



# 1. TiGL Workshop - How to contribute + Hands On

Jan Kleinert, Martin Siggel  
September 11./12. 2018, Cologne



Knowledge for Tomorrow

# TiGL is Open-Source

- TiGL is mainly developed at Airbus D&S, RISC and DLR
- But as TiGL is an Open-Source project, **anyone can contribute!**

## Three Ways to contribute:

1. Send **feedback** or **tell us about your work** in the TiGL ecosystem: [martin.siggel@dlr.de](mailto:martin.siggel@dlr.de), [jan.kleinert@dlr.de](mailto:jan.kleinert@dlr.de)
2. Report **issues** on <https://github.com/DLR-SC/tigl/issues>
3. Develop a feature and post a **pull-request**

Content of this session



Apache 2.0 License



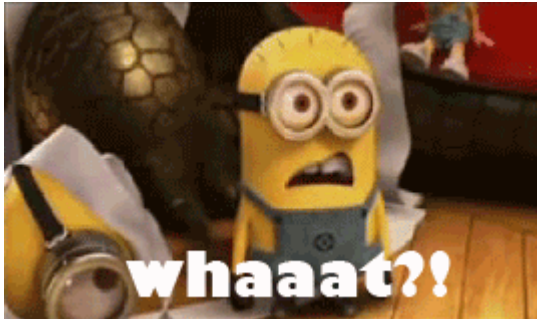
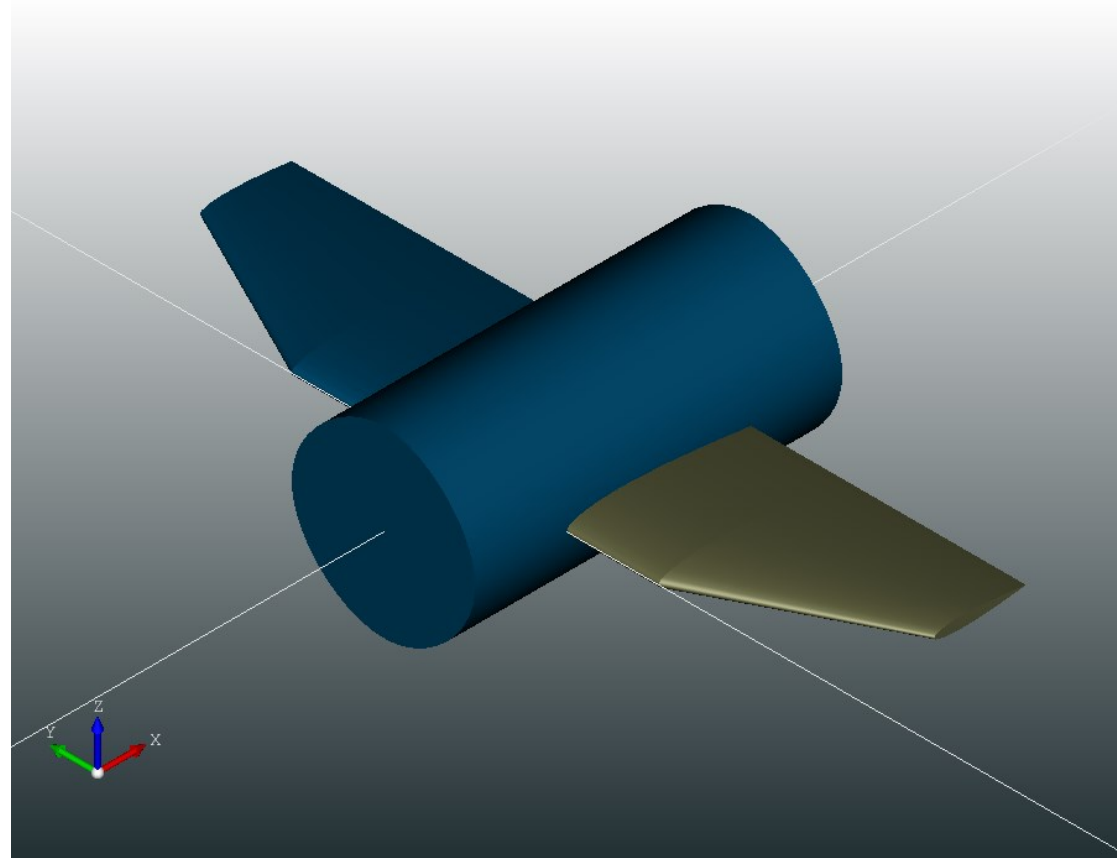
## Consider TiGL's “Hello World Script”

```
In [1]: from tigl3 import tigl3wrapper
        from tixi3 import tixi3wrapper

        tixi = tixi3wrapper.Tixi3()
        tigl = tigl3wrapper.Tigl3()

        tixi.open("simpletest.cpac.xml")
        tigl.open(tixi, '')

        nWings = tigl.getWingCount()
        print("\nThe airplane has {} wings.".format(nWings))
```



**We have found a bug!!**



- Luckily, the bug is not in the original TiGL repository, but only in my personal fork <https://github.com/joergbrech/tigl>
- Paul already spotted it and reported the bug a while ago using the issue tracker.
- Yet, the TiGL developers seem to be too busy to fix it ...



