving the modifications into the local branch).
    b. Pull = Download the commits from the remote branch, then merge the commits into the local branch
    c. Push = Upload the commits of the local branch into the remote branch
    d. Fetch = Download the commits from the remote branch, no merge
    e. Branch = Create a new code branch
    f. Merge = Merge a different code branch into the current one



4. From the https://github.com/github/gitignore address, we should download the VisualStudio.gitignore and the Java.gitignore files. Using our favourite text editor, we should create a **.gitignore** file (there is nothing before the dot!), and copypaste the contents of **both** aforementioned files nicely below each other.
   IMPORTANT: In the .gitignore file, comment out the lines refering to MDF/LDF files (the database is also part of our repository!)

5. SourceTree sees the modifications (as “Uncommitted changes”).



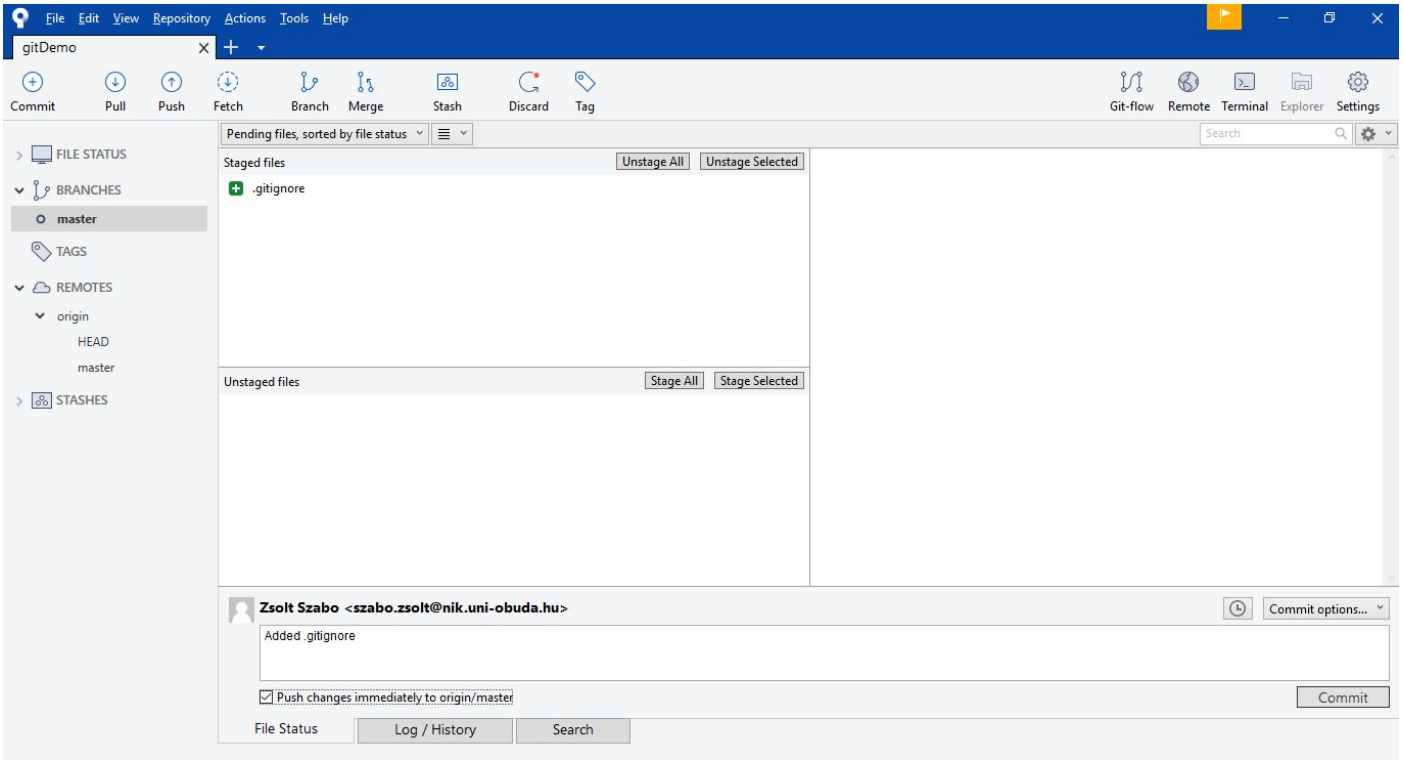
6. The staging area could be handled on this screen, but after clicking the COMMIT button it is more clear that we are actually selecting the modifications that will go into the next commit.

   Stage All/Unstage All = all files;
   Stage selected/Unstage selected = selected files;
   Stage hunk/Unstage hunk = modified section inside a file.

