

Laboratory

Lab Session 1

Introduction to Visual Studio and Azure DevOps

Software Engineering

ETS Computer Science
DSIC – UPV

Year 2019-2020

1. Goal

The goal of this lab session is to introduce the student to the main functionalities of *Visual Studio* 2019 and its relationship with the team project management and control version tool Azure DevOps. It is mandatory to complete the tasks described until section 5 (included). Section 6 is optional.

2. Creating a team project using Azure DevOps and an initial solution in the server

In order to correctly fulfil this lab session, you should follow the steps to create a team project with Azure DevOps in the cloud, register your team members in that project, and create a solution inside it. It is recommended to name the Azure DevOps server using the following schema: 2019-upv-isw-3EL1-<team name>. For instance, the server could reside at <https://dev.azure.com/2019-upv-isw-3EL1-team03>.

These preliminary tasks will be performed by only one member of the team (the Team Master).

The realization of the activities stated above has been described in the theory seminars of chapters 2 and 3. You should finish them before continuing to the next steps (consult the existing material in *PoliformaT* related to these seminars). At the completion of these activities, the development team will have a team project with a *Visual Studio* solution containing the *Presentation*, *Library* (with the *BusinessLogic* and *Persistence* subfolders) and *Testing* solution folders, as shown in Figure 1..

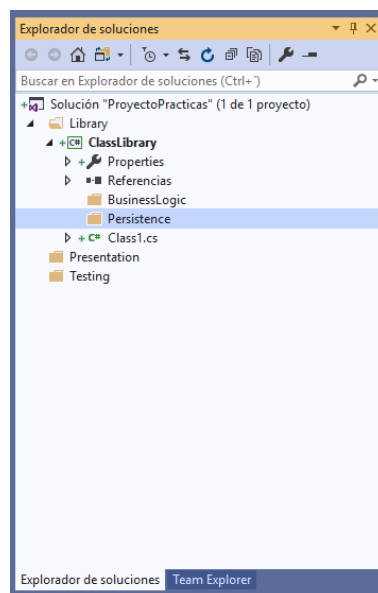


Figure 1. Initial Solution folders and modelling project

3. Connection to the existing project from Visual Studio

At this moment, each member of the team can connect **individually** from *Visual Studio* to the *DevOps* project, download a copy of the solution from the cloud server and assign it to a local

workspace (local folder in C:\1) to work with it. To do so, follow the next steps (you can do it **individually**):

- a. Start the development tool Microsoft *Visual Studio 2015 Enterprise*.
- b. Select the option *Continuar sin código* in the pop-up dialog:

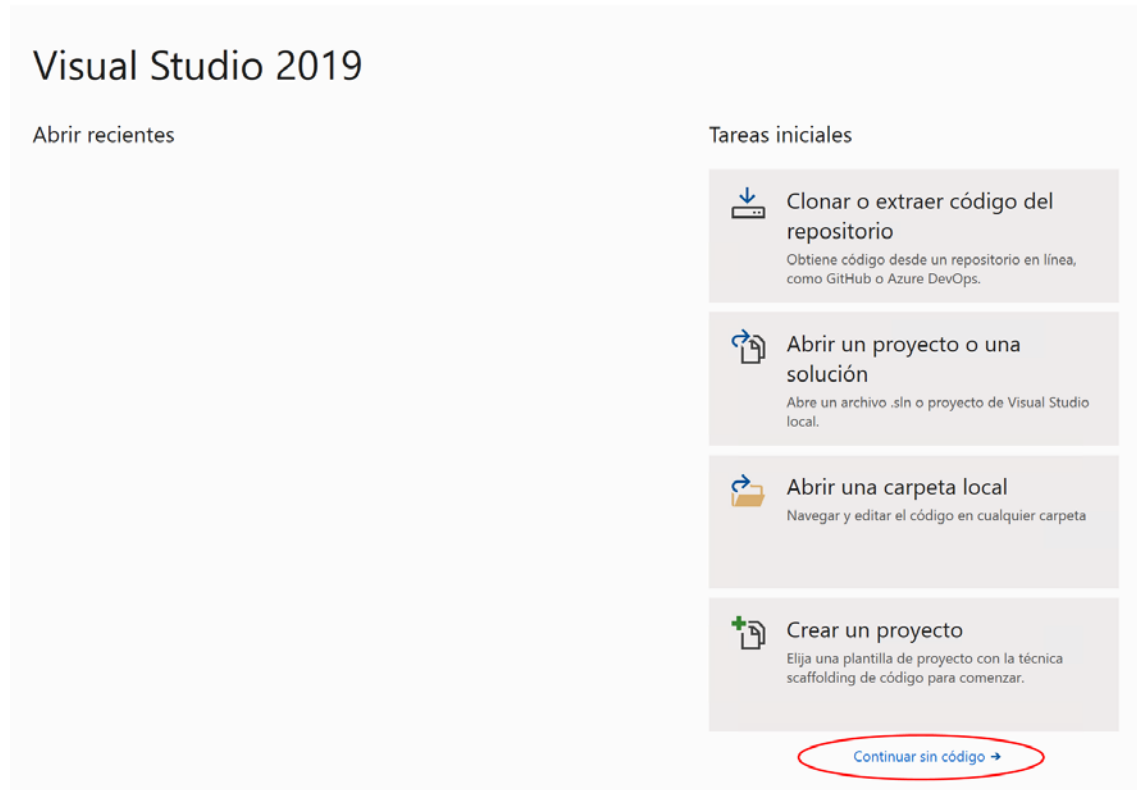


Figure 2. "Continuar sin código" option

- c. Login with your Microsoft user account:
 - Archivo->Configuración de la cuenta->Iniciar Sesión

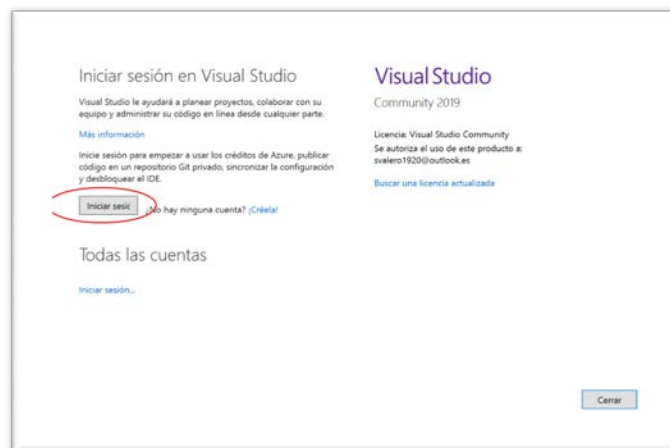


Figure 3. Starting a session in Visual Studio

¹ In DSIC laboratories, using W personal unit is not recommended, because is a network resource and the connection to it can be lost in the middle of the work session, causing errors in Visual Studio.

d. Connect to your team project:

- Choose Team Explorer (right frame) and then click on the *Team Services connection button* (green plug, Figure 4).
- Click on the option *Administrar conexiones* (Figure 6-2)