A screenshot of a GitHub issue page. The repository is 'joergbrech / tigl', forked from 'DLR-SC/tigl'. The issue title is 'Bug in tiglGetWingCount #1'. It was opened by 'pputin' 7 days ago. A comment from 'pputin' states: 'There seems to be an issue in the getWingCount function. No matter, which CPACS file I use, the function always returns a number of 42.' The issue has a 'bug' label added by 'joergbrech' 7 days ago. The right sidebar shows 'Assignees' (No one—assign yourself), 'Labels' (bug), and 'Projects'.

**It's time to take matters into our own hands!**



# Contents

## The TiGL Ecosystem

- TiGL architecture in a nut-shell
- 3rd-Party dependencies and useful tools
- Building TiGL from source

## Prerequisites

- Using Google Test for unit testing
- Basic GIT usage (fork, branch, commit, push, pull-request)

## Hands-On

- Write a Unit Test in our local TiGL build
- Fix a bug in your local TiGL build
- Create a Fork of the TiGL Github repository
- Push your local changes to your personal fork
- Share the changes by posting a Pull-Request on Github

## Final remarks

- Further ways to contribute
- Comments and further reading

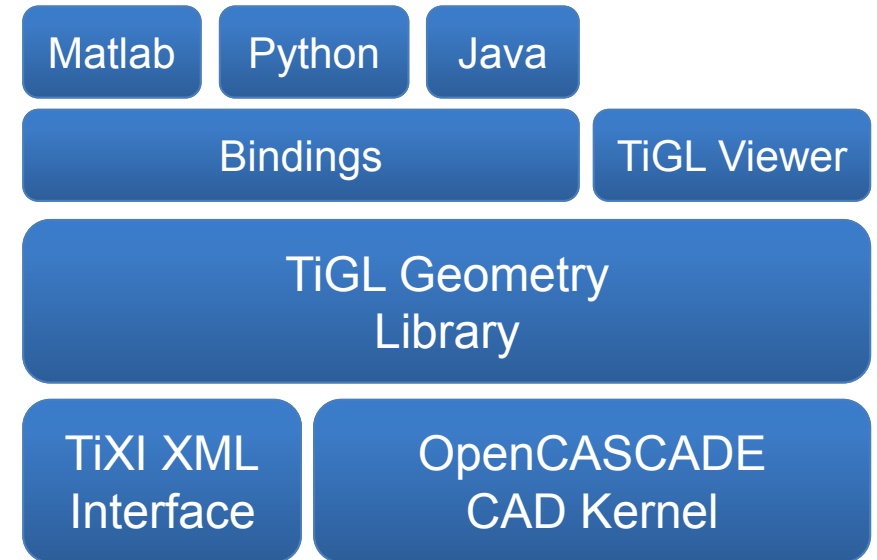


# The TiGL Ecosystem



# Architecture

- TiXI (<https://github.com/dlr-sc/tixi>)
  - Library to parse XML (CPACS) files
- OpenCASCADE (<https://www.opencascade.com/>)
  - Geometry (NURBS-based)
  - Topology (Boundary Representation)
  - CAD Exports, Visualization
- Language Bindings
  - Generated via SWIG (<http://www.swig.org/>)
  - Can access all C++ Data structures
- TiGL Viewer
  - 3D Visualization
  - Scripting
  - Debugging



# The TiGL SDK

- There are a **few requirements** and useful tools for building TiGL
  - some precompiled dependencies can be found here: <https://sourceforge.net/projects/tigl/files/Thirdparty/>

