Introduction to Testing and NUnit

Cpt S 321

Washington State University

Testing – what is this all about?

- **Testing**
  - Finding inputs that cause the software to fail.
  - Concentrates on the product (i.e., the software); testing versus Quality Assurance (QA).
  - Automated, repeatable, systematic
  - Exhaustive testing:
    - Can we do it?
    - Do we want to do it?
- Dijkstra, 1972: "Program testing can be used to show the presence of bugs, but never to show their absence"
- No absolute certainty can be gained from testing
- Must be integrated with other activities such as code reviews and quality assurance. More on code reviews later in the semester.

Testing – what does it take?

- Oracle, i.e., the gold set: a set of data to compare the output against
- Specification (when available)
- Implementation (when available)
- Different types of testing:
  - Ex. *white-box* versus *black-box*
  - Ex. *unit testing* versus *integration testing* versus *system testing*

Three simple steps when testing

1. **Setup** anything that needs to be setup
   - Not always necessary
   - Can be done in the 'setup' method if the setting is common to all tests or in the beginning of the test otherwise
2. **Call the method** that you want to test
3. **Check if the results** obtained from calling the method match the expected results.
   We use **assertion statements** (or **assertions**) for that.

Note: Steps 1 and 2 are typically what you would do in a normal program.

What are assertions?

- The Classic Model allows you to use the assert statements defined in the NUnit.Framework.Assert class, for example:

  - Assert.True
  - Assert.False
  - Assert.Null
  - Assert.NotNull
  - Assert.IsNotEmpty
  - Assert.AreEqual
  - Assert.AreNotEqual
  - Assert.AreSame
  - Assert.AreNotSame
  - Assert.Contains
  - Assert.Greater
  - Assert.Less
  - Assert.IsInstanceOf
  - Assert.IsNotInstanceOf
  - Assert.Throws
  - …

Note: See also StringAssert, CollectionAssert, FileAssert, DirectoryAssert classes.

How does a test file look like?

Attributes that signal what the entities are to NUnit

```
5   namespace HelloWorld.Math.Tests
6   {
7       using NUnit.Framework;
8
9       [TestFixture]
        0 references
10      public class MathTest
11      {
12          [SetUp]
            0 references
13          public void Setup()
14          {
15          }
16
17          [Test]
            0 references
18          public void TestAddUsingTheClassicModel()
19          {
20              // positive numbers (normal case):
21              Assert.AreEqual(
22                  2, // expected value
23                  Math.Add(1, 1)); // actual value obtained as a result of the method call
```

Target method

Assertions (cont.)

- Constraint Model
  - Using the **Assert.That** method
  - And specifying constraints
    - Ex.: Assert.That(myString, Is.EqualTo("Hello"));
- Examples of constraints:
  - Is.EqualTo
  - Is.Not.EqualTo
  - Is.GreaterThanOrEqualTo
  - Is.All.InstanceOf<Int32>
  - Is.Ordered.Ascending
  - …
- Check here for a quick set of examples

Note how the actual value is swapped (i.e., first parameter) compared to the Classic model

NUnit - let's get started (Tasks 1-6)

- VS 2022 and JetBrains Rider come with NUnit.
  (If you are using VS 2019, please check the installation manual on Canvas.)

1. Create a new NUnit Test Project **HelloWorldTests** in the same solution as our HelloWorld project.

2. We want to test a Math class in your existing project **HelloWorld** that we will implement in **namespace HelloWorld.Math**. To this end, create the class Math and method Add. Don't worry about the algorithm yet:

```
public static int Add(int a, int b){return 0;}
```

3. Create a new test file in project **HelloWorldTests** and call it **MathTest**; define it in namespace **HelloWorld.Math.Tests**. (You can use the example from the slides as a starting point.)

4. Add the HelloWorld project as a reference to your HelloWorldTests.
   (Option 1: Expand HelloWorldTests and right-click on Dependencies -> Add Project Reference…
   Option 2: Right-click on project HelloWorldTests -> Add -> Project Reference…)

5. Create a test method for method Add; call it **TestAdd** and add assertion statements to test method Add. **What should we test?**

6. **Run the tests**: they should **ALL (or almost all) FAIL!**

What did just happen?

- **We created the skeletal code** for a class **Math** and a method **Add** **without implementing any algorithm yet**.
  - The method body could/should have been empty if the return type was void.
  - When there is a return type other than void, some sort of default implementation is needed in order for the code to **compile**.

- We created a class **MathTest** that we will use to test class **Math**.