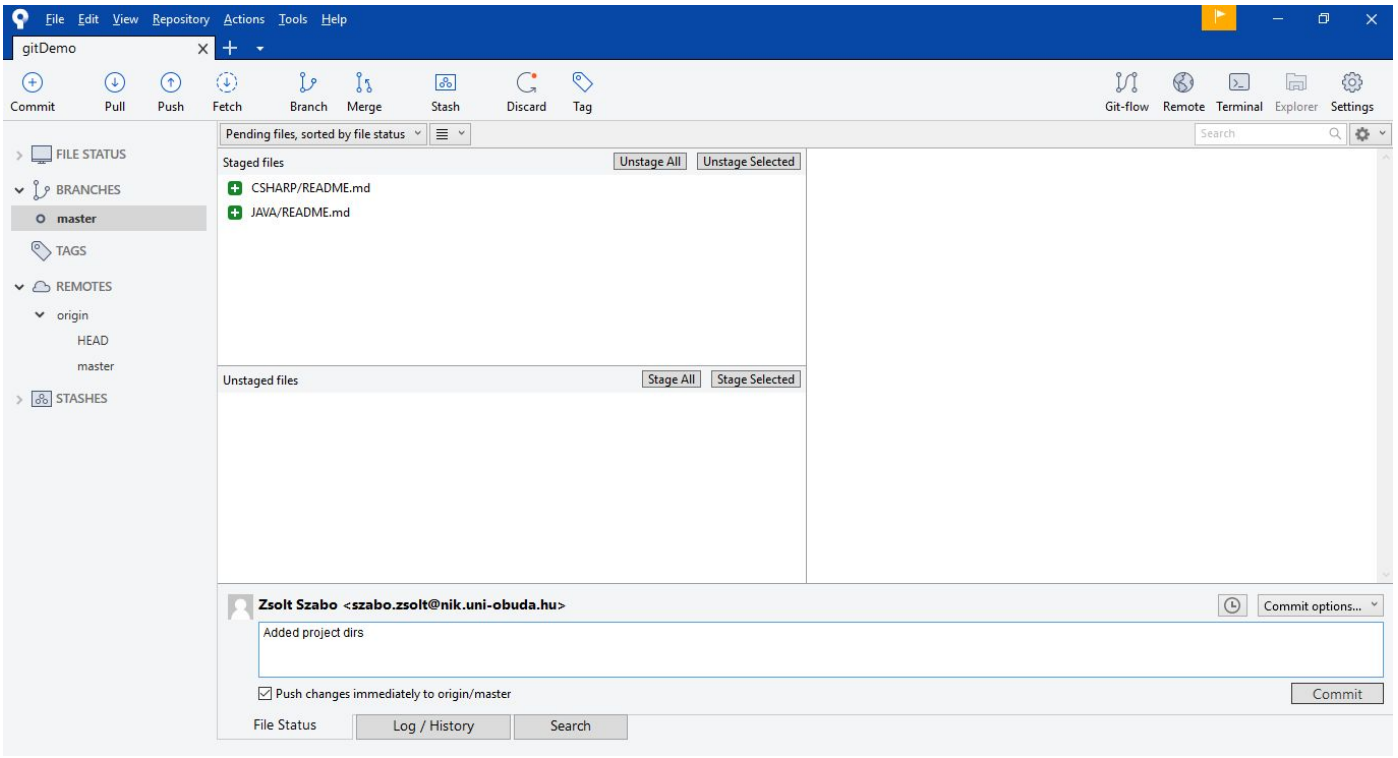
   Down there we can set the commit message, which should always be English! If we have to use the AND/ALSO words, then split the modifications into multiple commits!

   Checking the "Push changes immediately..." is advised, if we don't check the checkbox, then after the commit we have to do a manual PUSH.

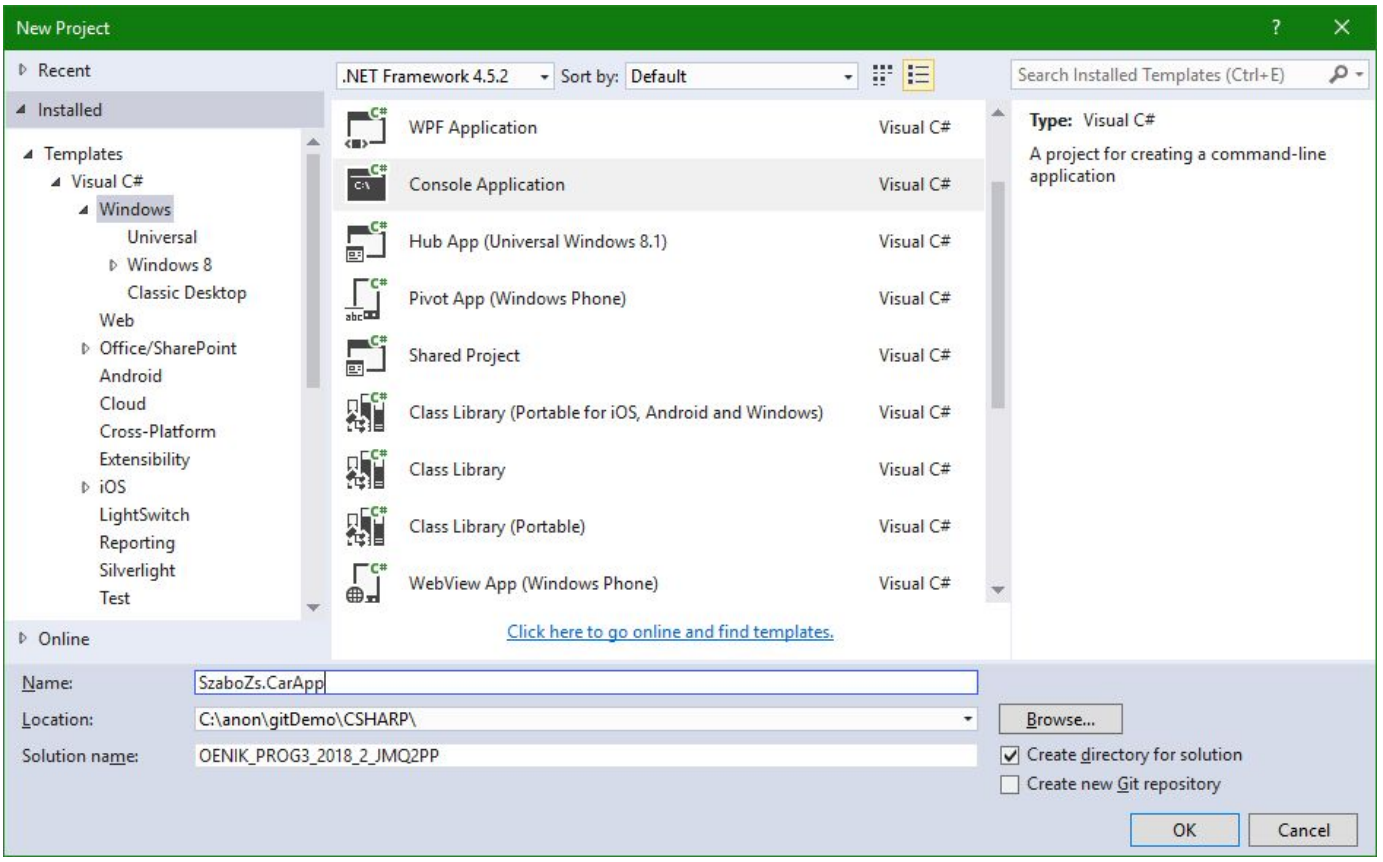
   After this, we can click on the COMMIT button in the lower-right corner.