Dependencies	Comments	SDK
<b>OpenCascade</b>	<i>The CAD kernel behind TiGL</i>	6.8.0 (patched)
<b>TiXI</b>	<i>Used to parse the CPACS (.xml) files</i>	3.0.2
<b>Required tools</b>		
<b>C/C++ compiler</b>		MinGW 5.3.0
<b>CMake</b>	<i>Used to generate the Make files for compiling and linking</i>	3.10
<b>A make program</b>	<i>Used by CMake for compiling and linking</i>	Ninja
<b>Recommended</b>		
<b>Qt</b>	<i>Needed to build TiGLViewer</i>	5.11
<b>Doxygen</b>	<i>Needed to build the TiGL HTML Documentation</i>	1.8.13
<b>Git</b>	<i>Version control to keep your build synchronized with Github and to help contribute</i>	2.15
<b>IDE and debugger</b>	<i>Easier coding (auto-completion, dependencies, debugging ...)</i>	QtCreator, GDB debugger

- For convenience, we prepared a **Windows 32 Bit SDK** that includes all these tools
  - <https://sourceforge.net/projects/tigl/files/DevTools/TiGL-SDK-win32.7z/download>





# Building TiGL from Source

- For this Hands-On we assume that

- You downloaded the **TiGL SDK** and followed the instructions in the documentation, i.e.
  1. You **cloned the TiGL repository**
  2. You **built** and installed **TiGL** successfully
  3. You know how to **run the the unit-tests and TiGLViewer** from your local build

- You **installed the TiGL Python package** from the TiGL SDK MinGW Command Prompt via

```
$ conda create -n tigl_ws python=3.5 tigl3 jupyter pythreejs numpy -c dlr-sc  
$ activate tigl_ws
```

- If you have set up everything correctly, building TiGL should be as easy as pressing **Ctrl+B** in **QtCreator**  
or

```
$ mkdir build  
$ cd build  
$ cmake ../tigl/  
$ ninja install
```

*Assuming CMake finds the dependencies and  
CMAKE\_INSTALL\_PREFIX is set to a path with write  
permission*



# Prerequisites: Google Test and Git



# Google Test Framework in a very small nutshell

- **Basic Idea:** Create a separate executable, that tests small portions of your code for correctness.
- TiGL uses the GTEST framework from Google for unit testing.
- GTEST provides abstract classes and macros for ease of use. Example:

```
1  #include "myAirplaneGeometryTool.h"
2
3  #include "gtest/gtest.h"
4
5  TEST (wing, numberOfControlDevices) {
6      int numDevices;
7      ASSERT_TRUE ( GetNumberOfControlDevices(&numDevices) == ERRCODE_SUCCESS );
8      EXPECT_EQ( 3, numDevices );
9  }
10
11 TEST (wing, controlDeviceMaxDeflection) {
12     int wingIdx = 1;
13     int deviceIdx = 2;
14     EXPECT_NEAR ( 2.0, controlDeviceMaxDeflection(wingIdx, deviceIdx), 1e-10);
15 }
```



# Google Test Framework in a very small nutshell

- Test Fixtures are a way to execute custom code before running unit tests, e.g. reading an input file.
- A Test Fixture is a class defining the behavior before and after the tests.
- Use the TEST\_F macro to define a test for a test fixture

