Workshop: Test-driven development for academic simulation

This workshop is about test-driven development (TDD) in the context of designing simulations for academic publication. TDD is a technique, that is conceptualised to speed up development speed while at the same time providing quality control. The examples are chosen to provide a meaningful context such that the participant is enabled to further use this technique in their academic endeavours.

To get started with the workshop, clone the repository
https://github.com/Monderkamp/testWorkshop
or alternatively by clicking the green button “code” and then “download as “.zip”.
The code, You are going to use is contained within the folder “code”.

The content of he workshop focusses on developing a simple intersection criterion for finite line segments, represented within a LineSegment class which is contained within LineSegment.hpp and LineSegment.cpp, respectively. Due to heavy use of vector algebra, the header file contains a class Vector3D. Even though the line segments are expected to be purely 2d within the context of the workshop, the code accommodates for three dimensions.
This is, because the intersection criterion for finite line segments can be written as a vector-algebraic expression, featuring 3d cross- and dot-products.

The file testLineSegment.cpp contains a framework for unit-testing the implementation, along with several examples, which in turn utilise the test-toolkit provided by unitTest.hpp.

You are going to develop within LineSegment.hpp, -.cpp and testLineSegment.cpp. You can compile and run the code in windows with compileandrun_unitTests.bat.
Since we are developing individual member functions, and not a whole simulation protocol, there is no simulationMain.cpp given.

If You prefer to work in python, the corresponding code is given in the python folder.
This code utilises the python-library unit test. The tests are defined within classes. The tests are run by invoking unittest.main().

For Questions, feedback and critique, feel free to address me (see below).
gl&hf

Paul

Paul A. Monderkamp, paul.monderkamp@gmail.com

Exercise 1 – Vector Algebra: Dot product

The goal of this exercise is, to get familiar with foundational level unit-tests and the workflow of test-driven development in a mathematical context, which is very familiar tot he participant.

Due to the simplicity of the fomulae, it will be easy to check fort he participants, whether they correctly implemented the test.

Since the workshop provides sample code, for C++ as well as python, the exercises are unspecific towards the implementation language. For syntax, consult the sample code, or ask Your instructor.

a) Shell-Implementation for the dot product for the class Vector3D

Set up a Shell-Implementation (i.e. executable code, that returns generic and preferably wrong results).

b) Write unit tests

Think of corresponding unit tests for the dot product. Consider important edge cases.
Set up the unit tests in the corresponding code file.
The best amount of tests is the minimal amount of tests, that you would need to guarantee that the implementation is correct, if all tests are passed.

c) Run the tests and make sure they fail

d) Implement the corresponding code

Implement the corresponding code. If You think You have a solution, run the unit tests o check for errors. If your tests pass, you are finished with this exercise. If not, repeat.

Aufgabe 2 – Vector Algebra: Cross product

Repeat Exercise 1 with the cross product.

Paul A. Monderkamp, paul.monderkamp@gmail.com

Aufgabe 3 – Wert der Option

The goal of this exercise is, to appreciate the usefulness of test-driven development in a setting, where the code is sufficiently complicated

Since the workshop provides sample code, for C++ as well as python, the exercises are unspecific towards the implementation language. For syntax, consult the sample code, or ask Your instructor.

a) Shell-Implementation of intersect function

Set up a shell-implementation of a member function of LineSegment that takes another LineSegment as argument. In C++ it may be useful to parse a reference to save computation speed.
The intersection function should return a Boolean.

b) Write unit tests

Think of important test cases to test for intersection of two line segments. Identical line segments should not be considered intersection. This is, such that in an application in a simulation code, a line segment must not consider itself explicitly, when checking whether it intersects with other line segments in, e.g., a cell list.

c) Run all unit tests and make sure they fail

The shell implementation in principle has to return either Boolean value to be executable. We should consider true and false cases for the tests. Therefore some of the tests are going to pass even with the shell-implementation. This is normal. Any tests failed, should be considered a general failure of the test cases.

d) Implement the corresponding code

Since this is a workshop on unit tests and not on vector algebra, the solution to the geometric problem is given in the repository. You may refrain from implementing this solution if you want to solve the geometrical problem yourself.