**Lab Exercise: Git Basics – the Scheduler**
In this exercise you will extend a class `Scheduler` from a handed-out project. The problem of coding the classes is minimal, so focus should be on version control and working in parallel.

**Prerequisites:**
It is a prerequisite for this exercise that…

- You have installed the Git Tools – this could be MsysGit + TortoiseGit. Check the web an Blackboard on where and how to do this.

- You have joined a team of 3-4 students. This exercise is solved in your I4SWT teams!

If you have not done one or both of these things, do this before you start the exercise.

---

A *scheduler* is the key part of an operating system which decides which thread is run next. In this exercise we will create a ridiculously simple scheduler, which models threads by their names (a `string`) only.

You have been provided with a handout for this exercise. This contains a clas Scheduler with two methods implemented: A constructor and Spawn(). The details of the handed-out implementation can be found below:

| | |
|---|---|
| `Scheduler` | Constructor. Initializes an empty `Scheduler` |
| `Spawn(string name, Priority pri)` | Spawn a new thread with the given name and priority |
| `Priority` | An enum. Can be either `LOW`, `MED` or `HIGH` |

Your task is twofold:

- Exercise 1: Get the hand-out under version control, so all team members can have a clone and you can work in parallel

- Exercise 2: Extend this implementation – and the unit test suite, of course!

See the next pages for the exercises.

**Exercise 1: Get the hand-out under version control**
A bare Git repository for this exercise is set up for each team. It is located at gitswat.ase.au.dk and can be accessed using ssh for read/write access at

```
team152xx@gitswat.ase.au.dk:repoyy.git
```

where *xx* is your team number and *yy* is one of the repositories available for your team

1. One (1) team member shall clone the bare repository mentioned in Exercise 3 above to a folder on his local machine. This will generate a folder called repo*yy* on his/her computer.

2. Then, this team member shall copy the hand-out for this exercise to the folder `repo`*yy*.

3. Then, this team member shall verify that he can build the solution and run the tests in the hand-out from the folder `repo`*yy*.

4. Then, this team member shall *add* and *commit* the solution files to the repository. This is easily done from within Visual Studio: Right-click the solution, choose "Add to version control", select Git if prompted.

5. Then, this team member shall *push* the files from his local repository to the distributed Git repository.

6. Then, all other team members shall clone the repository (see 1. above) and verify that they can build the solution and run the tests.

**Do not proceed with the remainder of this exercise before all team members can clone the repository, build the software and run the tests.**

## Exercise 2: Extend the implementation and test suite

Your implementation of the Scheduler shall be extended with the following methods and properties:

| | |
|---|---|
| `Schedule()` | Schedule the next highest-priority thread |
| `ActiveThread` | Property. Name of the active thread |
| `NThreads` | Property. Number of threads in scheduler |

The detailed requirements are listed below:

- The scheduler must maintain a list of threads for each priority.
- When `Spawn()` is called, the thread should be added to the end of the list of threads of the pertaining priority.  If a thread of the same name already exists, an exception should be thrown.
- When `Schedule()` is called, the scheduler must find the next highest-priority thread to run and make it active. If no thread exists for the scheduler, an exception should be thrown.
- When `ActiveThread` is accessed, the scheduler must return the name of the active thread. If no active thread exists for the scheduler, an exception should be thrown.

As a team, implement the class `Scheduler`, the necessary exceptions and the tests for it. As you do, adhere to the guidelines for structuring and naming unit tests set forth in TAOUT sections 7.4 and 8.

The way to do this is to briefly discuss the features, then divide up into two sub-teams of 1-2 each. Work in parallel to get the features implemented.

It is *highly* recommended that you implement one feature (i.e. one method or property) at a time and test it to completion before you continue with another. It may not always be possible, but try as best you can. Remember to commit your work early and often!