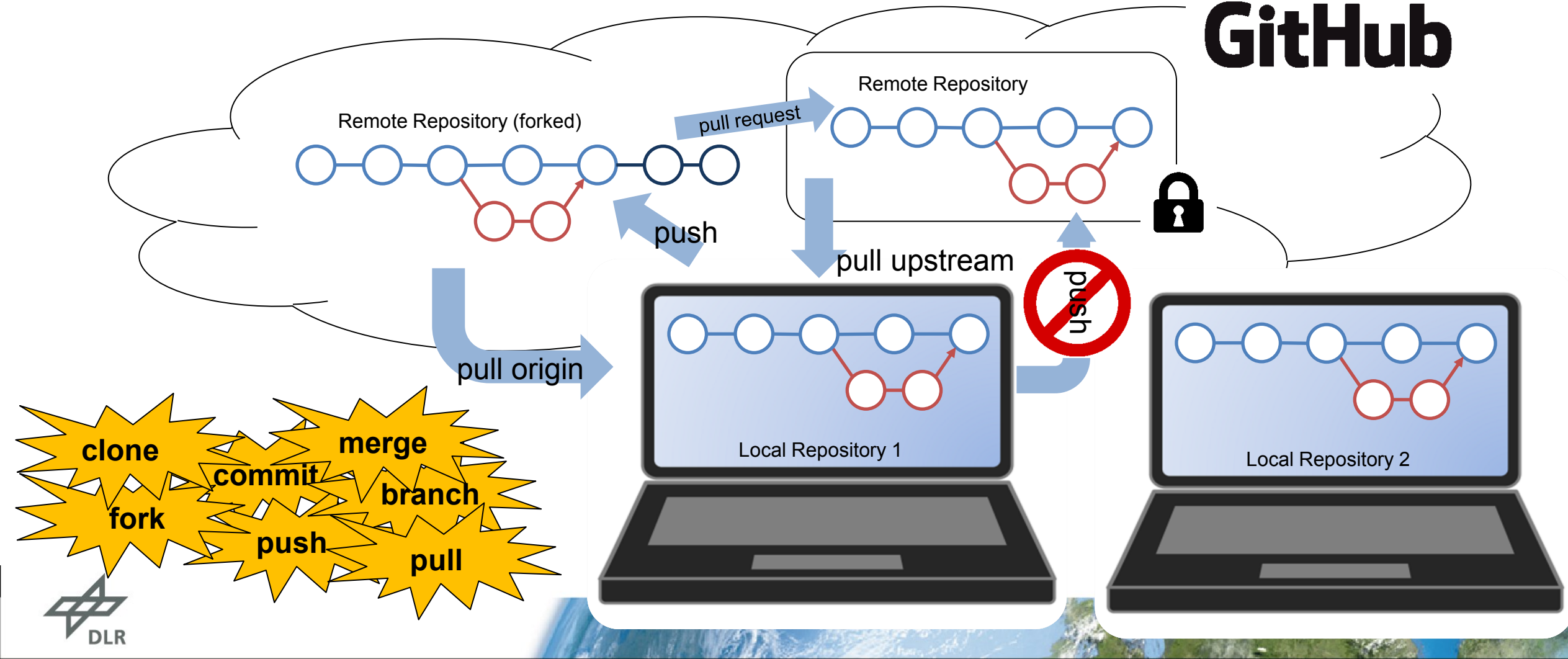
```
1  class GuideCurveTest : public ::testing::test {
2  public:
3      void SetUpTestCase() {
4          myAirplaneGeometryTool.open("guidecurvefile.xml");
5      }
6
7      void TearDownTestCase( ) {
8          myAirplaneGeometryTool.close();
9      }
10
11     // custom data members|
12 };
```

```
1  TEST_F(GuideCurveTest, TestSomething) {
2      // one unit test
3  }
4
5  TEST_F (GuideCurveTest, TestSomethingElse) {
6      // another unit test
7  }
```



# Basic Git and Github Usages

- Git is a version control system to help you keep track changes to your files
- Git assists you in distributed code development





# Hands-On

## Step 1: Preparation



## Preparation: Make sure you use the joergbrech/tigl fork on Github

Remember: The bug we want to fix is only in my fork joergbrech/tigl and not the official TiGLGithub repository

1. Open the SDK command prompt and navigate to your tigl repository

```
$ cd tigl
```

2. Change the “remote” url so that we synchronize with my buggy fork rather than the official repository

```
$ git remote set-url origin https://github.com/joergbrech/tigl.git
```

3. Verify that everything worked

```
$ git remote -v  
origin https://github.com/joergbrech/tigl.git (fetch)  
origin https://github.com/joergbrech/tigl.git (push)
```

4. Synchronize your local repository with the new remote

```
$ git pull
```



## Preparation: Enable the bug in you python package

5. Open QtCreator, rebuild and install TiGL (**Ctrl+B**). This will overwrite `TIGL_INSTALL_DIRECTORY/bin/libtigl3.dll` to include the bug
6. Replace the TiGL library used by the conda python package with our locally built one.
  - Navigate to `TIGL_SDK_DIRECTORY\tools\python3.6\envs\tigl_ws\Library\bin` and make a backup of the file `tigl.dll`
  - Now copy the file `TIGL_INSTALL_DIRECTORY/bin/libtigl3.dll` to `TIGL_SDK_DIRECTORY\tools\python3.6\envs\tigl_ws\Library\bin\tigl.dll`



## Preparation: Verify the bug

### 7. Verify the bug in Python

- Copy the file `TIGL_HOME\tests\unittests\TestData\simpletest.cpacs.xml` to a directory of your choice and create a script `tigl_bug.py` in the same directory with the following contents:

```
from tigl3 import tigl3wrapper
from tixi3 import tixi3wrapper

tixi = tixi3wrapper.Tixi3()
tigl = tigl3wrapper.Tigl3()

tixi.open("simpletest.cpacs.xml")
tigl.open(tixi, '')

nWings = tigl.getWingCount()
print("\nThe airplane has {} wings.".format(nWings))
```

Or download the two files from here:

<https://goo.gl/f237cw>

- Open the TiGL SDK MinGW Command Prompt, navigate to the directory of your python script and run it

```
$ python tigl_bug.py
```

```
The airplane has 42 wings.
```



# Hands-On

## Step 2: Write a Unit Test





## Creating a git branch for the bug fix

1. We do not want to develop in the main development branch `cpacs_3`. Therefore, we will create a new branch with a reasonable name (e.g. include your initials, the Github number of Paul's issue and a keyword describing the bug).