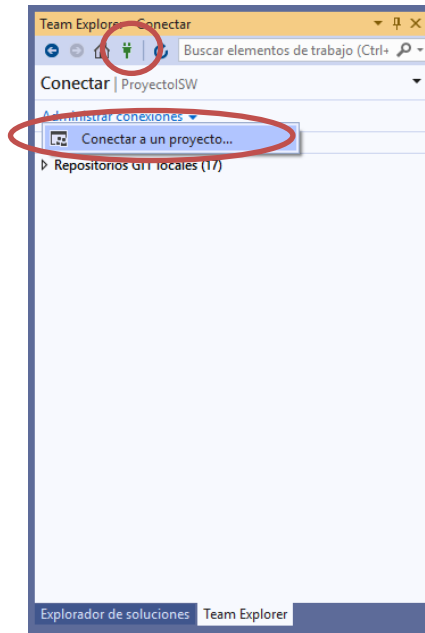


Figure 4. Connecting to a team project from "Administrar Conexiones"

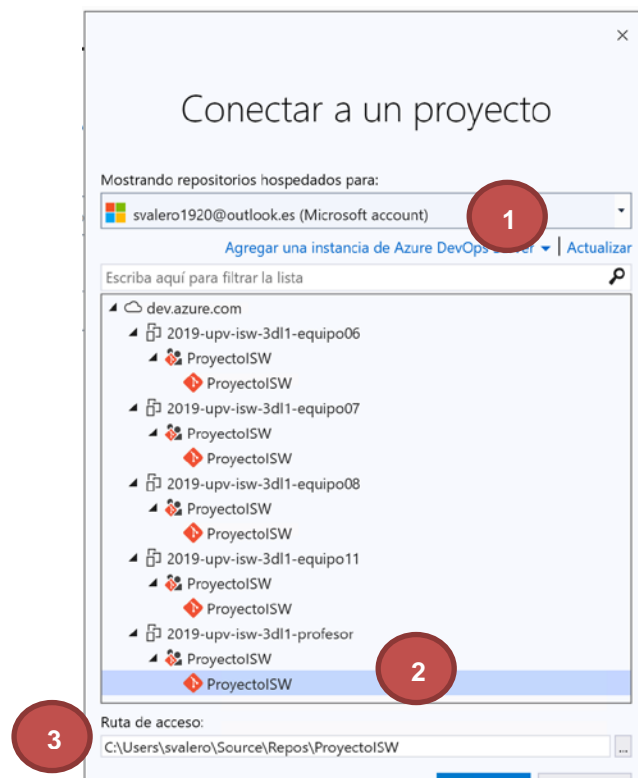


Figure 5. Connect to an organization's project

- In the dialog that appears (see Figure 5), you can see the Microsoft account used to login (Figure 5-1). Under this drop-down list, the list of organizations to which you belong appears (in your case, it is normal that only one appears). Also, for each organization, the projects you have appear (again, only one in your case). Selecting the git symbol of your project (Figure 5-2), confirm the route in which you want to save the project (Figure 5-3 and click on clone (Figure 5-4). You may have to confirm the account you are going to use to connect to the remote repository.

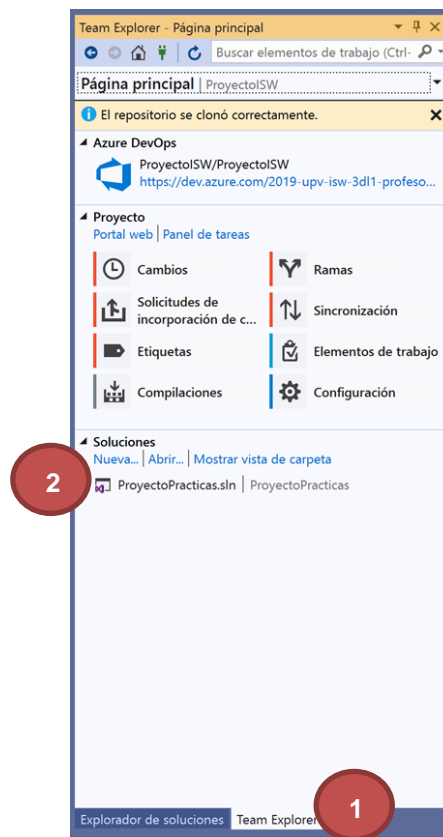


Figure 6. Open a solution from the Team Explorer

- Open the solution created by the Team Master. Once connected to the team project and the local repository has been obtained and assigned, you can open the solution that the Team Master created previously (step 2 in this guide):
 - Choose Team Explorer (right frame Figure 6-1)
 - In section *Soluciones*, click on your solution (Figure 6-2)
- Go to the solution explorer (tab *Explorador de soluciones* at the bottom-right side of *Visual Studio*) and check that you have a solution as the one the *Team Master* has originally created and stored in the repository, as shown in Figure 1.

4. Introduction to the Visual Studio debugging tools

The Visual Studio debugging tools are a key element for the development of applications in Visual Studio and the detection of bugs in the code. These tools allow to set break points in the code, run the code step by step, inspect the values of variables during the execution, etc.

4.1. Creating a console project

In order to get started with debugging tasks, we are going to **create a console project in the solution folder *Testing***:

- Create a subfolder named *Lab1* inside *Testing* folder..
- Right click on folder *Lab1*, and then select *Agregar -> Nuevo Proyecto-> Aplicación de consola* and name the application ***HelloWorldApp*** (select the console project type shown in Figure 7).

