Unit Testing

Course Software Testing & Verification

2024/25

Wishnu Prasetya & Gabriele Keller

Plan

- What is unit testing, and why we do it?
- Some essentials on unit testing:
  - Unit testing in C#
  - Specifying (and testing) with pre- and post-conditions
  - Mocking
  - Other types of specifications: classinv, ADT, FSM.

Note: The subject of unit testing is only glossed over in A&O. Since unit testing plays an important role in software engineering nowadays, in this lecture we will spend a bit more time to discuss it.

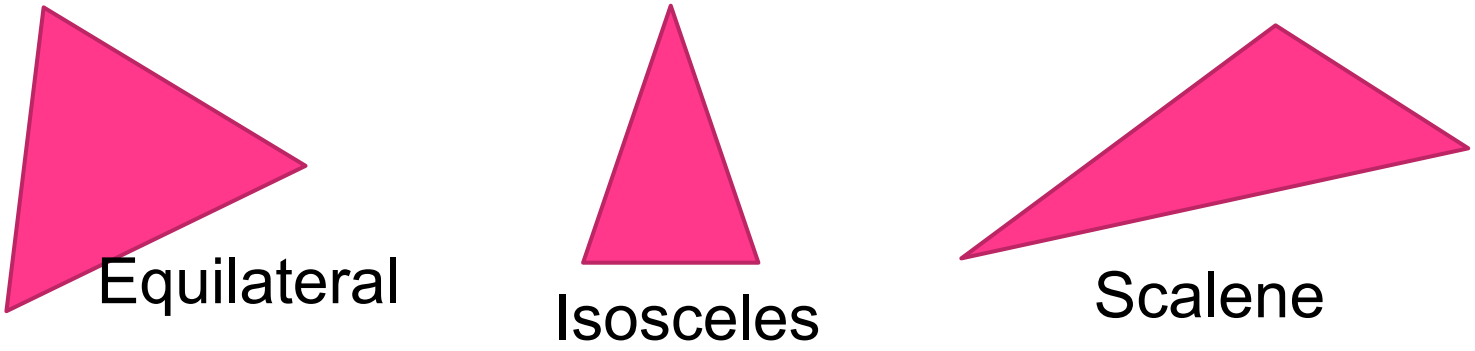
What is “testing” ?

Testing a program: verifying the correctness (or other qualities) of the program by inspecting a finite number of executions.

As such, testing is a pragmatic approach of verification (and we have to accept that it is inherently incomplete).

Note: we will discuss a more complete approach in the 2nd half of the course.

Example: determining triangle type



TriangleType **TriType**(Float a, Float b, Float c) { ... }

If a, b, c represent the sides of a triangle, this methods determines the type of the triangle.

An example of a test

```
Test1() {
        TriangleType ty = TriType(4,4,1) ;

        Assert.AreEqual(Isosceles , ty)

    }
```

Question: how shall we define what a “test” is?

What is a “test” ?

```
Test1() {  ty = TriType(4,4,1) ;  Assert.AreEqual(Isosceles,ty) }
```