- Navigate to the directory of the tigl source and enter

```
$ git branch jk_1_wingCountBug  
$ git checkout jk_1_wingCountBug
```

- Make sure you are on the correct branch, by checking the git status

```
$ git status
```



## Unit Testing: Write a unit test that checks for the bug

2. To automatically test for the bug after we fixed it, we will write a unit test. Open QtCreator.

- Before we create a new .cpp file in `TIGL_HOME\tests`, let us first check if there are already unit tests for the function `tiglGetWingCount`. To do so, press **Ctrl+Shift+F** in QtCreator to open the advanced search. Search for `tiglGetWingCount` in the entire project

- We should keep all unit tests for `tiglGetWingCount` in the same place, so we will add a unit test to `TIGL_HOME\tests\unittests\tiglWingSegment.cpp` some where near line 173.



## Unit Testing: Write a unit test that checks for the bug

- Look at the other unit tests for `tiglGetWingCount`. They are all based on the class `WingSegment`:

```

166  /**
167   * Tests tiglGetWingCount with null pointer argument.
168   */
169  TEST_F(WingSegment, tiglGetWingCount_nullPointerArgument)
170  {
171      ASSERT_TRUE(tiglGetWingCount(tiglHandle, NULL) == TIGL_NULL_POINTER);
172  }
173

```

- A look at the class `WingSegment` in the same file (**Ctrl+Click**) reveals in the member function `SetUpTestCase()` that the file "`TestData/CPACS_30_D150.xml`", is opened. This is an airplane configuration with three wings (*one wing and two tailplanes, the mirrored wings are not counted*)
- Write a unit test that checks if this wings are counted correctly using `ASSERT_EQ` and/or `ASSERT_TRUE`.

```

174  /**
175   * Tests successfull call of tiglWingGetWingCount.
176   */
177  TEST_F(WingSegment, tiglGetWingCount_success)
178  {
179      //Your Code here
180  }

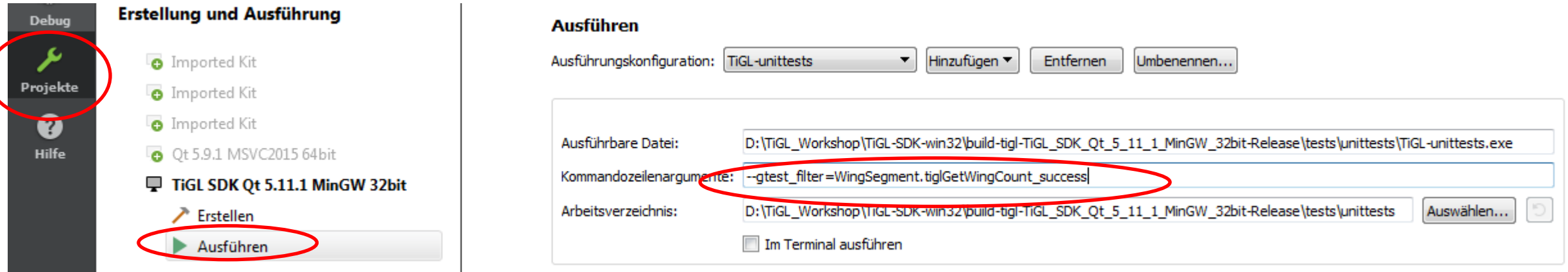
```

If you are unsure how to do this, take a look at a similar unit test for the function `tiglwingGetSegmentCount` in the same file.



# Unit Testing: Write a unit test that checks for the bug

- Let's see if our unit test fails. First, configure your QtCreator project to only run the newly written test:



The screenshot shows the Qt Creator IDE. On the left, the 'Projekt' (Project) icon is circled in red. The 'Erstellung und Ausführung' (Creation and Execution) panel shows the project 'TiGL SDK Qt 5.11.1 MinGW 32bit' selected, with the 'Ausführen' (Run) button circled in red. On the right, the 'Ausführen' (Run) configuration window is open. The 'Ausführbare Datei' (Executable) is set to 'D:\TiGL\_Workshop\TiGL-SDK-win32\build-tigl-TiGL\_SDK\_Qt\_5\_11\_1\_MinGW\_32bit-Release\tests\unittests\TiGL-unittests.exe'. The 'Kommandozeilenargumente' (Command line arguments) field is circled in red and contains '--gtest\_filter=WingSegment.tiglGetWingCount\_success'. The 'Arbeitsverzeichnis' (Working directory) is set to 'D:\TiGL\_Workshop\TiGL-SDK-win32\build-tigl-TiGL\_SDK\_Qt\_5\_11\_1\_MinGW\_32bit-Release\tests\unittests'.