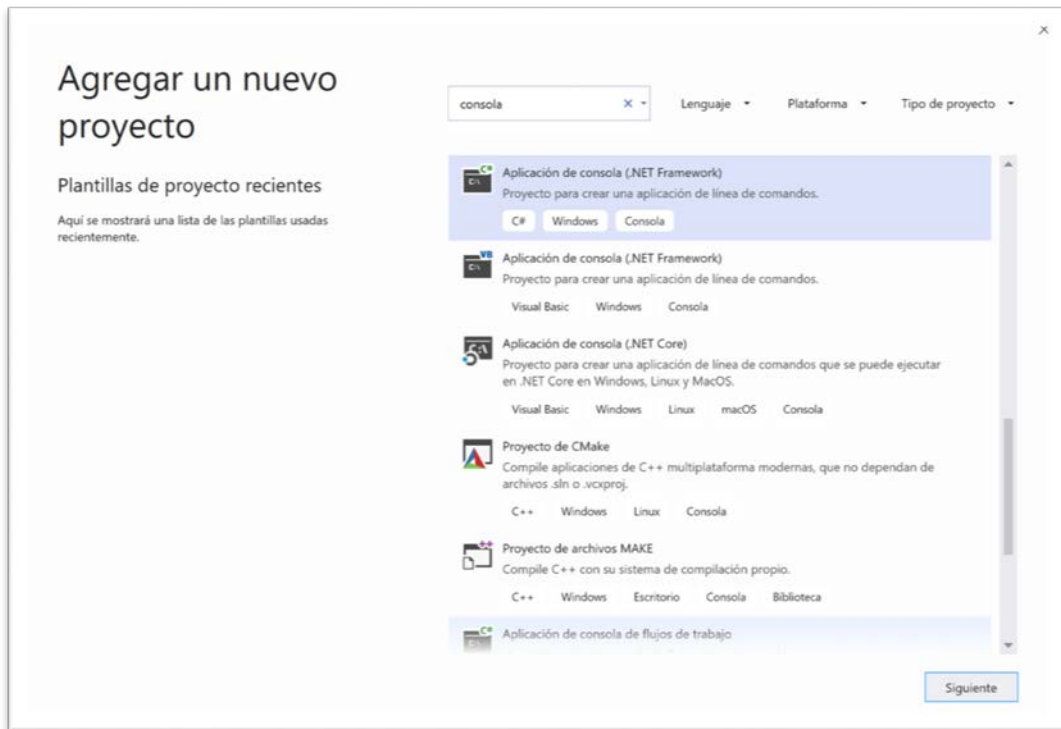


Figure 7 Creation of HelloWorldApp

- Once the application has been generated, observe the different windows that Visual Studio provides (see Figure 8): *Cuadro de Herramientas* (tool box, left panel); *Explorador de Soluciones* (solution explorer, upper-right panel); *Propiedades* (properties, lower-right panel); *Editor de Código* (code editor, upper-center panels); *lista de errores y salida* (error lists and output, lower-center panels).
- The *HelloWorldApp* project should be marked as the startup project (being highlighted in bold). Otherwise it should be marked as such before compiling and executing it. To mark the project as the startup one, click the mouse right button on the *HelloWorldApp* project and select the option ***Establecer como proyecto de Inicio***. Observe that the name of the project is in bold now.

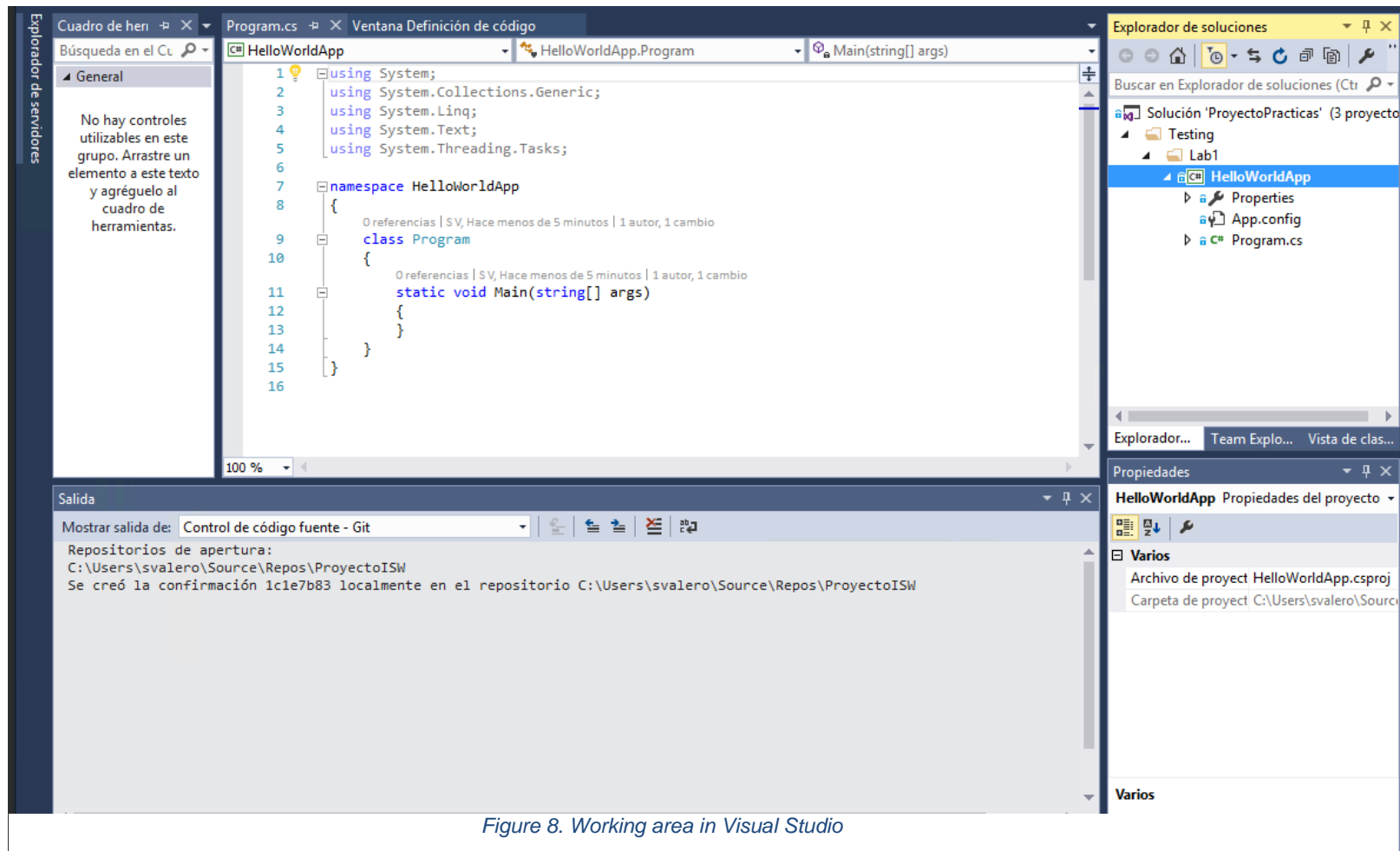
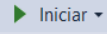


Figure 8. Working area in Visual Studio

4.1. Compile and run

To compile and run this application, click on the green triangle button *Iniciar* on the upper area of the development tool (). The output window will show the result of the compilation and the executing, although the current program will not perform any action.

4.2. Exploring compilation errors

Open the file *Program.cs* and type an error artificially in the main method, for example, by writing the sentence *fadsfsde*. If you compile again the program you will observe the list of errors in the corresponding window. If you click on the error, the cursor will appear where it is in your code.

4.3. Creating a HelloWorld application in C#

For practicing the debugging abilities, we are going to create a simple program as an example. Open the file *Program.cs* and modify its contents to make it look as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HelloWorldApp
{
    class Program
    {
        private static List<string> nacionalities;
        private static void InitList()
        {
            nacionalities = new List<string>()
            {
                "Australian",
                "Mongolian",
                "Russian",
                "Austrian",
                "Brazilian"
            };
        }