7. [**ONLY WITH PROG3**] Change ot a file manager, and inside the GIT repository create a CSHARP and a JAVA folder. Both folders should have a suitable readme.md file. After this, we follow the usual sequence (that we should remember): one click on the COMMIT button above, then STAGE ALL, then one click on the lower COMMIT buton (with “Push immediately” still checked).



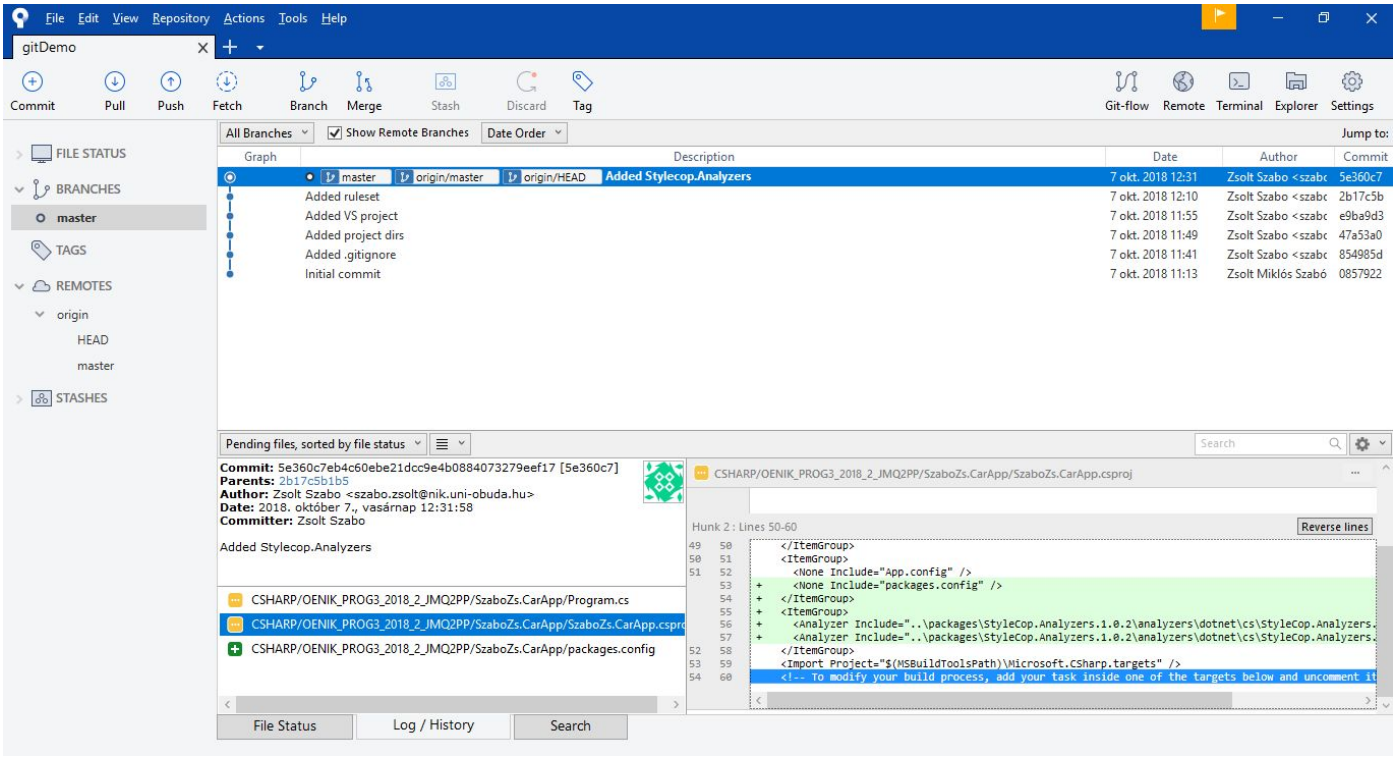
8. From the VS, we save a new solution with the first new project inside. The name of the solution is **obligatory: OENIK_SUBJECT_YEAR_SEMESTER_NEPTUN**, where SUBJECT must be PROG3 or PROG4; YEAR must be the year when the semester starts; SEMESTER means 1 = spring, 2 = autumn. After this, you must put the neptun code / neptun codes, multiple neptun codes must be separated with underscores.