- Press **Ctrl+R** to build, install and run the code. We expect it to fail, the bug has not been fixed yet:

```
[-----] 1 test from WingSegment
[ RUN      ] WingSegment.tiglGetWingCount_success
D:/TiGL_Workshop/TiGL-SDK-win32/tigl/tests/unittests/tiglWingSegment.cpp:181: Failure
Value of: wingCount == 3
    Actual: false
Expected: true
D:/TiGL_Workshop/TiGL-SDK-win32/tigl/tests/unittests/tiglWingSegment.cpp:181: Failure
Value of: wingCount == 3
    Actual: false
Expected: true
[  FAILED  ] WingSegment.tiglGetWingCount_success (0 ms)
[-----] 1 test from WingSegment (0 ms total)
```



# Hands-On

## Step 3: Fix the bug





## Fix the bug

3. There is a fairly easy to spot bug in `tiglGetWingCount`. (For the cheaters: <https://goo.gl/tun2pW>)

- Fix it!
- Rerun the unit test (**Ctrl+R**). This time we expect it to succeed:

```
Note: Google Test filter = WingSegment.tiglGetWingCount_success
[=====] Running 1 test from 1 test case.
[-----] Global test environment set-up.
[-----] 1 test from WingSegment
[ RUN      ] WingSegment.tiglGetWingCount_success
[          OK ] WingSegment.tiglGetWingCount_success (0 ms)
[-----] 1 test from WingSegment (0 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test case ran. (104 ms total)
[ PASSED  ] 1 test.
```

- To be absolutely sure that everything works now, go ahead and copy the file `TIGL_INSTALL_DIRECTORY/bin/libtigl3.dll` to `TIGL_SDK_DIRECTORY\tools\python3.6\envs\tigl_ws\Library\bin\tigl.dll` and rerun your python script:

```
$ python tigl_bug.py
```

```
The airplane has 1 wings.
```



## Commit your changes locally

- You want to commit the changes to your newly created branch. Navigate to your tigl directory and enter **git status**:

```
$ git status
On branch jk_1_wingCountBug
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   src/api/tigl.cpp
        modified:   tests/unittests/tiglwingSegment.cpp
```

- Add the files you would like to commit

```
$ git add src\api\tigl.cpp
$ git add tests\unittests\tiglwingSegment.cpp
```

- Commit the changes using a speaking commit message

```
$ git commit -m "fix a bug in tiglGetWingCount, fixes issue #1"
```

*The hashtag in front of the issue number is interpreted as a reference on Github.*

# Good job!

**We are well on our way!** So far you have:

1. Verified the bug in Python
2. Created a unit test
3. Fixed the bug
4. Verified that the bug is fixed using the unit test and your Python script
5. Committed the bugfix locally to your version control system (i.e. git).

But so far you are the only one profiting from your bug fix. **It's time to share it with the community.**



# Hands-On

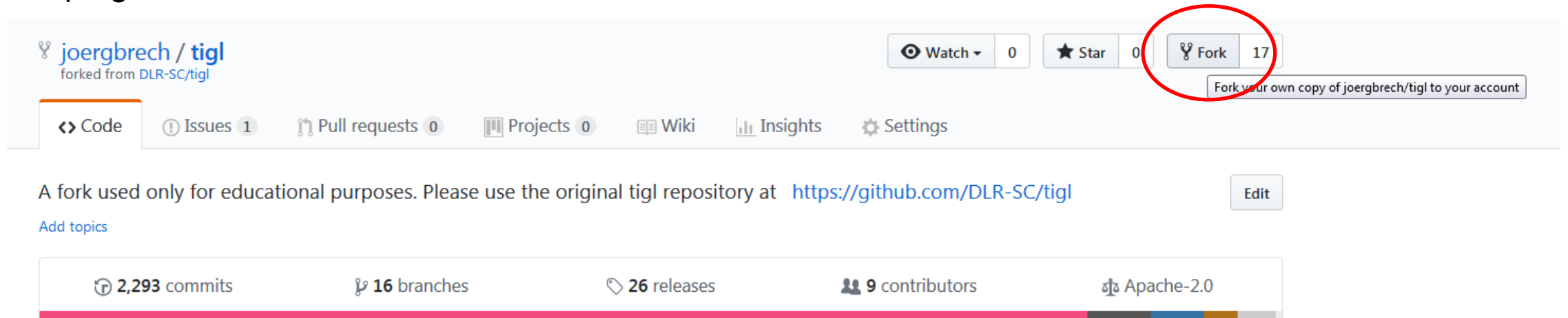
## Step 4: Share your changes on Github



## Create a fork of the Github repo to which you want to contribute

4. You want to push your local commit to the Github repository. You do not have any rights to push to the original repository, so you must first create a personal fork to which you can push your changes.