        static void Main(string[] args)
        {
            InitList();
            nacionalities.Sort();
            foreach (string nationality in nacionalities)
            {
                Console.WriteLine(nationality);
            }
        }
    }
}
```

The previous program starts with several *using* sentences, which act as compiler directives. These directives make possible to use types defined in the imported namespace without requiring us to specify them again explicitly in the code. After this, the namespace *HelloWorldApp* is defined and within it the class *Program*. This class contains the definition of a list, its initialization method *InitList* and a static method *Main*. The program creates a list of *strings*, orders it alphabetically and then goes through this list and shows its contents on the console. This code will be used to practice our debugging abilities. Compile and run the program. You will see a console window in which the program executes, but it will run so fast, preventing us from observing the output.

4.4. Run a program step by step

Most IDEs currently allow including break points in which the execution is stopped in order to run the code sentence by sentence from that point and observe its behavior in a controlled manner. In order to practice this aspect, we will perform the following actions:

- Insert a break point in the code line *InitList()*, clicking on the grey column which appears left to the code, at the same level of the line in which you would like to stop the execution (see Figure 9-1).

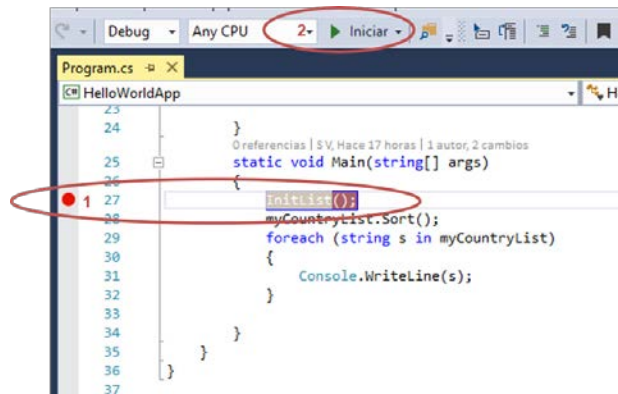


Figure 9. Adding break points

- Compile and start debugging the code (click on the green triangle, Figure 11-2) and observe how the execution is stopped at the line in which you previously set the red point (a yellow arrow appears indicating where the execution is). If you observe the console window created, this would be empty.

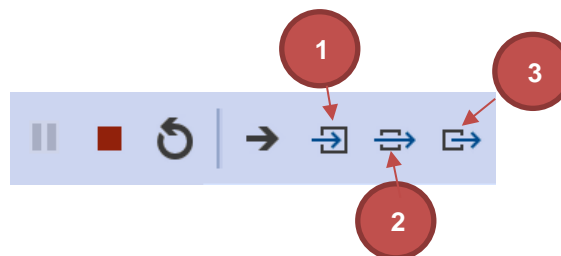


Figure 10. Step by step execution

- Run the application step by step. Use the step-by-step execution controls (Figure 10). You can stop the execution (red square), run the next instruction (Figure 10-1 or F11), run a whole procedure/method completely without stepping into it (Figure 10-2 or F10), or run all the sentences in a procedure to go out of it Figure 10-3 or Shift+F11).

4.5. Observing the data values of the program

While the program is being executed, it is possible to observe the values of objects, attributes, variables, etc. This is **extremely useful** when debugging a program. To observe the value of any element, place the cursor over it.

- When your program arrives to the execution of the sentence *nacionalities.Sort()*, place the cursor over the variable *nacionalities* and observe its content (Figure 11).

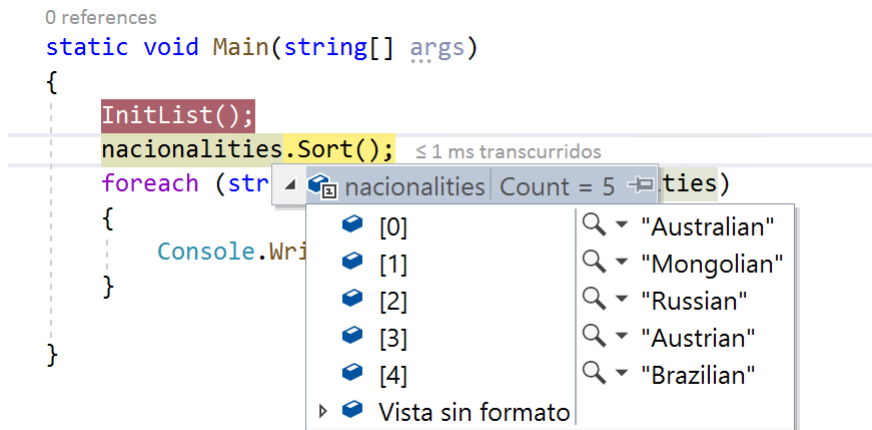


Figure 11. Inspecting data values of variables

- b. Execute this sentence step by step (F11) and observe again the variable *nacionalities*, which has been ordered alphabetically.
- c. Keep running the program step by step and observe how the console window shows the content of the list (Figure 12).