9. Then in SourceTree: COMMIT, STAGE ALL, COMMIT.

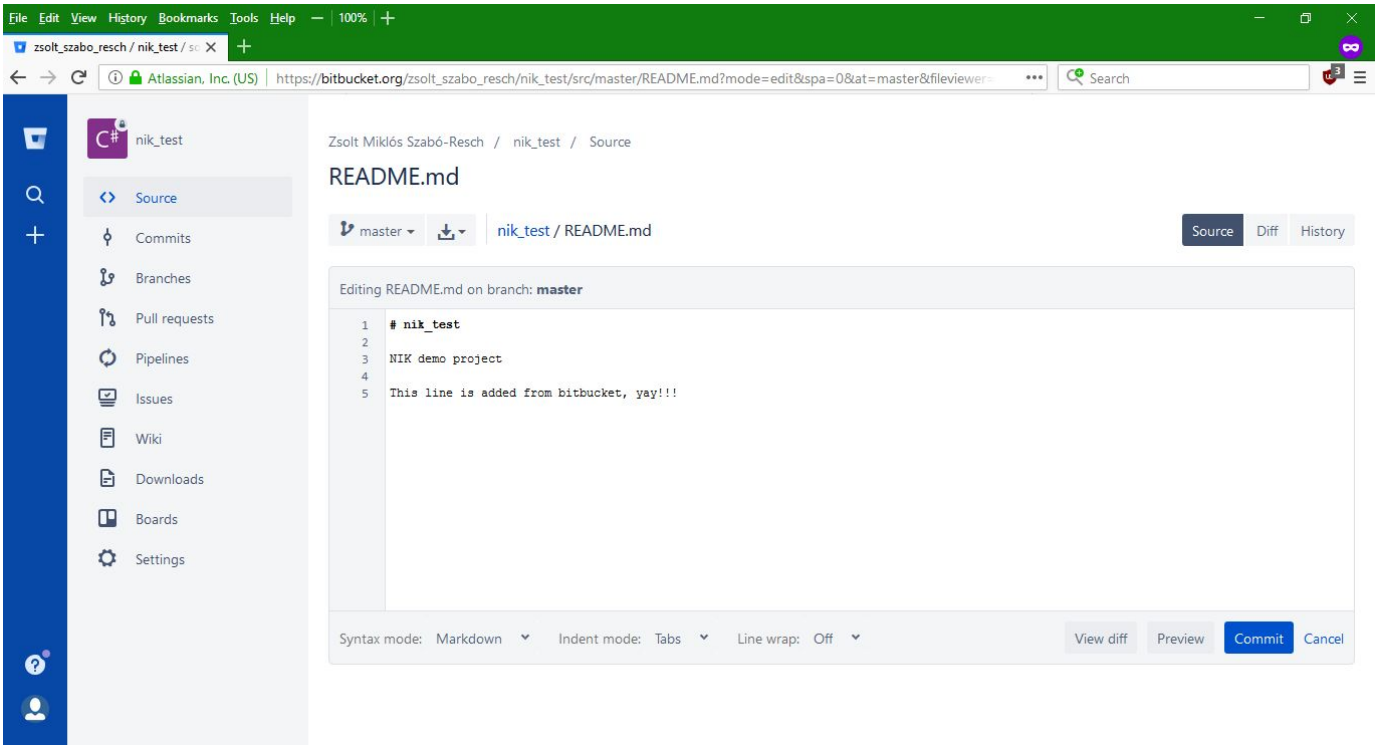
10. Then, copy the downloaded oenik.ruleset file into the Solution directory. Then right click in the solution explorer to the solution's name, Properties, Code analysis settings, and here choose the "Copy of NIK rules" elemet. (alternatively, if this is not shown, the manual installation steps are detailed in the StyleCop Quickstart section of this document)
11. Tools/NuGet package Manager/Manage NuGet Packages for Solution, here search for Stylecop.Analyzers, and add this to the project. These two steps (assign ruleset + enable Stylecop for project) have to be done later with all subsequent projects!

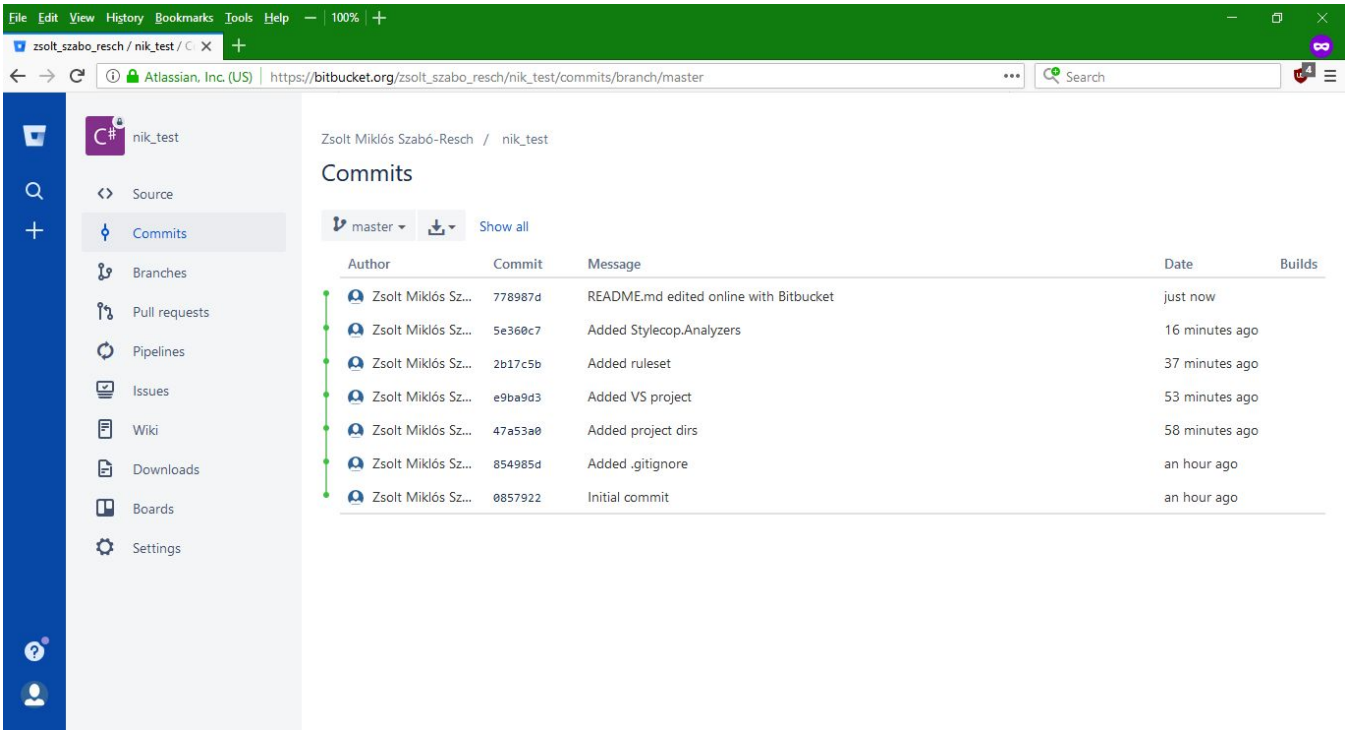
    Then again COMMIT, Stage All, COMMIT.