- Log into Github
- Navigate to the Github repository <https://github.com/joergbrech/tigl> and press the „**Fork**“ button on the top right:



The screenshot shows the GitHub repository page for `joergbrech / tigl`, which is forked from `DLR-SC/tigl`. The repository has 0 stars and 17 forks. The `Fork` button is circled in red. Below the repository name, there are tabs for `Code`, `Issues` (1), `Pull requests` (0), `Projects` (0), `Wiki`, `Insights`, and `Settings`. A message states: "A fork used only for educational purposes. Please use the original tigl repository at <https://github.com/DLR-SC/tigl>". Below this, there is a bar showing repository statistics: 2,293 commits, 16 branches, 26 releases, 9 contributors, and Apache-2.0 license.



# Change the remote of your local repository (again) and push your commit

- You want to make sure that you push your local changes to your personal tigl fork. So navigate to your tigl directory and enter

```
$ git remote set-url origin https://github.com/YOURUSERNAME/tigl.git
```

- Verify that everything worked:

```
$ git remote -v  
origin https://github.com/YOURUSERNAME/tigl.git (fetch)  
origin https://github.com/YOURUSERNAME/tigl.git (push)
```

- Your local branch does not exist in your github fork. You can create the remote branch and push your changes to it by entering

```
$ git push --set-upstream origin jk_1_wingCountBug
```

- Enter your Github credentials.

Replace *YOURUSERNAME* with your Github user name and *jk\_1\_wingCountBug* with the name of your local branch.

# Post a Pull-Request

- Take a look at the TiGL fork in your Github account. You will see that you committed a new branch and can now post a pull-request



2,293 commits   16 branches   26 releases   9 contributors   Apache-2.0

Your recently pushed branches:

jk\_1\_wingCountBug (less than a minute ago)

Compare & pull request

- Press the button “**Compare & pull request**” to ask the maintainers (in this case joergbrech) to merge your changes into the main development branch. You can comment on your changes and press the button “**Create Pull Request**”

**CONGRATULATIONS! YOU ARE DONE! IT IS UP TO THE CODE MAINTAINERS TO REVIEW YOUR CHANGES AND MERGE THEM TO THE MAIN DEVELOPMENT BRANCH.**

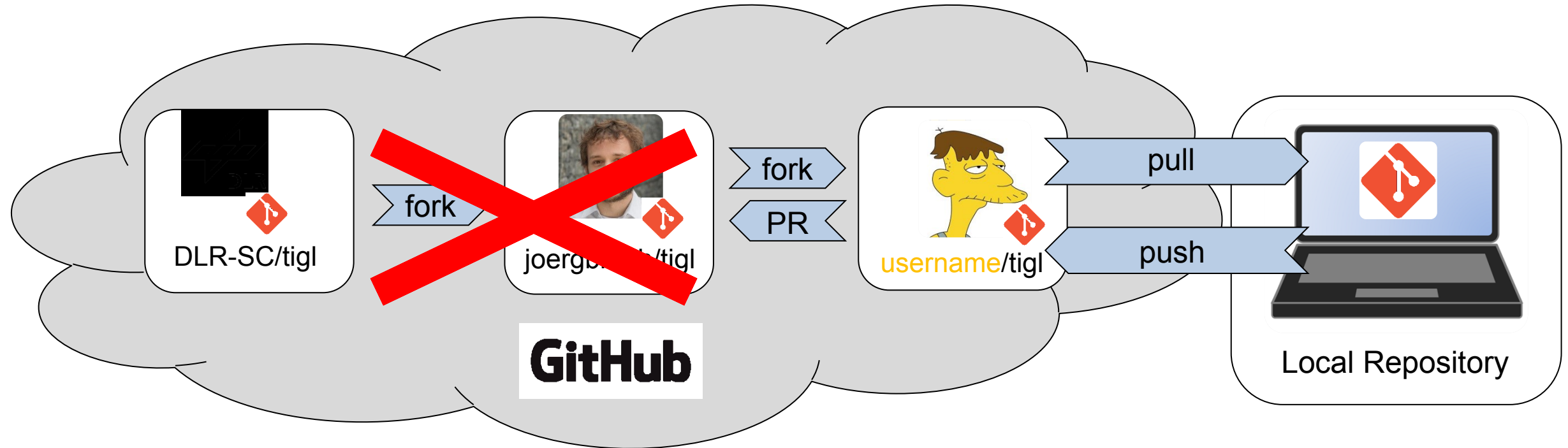


# Final Remarks



## Next steps

- In this hands-on we used an intermediate TiGL fork joergbrech/tigl as an example.