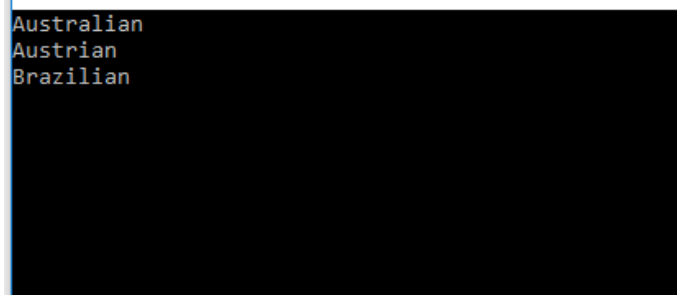


Figure 12. Console output

- d. You can go back in the execution just by dragging the yellow arrow (see Figure 13) up towards the line of code in which you would like to repeat the execution of your program.

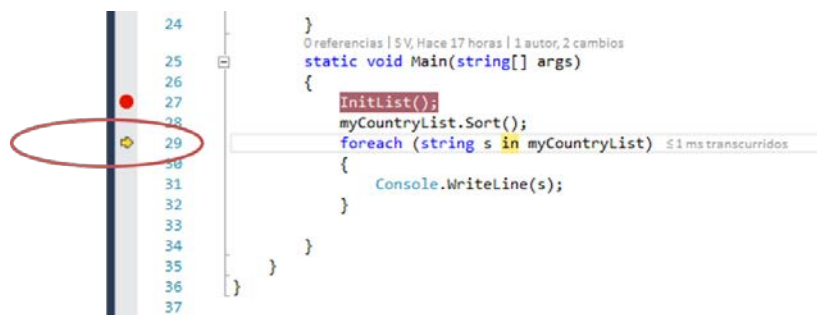


Figure 13. Changing the execution flow in debugging

5. Version Control Basic Operations

An advantage of using a collaborative development environment like Azure DevOps is being able to control and manage the different versions of the project as it is being generated. In fact, when

you created your project, you indicated whether you wanted to use GIT or Team Foundation version control. A version control system is a combination of technologies and practices to control the changes made to the different elements that comprise a project, especially the source code, the documentation, the sets of tests, etc.

It is convenient to know some nomenclature associated to Git, as it will be the version control system we will use:

- **Remote repository:** a repository in which all the team members will store its changes.
- **Local repository:** an instance of the remote repository in which each member will work locally in its machine, and not always is synchronized with the remote one.
- **commit:** Stores the last changes made in your repository. It not only stores which files were changed and the lines affected, because also is linked who makes the changes, in which data and a user message to explaining the changes. This message is mandatory, and it is convenient also writing in it some references to the related tasks from the plan project.
- **clone:** Obtaining a copy of the project from the remote repository. A local repository is created, in which is possible to see all the commits made from the beginning of the project.
- **pull:** Retrieving the last commits stored in the remote repository to your local repository.
- **push:** upload the las commits stored in the local repository to the remote one.
- **merge:** it is the process in which the different commits (and their changes) are combined to obtain a new consistent version of the project.
- **conflict:** It happens when two or more people try to perform different changes in the same piece of code. Thus, when two different commits are combined in a merge process (for example during a pull) and it is detected that they contain changes in a same file, the system request the user to confirm the final change which will be stored. As a result of resolving a conflict, we will get a new commit and a coherent version of the project

Git version control follows a workflow that most developers follow when they write code and share it with the development team. The steps are the following:

1. Obtain a local copy of the project code
2. Make changes to the code
3. Once the changes have been completed, publish the modified code to the rest of the team
4. Once accepted, merge with the code in the team repository.

Graphically, the previous workflow can be represented as Figure 14 shown:

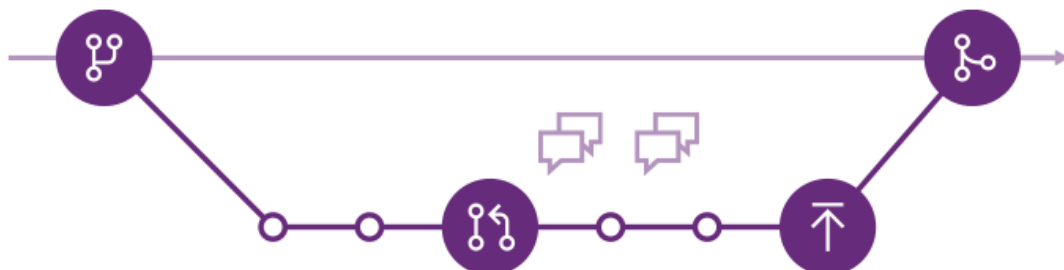


Figure 14. Git workflow

5.1. Committing changes

We are going to simulate the normal workflow in a developing team:

- Modify any part of your code. For instance, the *Program* class so that the first element in the list is different. Each member of the team may use a different string (*"Italian"*, *"spanish"*, etc.)
- Go to *Team Explorer* -> *Cambios* (Figure 15-2), and then add a descriptive comment for the new version (Figure 15-3).
- Click on *Confirmar todo* button (Figure 15-4). A new commit with the last changes will be created.

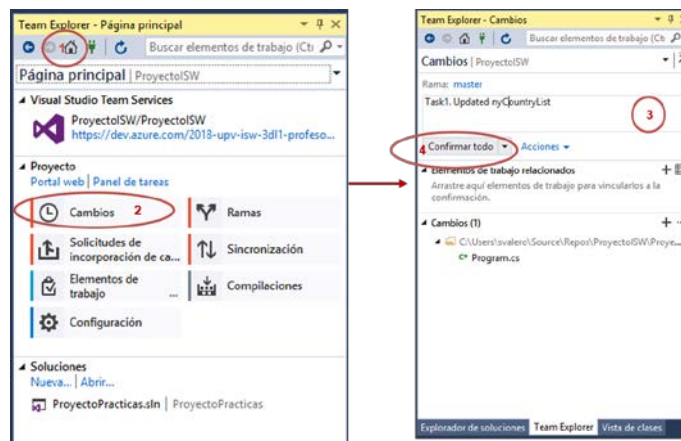


Figure 15. Committing changes

5.2. Pushing changes and resolving conflicts

After working locally, you will share your work with your development team, in this process conflicts can appear, and you will need to resolve them.

- a) Select the option *Team Explorer -> Sincronización*. In the new dialog, clicks on the link *Sincronizar*². All pending commits will be added to the remote repository.

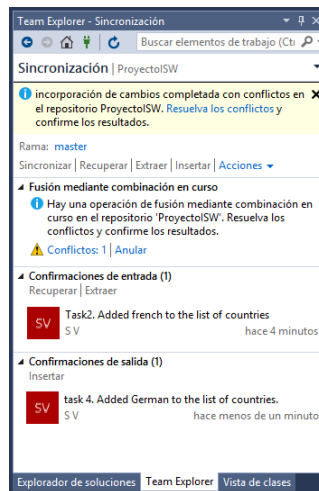


Figure 16. Conflicts in push/pull process

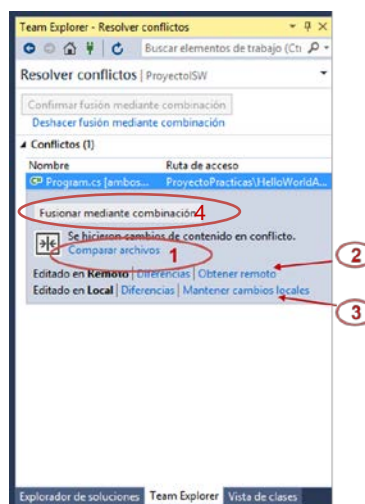


Figure 17. Access to the tool to resolve conflicts

- b) **Resolving conflicts.** It is possible that conflicts arise if another user generated a new version before and you do not have the latest version stored in your local repository (see Figure 16 for an example). Conflict resolution may be done in a manual way, by using a tool for conflict resolution, to decide the code version to be kept when the solution is uploaded to the repository. In Figure 16, we can initiate the process clicking on the link *Resuelva los conflictos* or in the link *Conflicts*.
- c) In the dialog *Resolver conflictos*, select the link *Comparar archivos* (see Figure 17-1).

² When you add a commit, also the option to synchronize in this moment appears.

- d) The tool to compare the conflict appears (Figure 18). It is possible to compare the commit in the remote repository (left side) which has conflicts with the local commit in conflict (right side)

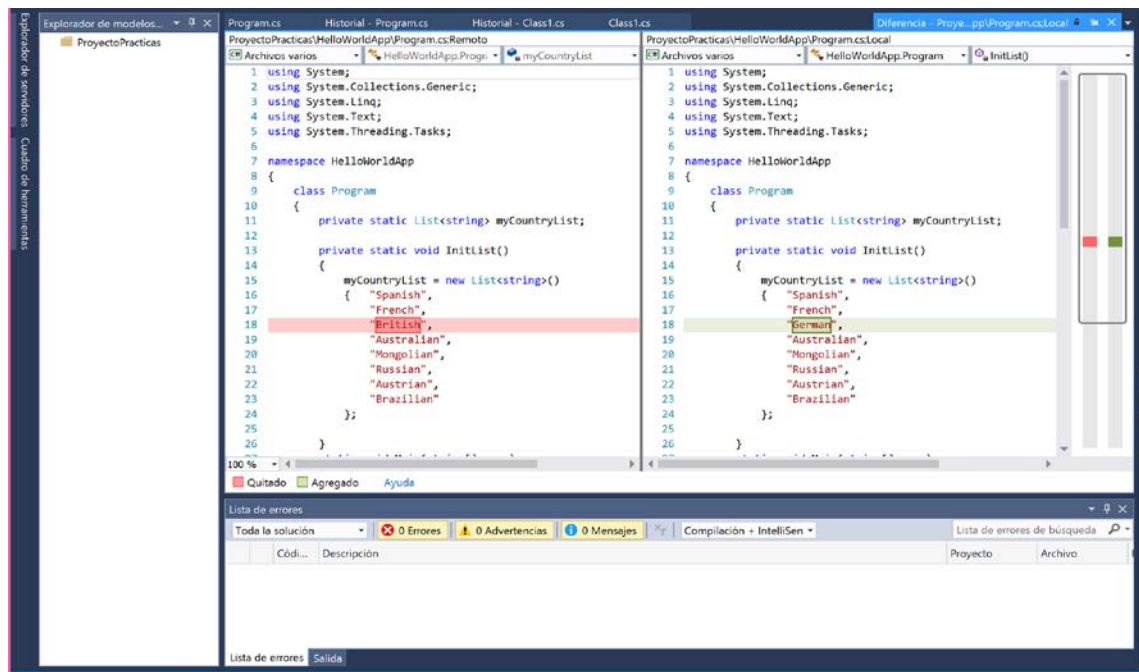


Figure 18. Tool to compare commits

- e) In this moment, you can decide to maintain your local changes clicking the option *Mantener los cambios locales* (see Figure 17-3), or the remote ones by selecting the link *Obtener remoto* (see Figure 17-2). In this example, you can see a conflict due to different values in line 18.

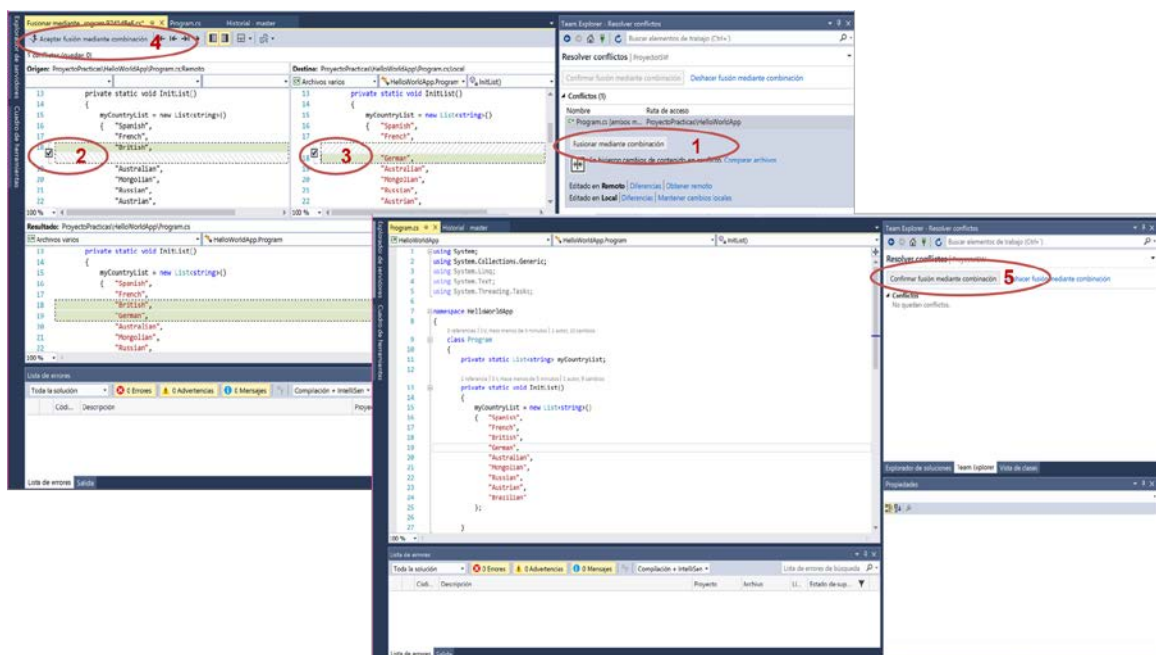


Figure 19. Combining both commits

- f) If you need to combine both, you need to select the option *Fusionar mediante combinación* (see Figure 17-4). Then, a new tool appears in which you can select which changes will be finally added to the final result (see Figure 19). After selecting them, you need to accept the changes by clicking the option *Aceptar fusión mediante combinación* (see Figure 19-4). Finally, you can see the resulting file and, in order to finish, you should click on the button *Confirmar fusión mediante combinación*, in the dialog *Confirmar cambios* (see Figure 19-5).