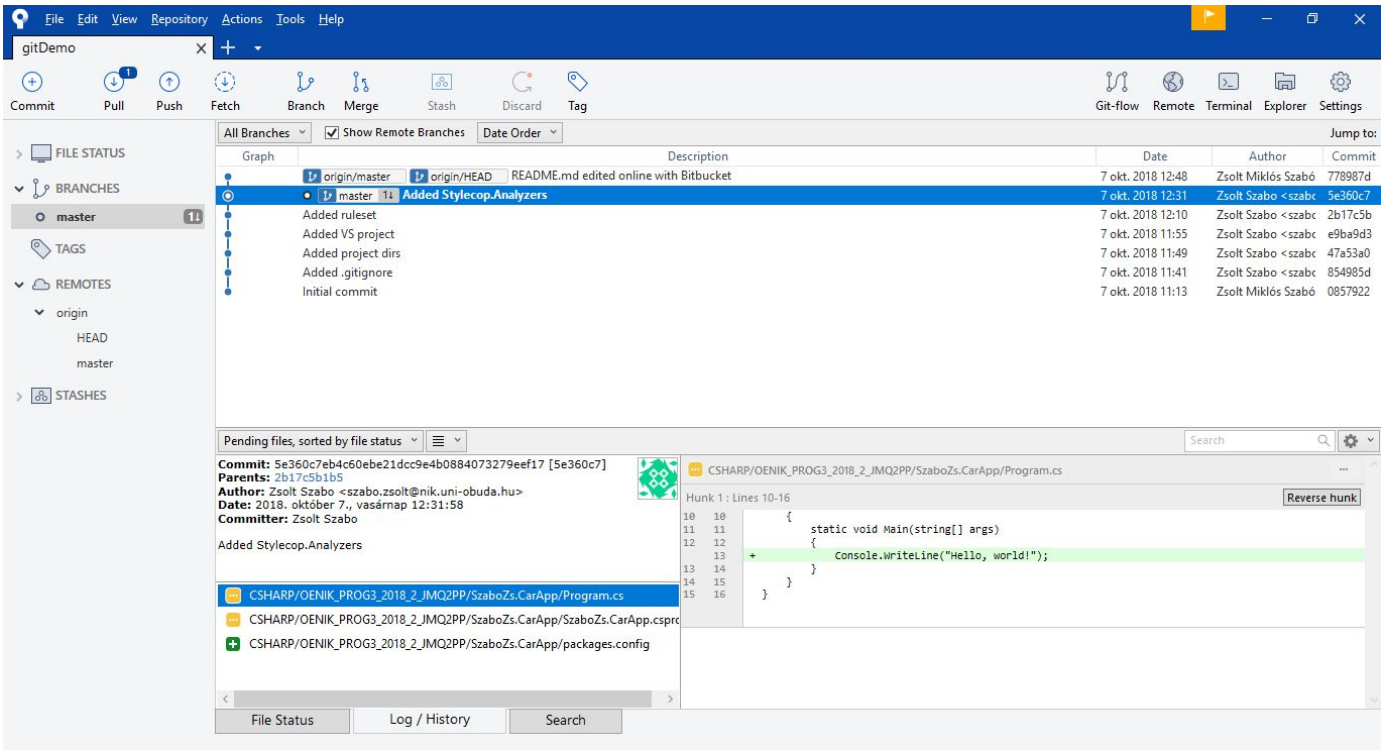
12. Changes can come from multiple sources: from our own VS version, or from external actors (other developers / even from the Bitbucket website). In case of external modification, the local repository has to be refreshed (PULL) before uploading the commits (COMMIT/PUSH); and if the external modification and our own modification changes the same file (CONFLICT), then this has to be taken care of (usually manually, with great care). For this, it is suggested to perform a FETCH/PULL before every COMMIT/PUSH.