- Let us keep our repository synchronized with the **official TiGL Github repository**

## Next steps

**Optional first step:** Delete your fork and create a new one, directly from the DLR-SC repository

1. Set the upstream repository to be DLR-SC/tigl:

```
$ git remote add upstream https://github.com/DLR-SC/tigl.git
```

2. Verify that everything worked:

```
$ git remote -v  
origin https://github.com/YOURUSERNAME/tigl.git (fetch)  
origin https://github.com/YOURUSERNAME/tigl.git (push)  
upstream https://github.com/DLR-SC/tigl.git (fetch)  
upstream https://github.com/DLR-SC/tigl.git (push)
```

3. To fetch all changes of the official TiGL repository type

```
$ git fetch upstream
```

4. And finally, to keep your local cpacs\_3 (or any other) branch synchronized with the official TiGL release:

```
$ git checkout cpacs_3  
$ git merge upstream/cpacs_3
```



## Summary: The Contributing Workflow

1. **Fetch changes** from DLR-SC/tigl to your local repository **and synchronize** the main development branch

```
$ git fetch upstream  
$ git checkout cpacs_3  
$ git merge upstream/cpacs_3
```

2. **Create a branch** for your feature and start coding. Don't forget to commit your changes!

```
$ git branch my_fancy_tigl_feature  
$ git checkout my_fancy_tigl_feature  
# CREATE SOME AWESOME CODE  
$ git add FILE1.cpp FILE2.cpp  
$ git commit -m "Implemented a fancy feature"
```

3. **Push your changes** to your Github Fork

```
$ git push
```

4. **Post a Pull-Request on Github.** *Note that you can choose any Github Fork of DLR-SC/tigl to which you would like to post the pull request.*





**Congratulations!**  
**You are now a TiGL developer.**



## Further Ways to Contribute: Adaptations to CPACS

- TiGL contains a class/enum for every **geometry-related** CPACS object in the schema.
- Source Code for abstract classes for these CPACS objects are automatically generated by **CPACSGen**
- The TiGL behavior of these objects is implemented (manually) in derived classes
- **CPACSGen** (short for CPACS generator) is developed by **RISC**
- It creates source code for classes and enums from the types defined in a CPACS XML schema file for the TiGL library
  - [https://github.com/RISCSoftware/cpacs\\_tigl\\_gen](https://github.com/RISCSoftware/cpacs_tigl_gen)
- The .h and .cpp files in src/generated are created using **CPACSGen**
- If you make adaptations to the CPACS schema, the code must be regenerated, derived classes must be adapted or created
- Don't forget to discuss your CPACS adaptations with the CPACS maintainers at DLR-SL
  - <http://www.cpacs.de/>
  - <https://github.com/DLR-LY/CPACS>



## Further Ways to Contribute: cpacs2to3

- CPACS 3 is released and we have a release candidate for TiGL 3.
- **TiGL 3 is NOT backwards-compatible to TiGL 2!**
  - There are some major changes in regard to the component segment coordinates and guide curve definitions
  - For the transition from CPACS 2 to CPACS 3 we started developing a converter for cpacs files, hosted on Github: <https://github.com/DLR-SC/cpacs2to3>

**Feel free to contribute!**



## Concluding remarks

- We do not expect you to do our bugfixes for us!
- But please do:
  1. Let us know what you are working on
  2. Report issues, post pull-requests
  3. Don't hesitate to contact us if you have any questions

### Additional Information:

- ✓ [Google Test Primer https://goo.gl/mdYNvp](https://goo.gl/mdYNvp)
- ✓ [Standard Github Forking https://goo.gl/FXE5ye](https://goo.gl/FXE5ye)
- ✓ [TiGL Programmer's Guide https://goo.gl/oBjVpg](https://goo.gl/oBjVpg)
- ✓ [OpenCascade CheatSheet https://goo.gl/zZ9nrD](https://goo.gl/zZ9nrD)



# Questions



jan.kleinert@dlr.de  
martin.siggel@dlr.de