5.3. History of changes

The history of changes of any item of your project may be observed by using the code version explorer from Visual Studio. Thus, select *Explorador de Soluciones*, then click with the right button of the mouse over the element to explore and select *Ver historial* (see Figure 20).

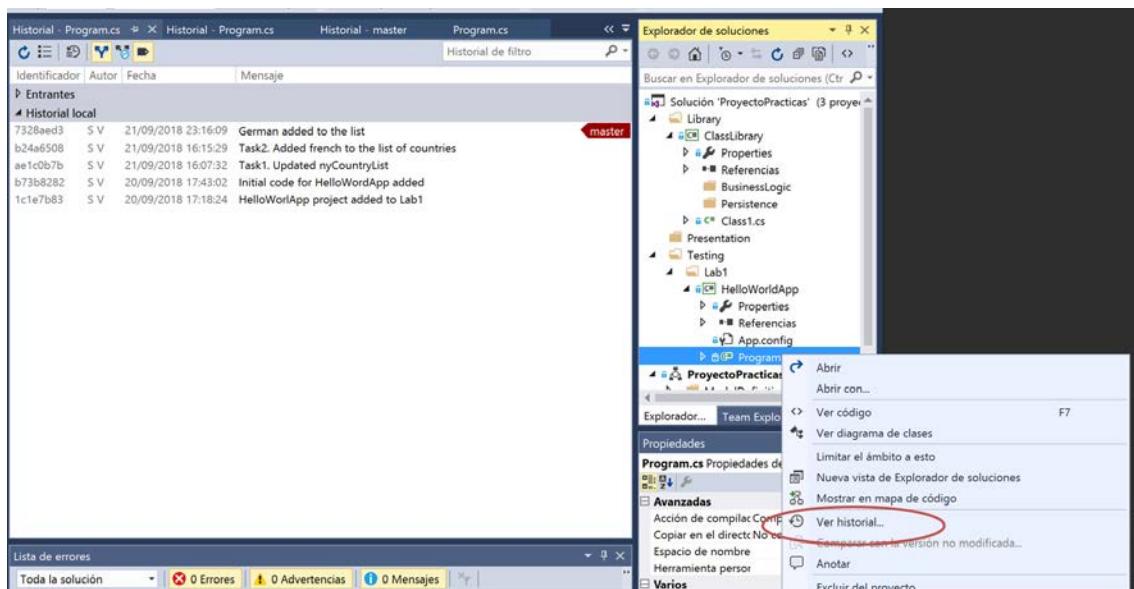


Figure 20. See repository history

In a similar way, Azure DevOps allows seeing the history of changes of any item using the option *Repos>Files*. Then, you select the target file and press on the tab *History* (see Figure 21).

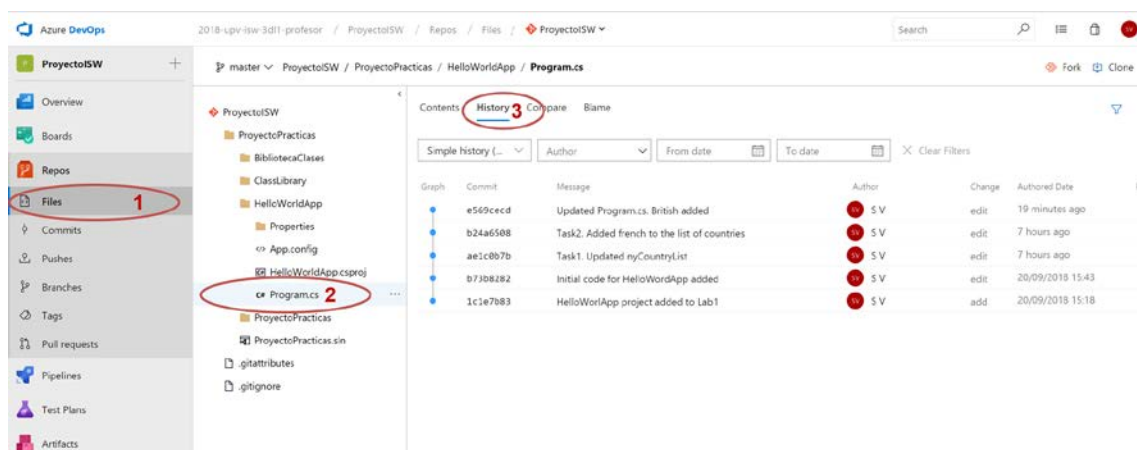


Figure 21. Visualize the change history

Also, it is possible to see all the commits of a branch in the option *Repos>Commits* of Azure DevOps. This option also is available in Visual Studio, from *Team Explorer>Ramas*. In this case, you should select the branch and then select from the context menu the option *Ver Historial*.

6. (Optional Task) Advanced Operations with the Version Control

In this section we will learn some additional concepts related to version control:

- **branch:** you can see a branch as a development line, in Git as a sequence of commits. In Git, by default, a repository has only a branch named master, in which are stored all the commits. But, it is possible to create a new branch from any point, so developers can work independently, and store its changes in a new development line, without affecting the users who work in the original branch. Branches can be merged, so the work can be added from one branch to another. It is convenient using consistent naming convention for your feature branches to identify the work done in the branch. For instance:
 - users/username/description
 - users/username/workitem
 - bugfix/description
 - features/feature-name
 - features/feature-area/feature-name
 - hotfix/description

6.1. Generating a development line (branch)

To generate a development line in a way that it isolates the changes made by you from the ones made by others, you can proceed as follows:

- a. The Team Master must grant branch creation permission to the rest of the team members at the Azure DevOps (by default, contributors already have got this permission) from *Project settings>Code>Repositories>Users>User_Name>Access Control>Create Branch* of the project (see Figure 22).

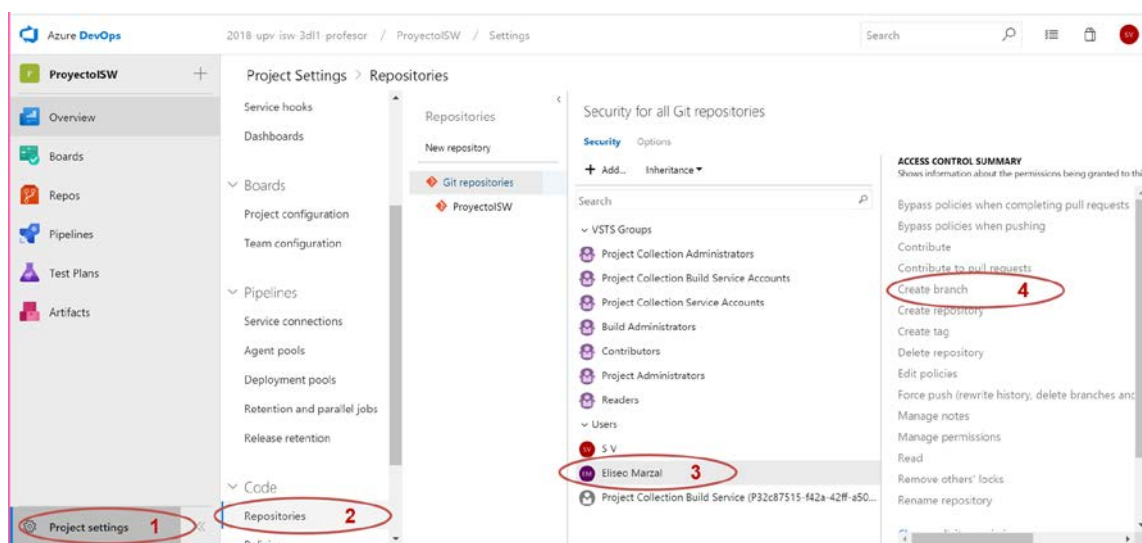


Figure 22. Project settings. Check the repository permissions

- b. Once they are granted permission, users can create branches. In Visual Studio, select *Team Explorer>Ramas* (Figure 20-1).
- c. Select the branch from you want to create a new one (local branch master in Figure 23), and then from the context menu select the option *Nueva rama local de...* (Figure 23-3).

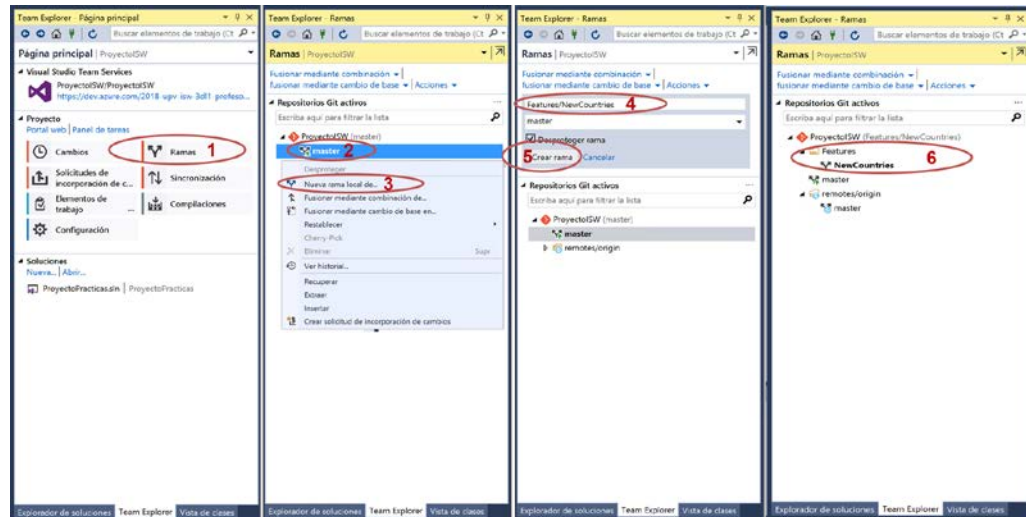


Figure 23. Creating a new local branch

- d. Finally, assign a new name for the branch (Figure 23-4) and click on the button *Crear rama* (Figure 23-5). As a result, a new feature branch³ is created in the local repository (Figure 23-6)

6.2. Updating and pushing a local branch