13. We simulate an external modification: we modify the README file using the Bitbucket website!

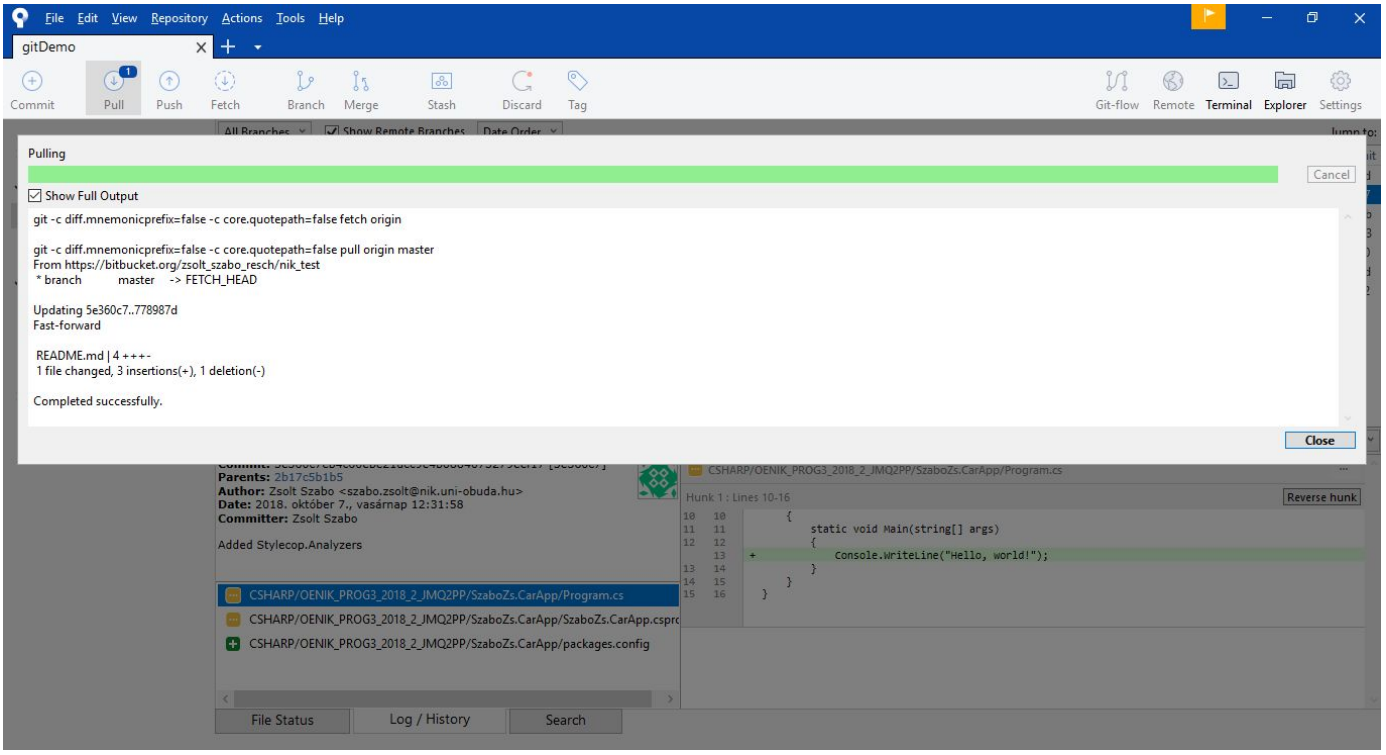


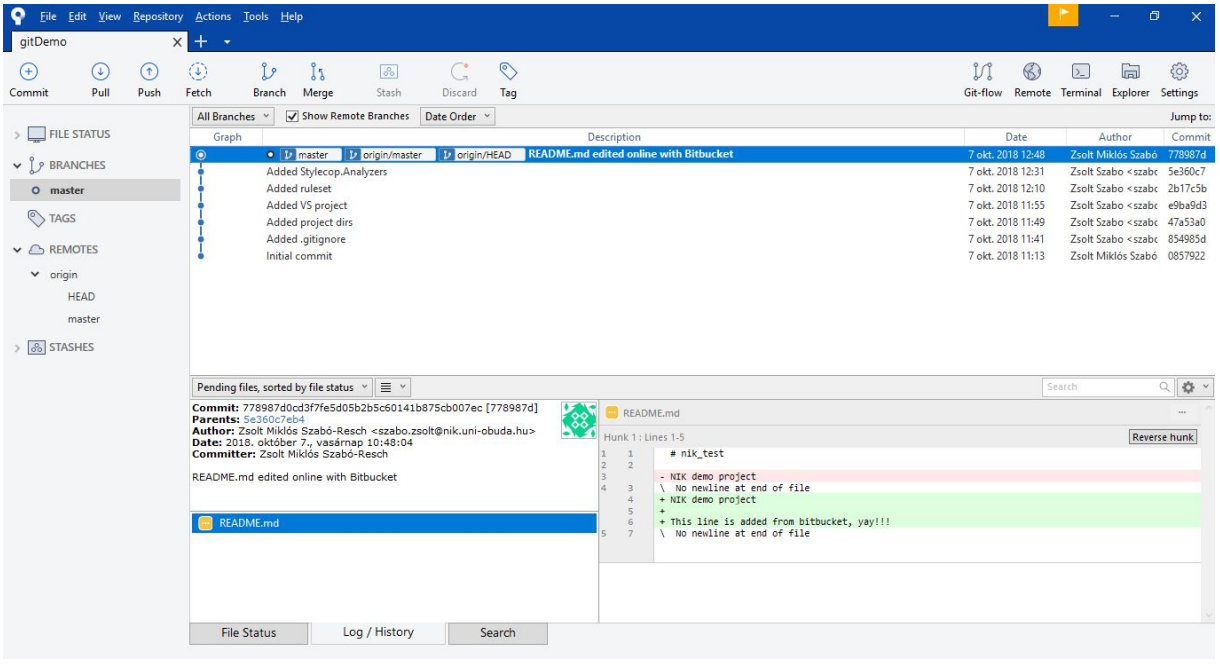


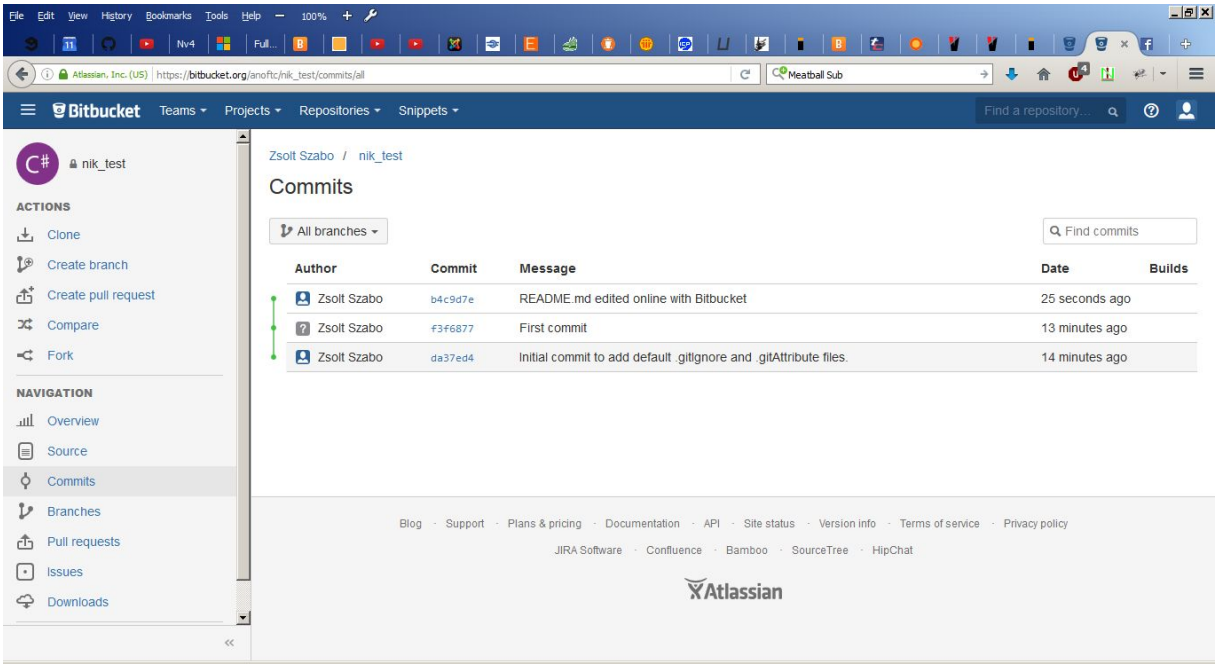
After Fetch (local tree = 1 behind):



After Pull:







This document only shows the basic Git operations. The strength of the truly good version control systems (GIT, TFS) is that they are capable of handling multiple code sections (BRANCHes) and also they are capable of effectively handling the separation and the unification (FORK, MERGE) of these branches. From the students, the handling of branches is not expected, but in a typical company environment, only the experienced senior programmers have MERGE permission to the main (master) branch, every other developer works on his/her own developer branch.