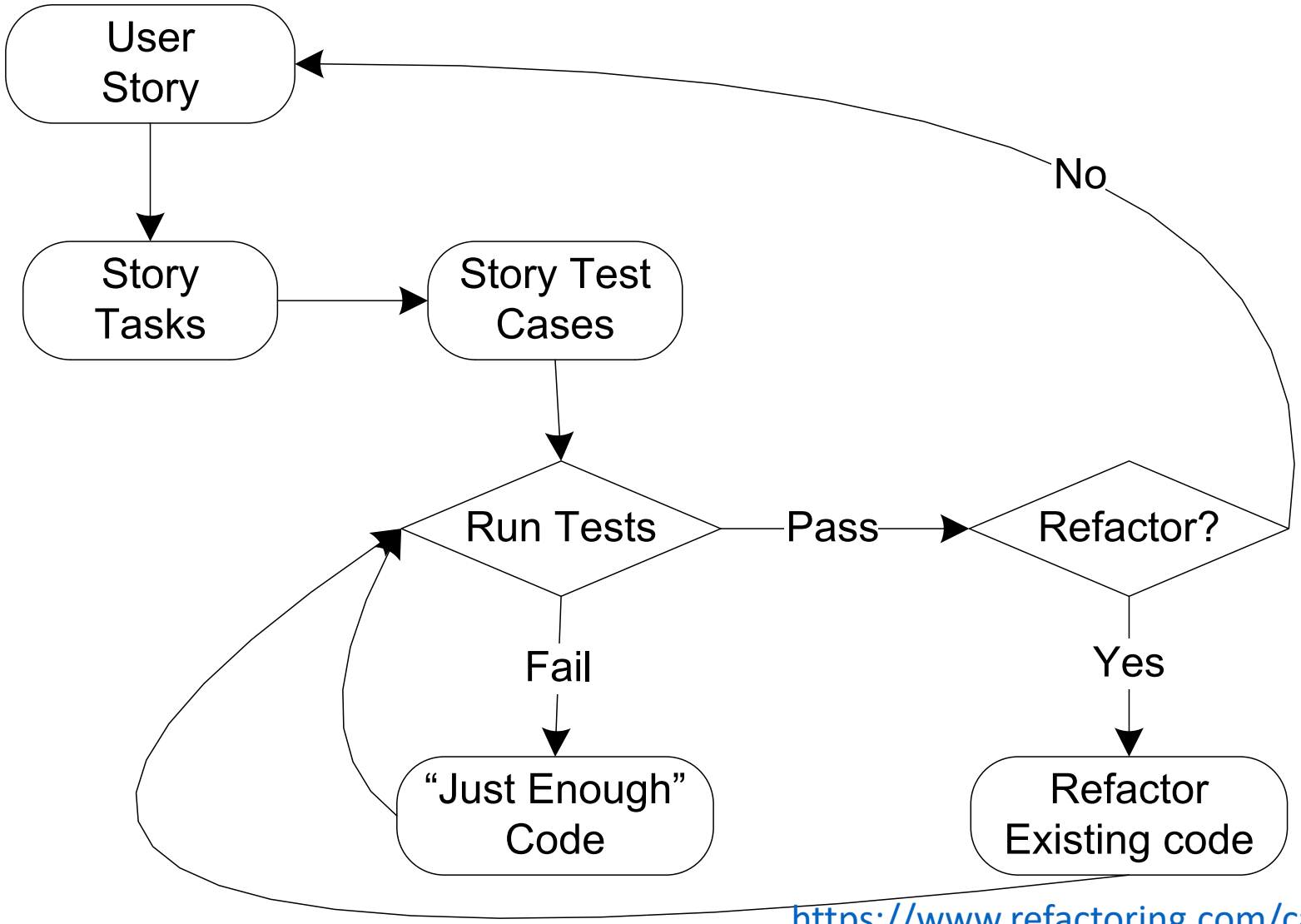
- We created a method **TestAdd** to test method **Add**.

- **We tested** method Add! I.e., we proved that the method is not working properly!

- Next, we can start **working on the implementation** of method Add **until our tests pass**.

- This process is called **Test Driven Development (TDD)**

Activities Overview in TDD



https://www.refactoring.com/catalog/

TDD in this class

- For ALL HWs :
  1. Write skeletal code and tests.
     The tests will (should) FAIL!
  2. **Commit skeletal code and tests**
  3. Write implementation (i.e., the algorithms). ALL tests should PASS!
  4. **Commit the implementation**
  5. Refactor code if needed and **commit again**. The number of passing tests should be the same, i.e., refactoring should not break your code!
  6. Add more tests if needed and **commit again**.
  7. If the tests in 6 do not pass, keep working on the implementation and **commit again.**

=> We need to see at least 2 commits per feature/task that you decide to tackle.

Iterate **multiple times over 3-5** if a feature is complex. In that case, **in every iteration more tests should be passing**!

Let's practice TDD (aka: in-class coding exercise!***)

- Implement method **IsLeap** in class **Date** by following TDD. Recall that:
  - A year that is divisible by 4 is a leap year
  - A year not divisible by 4 is a common year
  - A century year not divisible by 400 is a common year
  - A century year divisible by 400 is a leap year

- Remember:
  - **Use the process from the previous slide!**

***Note: Don't forget to tag the last commit for this in-class exercise if you want bonus points for it.

White Box Testing and code coverage in VS

- VS Enterprise Edition reports coverage
  - Check the documentation
- Rider has a Coverage Window
  - Check documentation here

- What is the coverage of our tests?
  - Date.IsLeap?
  - Math.Add?

- What is a good coverage?
  - As high as possible (>80%)

What if we don’t have a coverage tool?

- For this course we will not require but we highly recommend the use of a coverage tool when your IDE supports it.

- When designing tests, we will use the following rules:
  - **Normal cases**: To test for normal cases, think about what are inputs and outputs that will show that the program is behaving as expected under **normal conditions**.
  - **Edge cases:** To test for edge cases, think about what are boundary values for inputs and outputs that will show that the program is behaving as expected under **normal conditions**.
  - **Exceptional cases:** To test for exceptional cases, think about what inputs will make the program crash and check if it is indeed crashing.

Announcements

- HW2 has been posted (or it will be as soon as we reach this slide in class):
  - Have fun!
  - Don’t forget to use TDD for it!