As it has been discussed earlier, at any time a local branch can be created from other branch, creating a new line of development. From this point, new changes can be committed in the new branch and also pushed to the remote repository, without interfere in the original branch or line of development. Now, let is practise with this:

- a. Modify again the class *Program* by changing another element in the list (“Italian”, “Portuguese”, etc.)
- b. Create a new commit with your changes from *Team Explorer> Cambios* and then push them to the remote repository, clicking on the link *Sincronizar* from the last dialog (see Figure 24).
- c. Finally, we need to click on *Publicar*, from the section *Confirmaciones de salida* to push the new branch to remote repository (Figure 25). You can check it from the option *Team Explorer> Ramas* (third dialog in Figure 25)

³ As the selected name use the convention Feature/NewFeature, a new folder into the local repository is created to store the branch.

- d. Also, you can see the two existing branches on Azure DevOps web, selecting the option *Repos>Branches* (Figure 26)

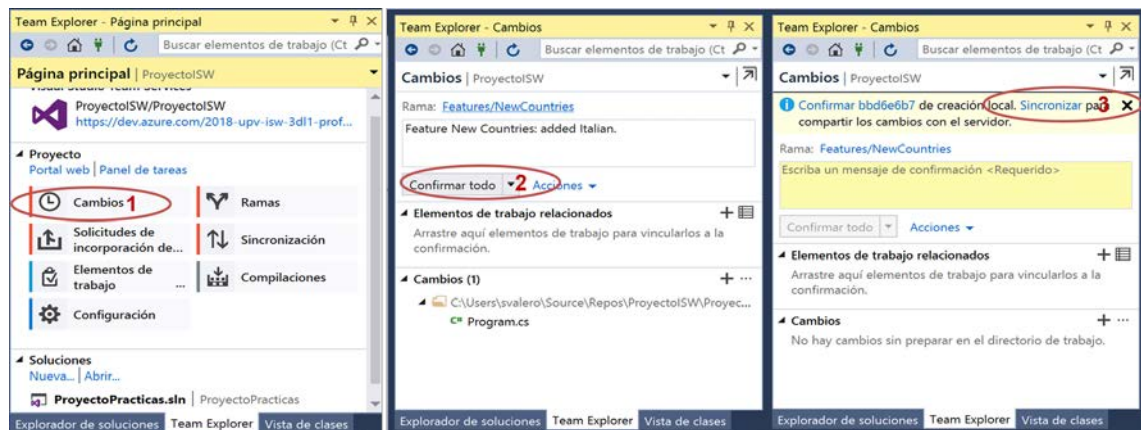


Figure 24. Commit in the local branch

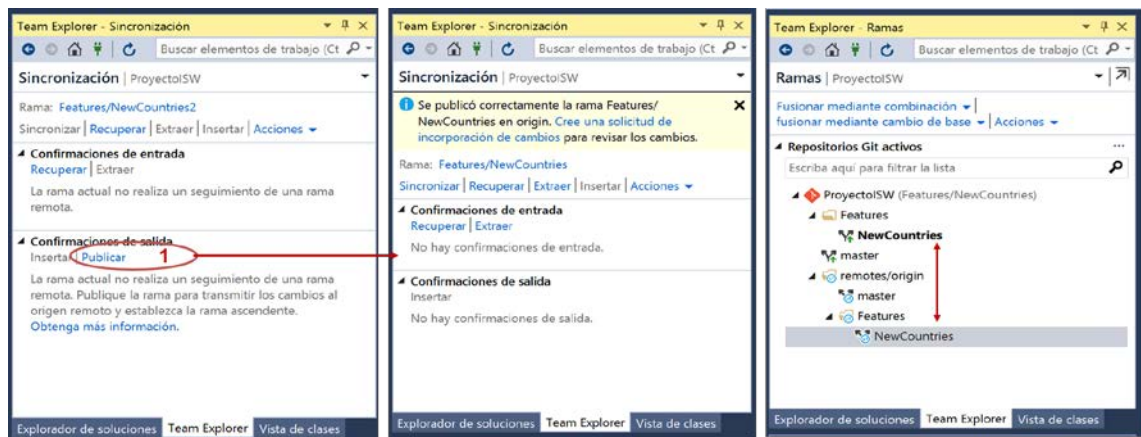


Figure 25. Push changes to the remote repository

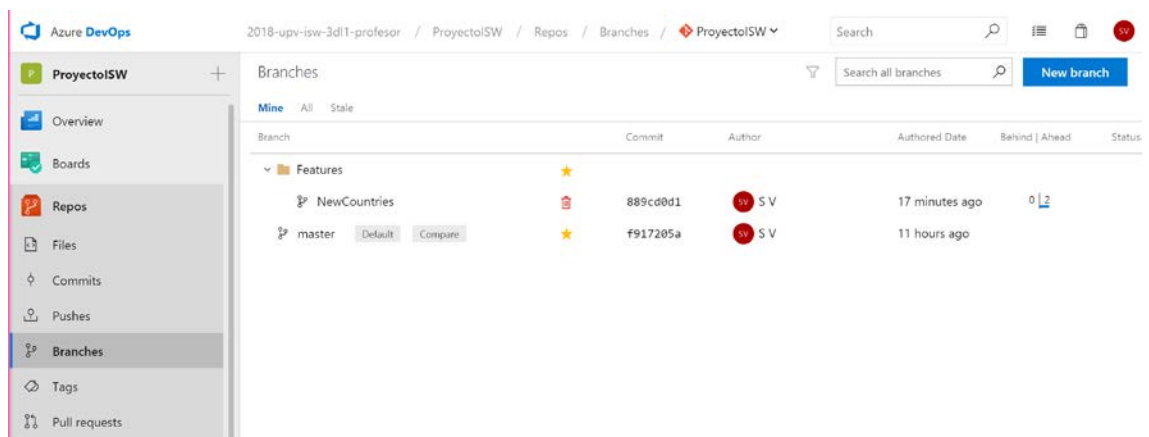


Figure 26. See branches in the Azure DevOps Web

6.3. Branches combination by merging

It is possible to merge two branches in a way that the changes of both are combined. This is useful to add the changes from a development branch into release branch or to combine two development branches. In this case, the commits made in the branch *Features/NewCountries* will be added to the main branch *master*.

- Select the branch *master* as the working branch, choosing the option *Team Explorer/Ramas* and then double click on *master* from your local repository (it becomes in bold, Figure 27-1).
- Next, click on the link *Fusionar mediante combinación* (Figure 27-2). A new page will be shown.
- In the new page, select from the dropdown *Fusionar mediante combinación de la rama* the branch *Featues/NewCountries* (Figure 27-3).
- Finally, click on the button *Fusionar mediante combinación* (Figure 27-4)

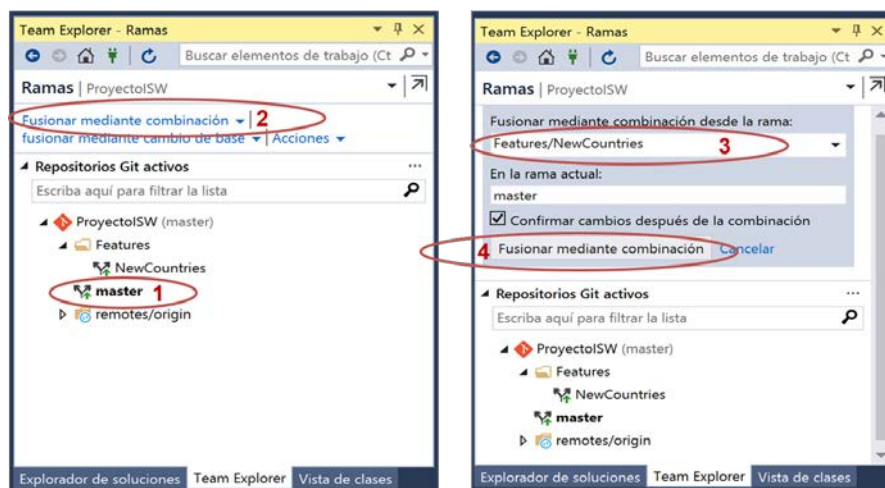


Figure 27. Merging branches

In this example there are no conflicts, but it is possible that some conflict occurs in this process. In that case, you should act as explained in section 5.2.

For more information on the version control model in branches in *Azure DevOps with Git* visit the information available in:

<https://docs.microsoft.com/es-es/azure/devops/repos/git/branch-policies-overview?view=vstsa>

Also, knowing about GitFlow workflow to work with branches can be a good idea for your team. There are several resources in Internet to know about it, but we can suggest you the book available online in the UPV Library: [Santacroce, F. \(2015\). Git Essentials. Olton Birmingham: Packt Publishing.](#)

NOTE: The development team should decide how many branches they want to have; only the main branch, the main and a development branch, etc.