Other resources:

- https://en.wikipedia.org/wiki/Version_control
- https://en.wikipedia.org/wiki/Git
- http://blog.osteele.com/posts/2008/05/my-git-workflow/
- http://nvie.com/posts/a-successful-git-branching-model/
- https://tanarurkerem.hu/git-suli/
- SourceTree (GIT GUI)
- Tortoise GIT (GIT GUI)
- https://confluence.atlassian.com/bitbucketserver/basic-git-commands-776639767.html (GIT parancssor: Actions / Open In Terminal)
- https://bitbucket.org/zsolt_szabo_resch/nik_test/src

Teamwork

The creation of the project is now divided into two parts:

1. One team member creates the common GIT repository with the common VS solution with one project, basically following the description above. Also, this one team member gives permission to the other repo users (plus the usual oe_nik_prog user).
2. The other team members should simply clone the repository.

When using the repo from VS, the following links can be useful:

- https://docs.microsoft.com/en-us/vsts/git/tutorial/creatingrepo?tabs=visual-studio
- https://docs.microsoft.com/en-us/vsts/git/tutorial/clone?tabs=visual-studio
- https://docs.microsoft.com/en-us/vsts/git/tutorial/commits?tabs=visual-studio
- https://docs.microsoft.com/en-us/vsts/git/tutorial/branches?tabs=visual-studio
- https://docs.microsoft.com/en-us/vsts/git/tutorial/pushing?tabs=visual-studio
- https://docs.microsoft.com/en-us/vsts/git/tutorial/pulling?tabs=visual-studio
- https://docs.microsoft.com/en-us/vsts/git/tutorial/merging?tabs=visual-studio
- https://docs.microsoft.com/en-us/vsts/git/tutorial/undo?tabs=visual-studio

This example shows a good branch structure (this can be simplified in our case, the usage of the dev+master branches are obligatory, the usage of feature branches are recommended):

- http://nvie.com/posts/a-successful-git-branching-model/

StyleCop Quickstart

StyleCop is a VS plug-in that can help us in finding the styling / documentation errors in our source code.

Usage in VS2015

As a part of VS2015 the .NET Compiler Platform, or on its better known codename, Roslyn became available to the public. This is a toolset that contains opensource C# and VB compilers and code analysis APIs. With the appearance of Roslyn, all the previously used code analysis tools (among them VS builtin FXCop and StyleCop) needed to be migrated to this new system.
The best StyleCop version that already uses the new system is „StyleCop.Analyzers" which can be found on NUGet (installation: Tools / NUGet Package Manager / Manage NUGet Packages for Solution. Search for StyleCop.Analyers on „Browse" tab, select it, and then in the newly appearing window on the right select your project, and then press „install".)

After installation set up the ruleset to use:
1: Add oenik.ruleset to the project: Right click on project > Add existing item > All files (*.*) > browse for oenik.ruleset
2: Activate the ruleset: right click on oenik.ruleset, and Set As Active Ruleset.

StyleCop (and Studio-builtin FXCop) errors appear as standard warnings in the project after running the code analysis (project right click > Analyze > Run Code Analysis on Solution). After fixing some issues, the analysis can be re-run again like this.

In every code analysis tool, errors can be suppressed in case we do not want to fix them. Suppression of errors is typically not allowed in the projects (except: the auto-generated AssemblyInfo.cs and all other, truly auto-generated files. In these cases, prepend this line to the beginning of the files):

```
// <auto-generated/>
```

Take special care to only use Analyze > Run Code Analysis and **NEVER** Suppress Active Issues!

DoxyGen Quickstart

In addition to the manually written documentation and diagrams, an important part of the developer documentation is the automatically generated documents (using the source code). One of the tools for this task is the DoxyGen:
http://www.stack.nl/~dimitri/doxygen/download.html

XML documentations

All the public parts of the source code (including classes, methods, variables, properties, events and everything else) must be documented with XML documentation. Also all documentation must be added that is requested by the StyleCop ruleset.

The XML documentation parts of the C# code can be generated into a real XML file (because of StyleCop, this is required anyways): **Project Properties / Build / XML Documentation file**.

For the texts, English language must be used, however we will not accept automatically generated „english-like" texts created by Resharper and other similar programs – projects that are handed in containing documentation like that will count as non-documented (e.g. not handed in).

The format of XML documentation is as follows:

```
/// <summary>
/// Moves the current shape, if possible.
/// </summary>
/// <param name="x">x component of movement vector</param>
/// <param name="y">y component of movement vector</param>
public void Move(int x, int y)
{
    // ...
}
```

If „///" is typed over a finished method or class, Visual Studio will add the necessary template for this, which then just needs to be filled in. The completed XML documentation will appear in Intellisense which could help while writing code.

The main advantages of the DoxyGen are that it is open source, and it supports nearly all programming languages. It is much more customizable, however, the zillion of configuration parameters are hard to follow, and it is hard to wire all components together. In addition to this, it cannot be integrated into the VS. A good alternative might be SandCastle, but in a real development project the software used for generating documentation is very likely already given.

DOXYGEN

Command line:

- https://sourceforge.net/projects/doxygen/files/ , download appropriate version
- doxygen -g doxygen.cfg (this command generates the config file)
- Edit the config file
- doxygen doxygen.cfg

GUI (doxywizard):

- http://www.stack.nl/~dimitri/doxygen/download.html#srcbin , doxygen-1.x.y-setup.exe
- Working directory: same directory from where doxywizard is running.
- Project name: any name for the project.
- Source code directory: the directory that contains the .sln file for the project.
- Scan recursively: yes
- Destination directory: create a new directory called „Doxygen” in the directory that contains the .sln file.

<Next>

- Documented Entities Only
- Include cross-referenced source code: yes
- Optimize for Java or C# output

<Next>

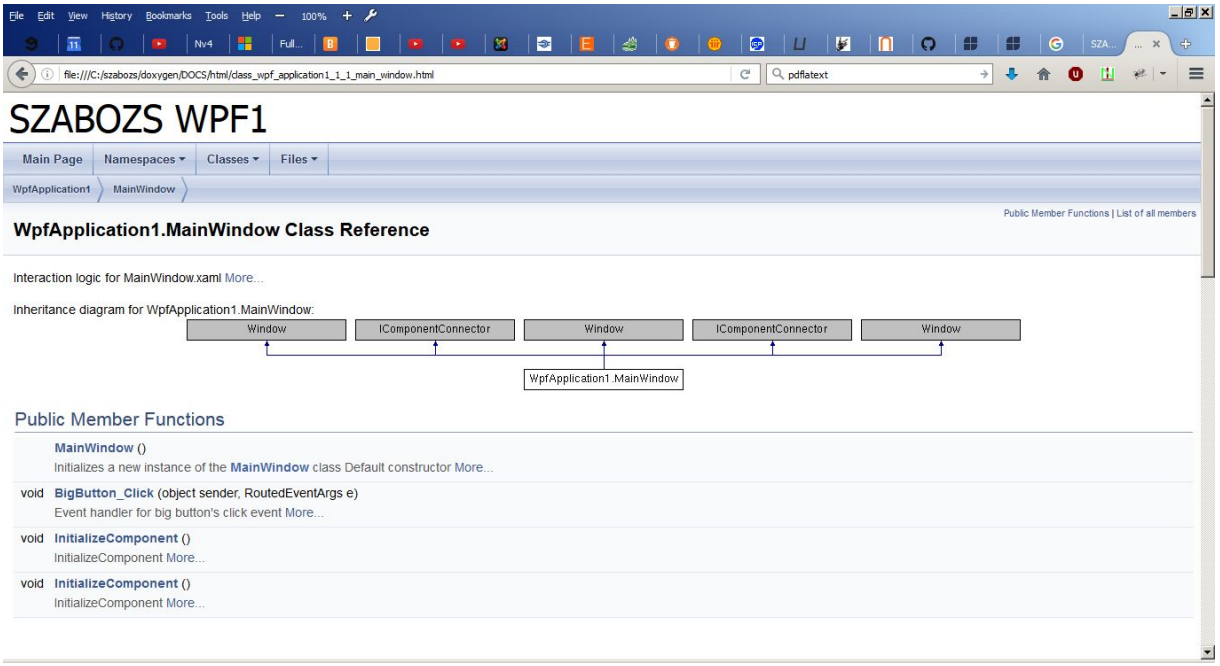
- HTML: With navigation panel
- LaTex: no (html tickbox stays ticked).

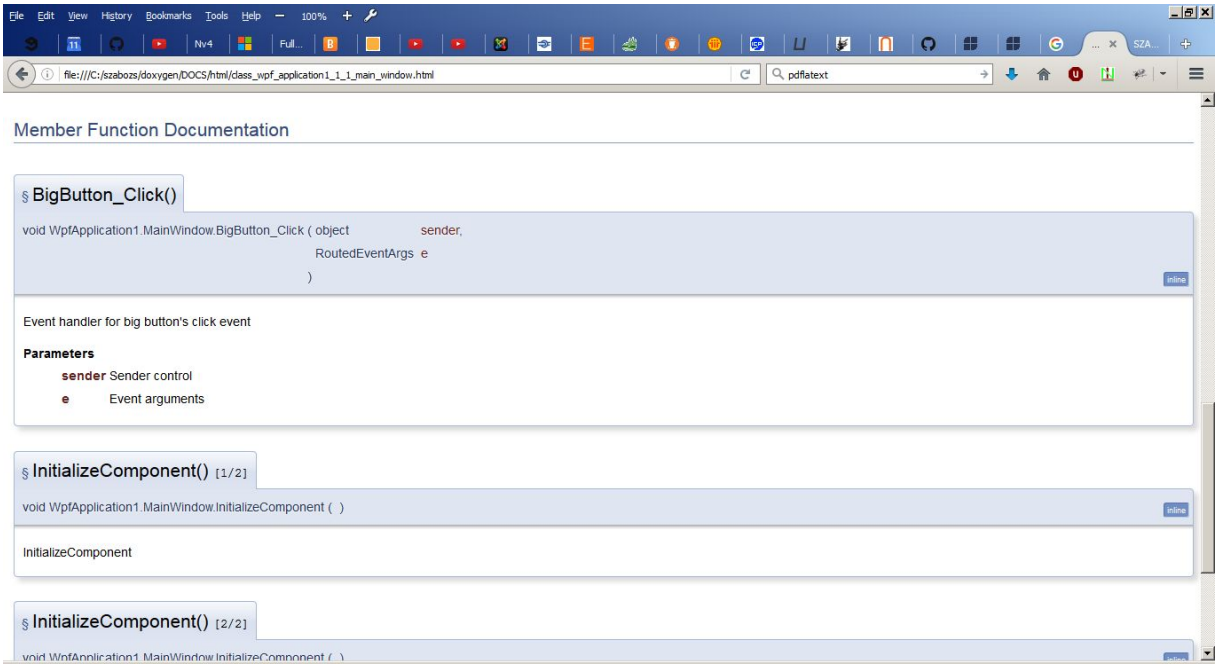
After some additional „Next”s press „Run doxygen”.

If we have lots of internal/static members, do not forget that those are by default ignored by the system. In this case, add the following to settings:

EXTRACT_PACKAGE = YES
EXTRACT_STATIC = YES

Check if the documentation has been generated correctly starting at index.html: the documentation should contain the XML documentation for the public classes and methods.


SZABOZS WPF1
Main Page
Namespaces
Classes
Files
WpfApplication1
MainWindow
Public Member Functions | List of all members
WpfApplication1.MainWindow Class Reference
Interaction logic for MainWindow.xaml More...
Inheritance diagram for WpfApplication1.MainWindow:
Window
IComponentConnector
Window
IComponentConnector
Window
WpfApplication1.MainWindow
Public Member Functions
MainWindow ()
Initializes a new instance of the MainWindow class Default constructor More...
void BigButton_Click (object sender, RoutedEventArgs e)
Event handler for big button's click event More...
void InitializeComponent ()
InitializeComponent More...
void InitializeComponent ()
InitializeComponent More...


Member Function Documentation
§ BigButton_Click()
void WpfApplication1.MainWindow.BigButton_Click (object sender,
RoutedEventArgs e
)
inline
Event handler for big button's click event
Parameters
sender Sender control
e Event arguments
§ InitializeComponent() [1/2]
void WpfApplication1.MainWindow.InitializeComponent ( )
inline
InitializeComponent
§ InitializeComponent() [2/2]