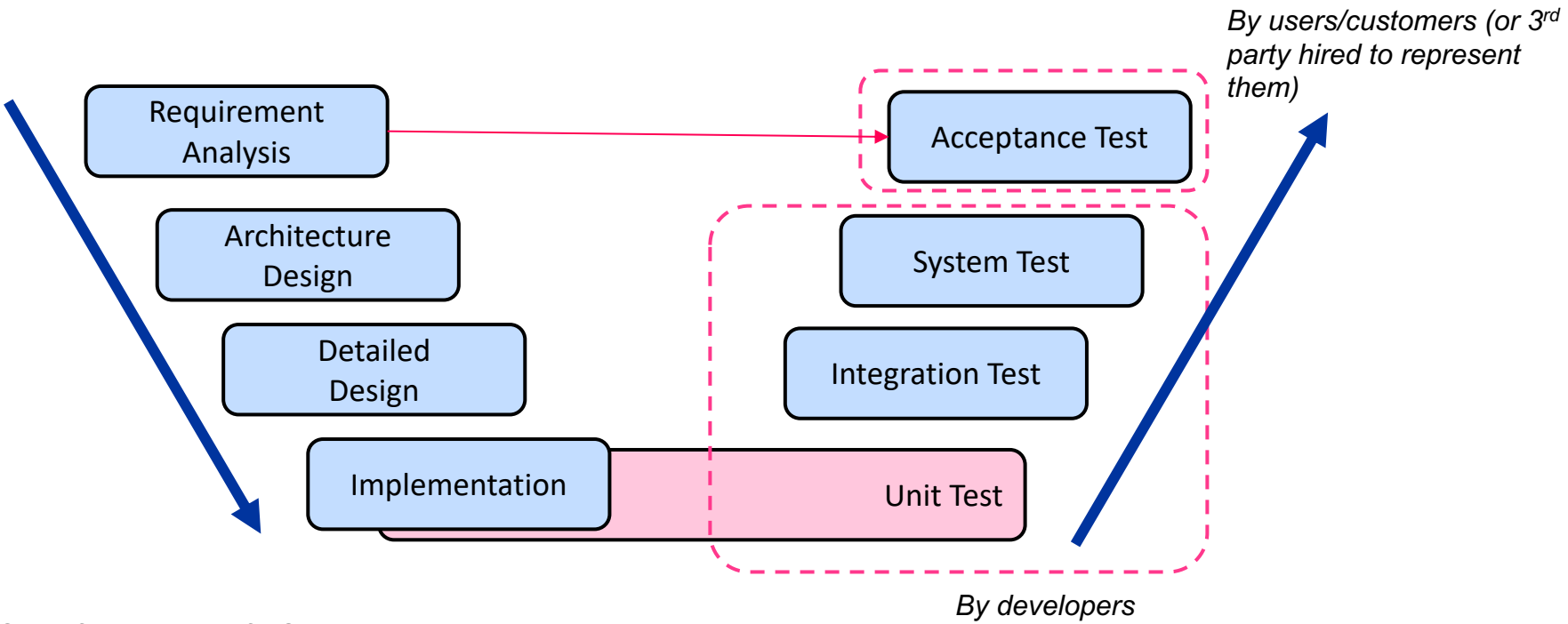
- A “test” (also called *test-case*) for a program P(x) specifies an input for P(x) and the output it expects.
- Note: the formulation of the ”expectation” part is also called *oracle*.
- The definition fits nicely for a function/method-like P.
  - What if P is an interactive program e.g. a web application?
  - What if P is a continuously running system, e.g. a car control system?

A more general definition

*A **test** for P(x) specifies a sequence of interactions along with the needed parameters on P, and the expected responses P should produce.*

- Compare this with AO Def. 1.17 (both definitions try to say the same thing).

Typical V-model testing approach



Simplified version of AO Fig. 1.2.

Ch. 7 provides you more background on “practical aspect”, e.g. concrete work out of the V-model, outlines of “test plan” → read the chapter yourself!

Unit Testing

Make sure that your units are correct!

- Invest in unit testing! Debugging an error at the system-test level is *much* more costly than at the unit level.
- Note: so-called “unit testing tool” can often also be used to facilitate integration and system testing.

What is a “unit” ?

- Program “units” should not be too large, that you can still easily comprehend of its logic.
- Possibilities: functions, methods , or classes as units.

What is a “unit” ?

- **However**, different types of units may have different types of interactions and complexity, thus requiring different approaches to test them:
  - a *function*’s behavior depends only on its parameters; does **not** have any side effect.
  - *A procedure* depends-on its parameters, but may additionally have side effect on its parameters.
  - *A method* may additionally depend on and affect instance variables, or even class (static) variables.
  - *A class*: is a collection of potentially interacting methods.

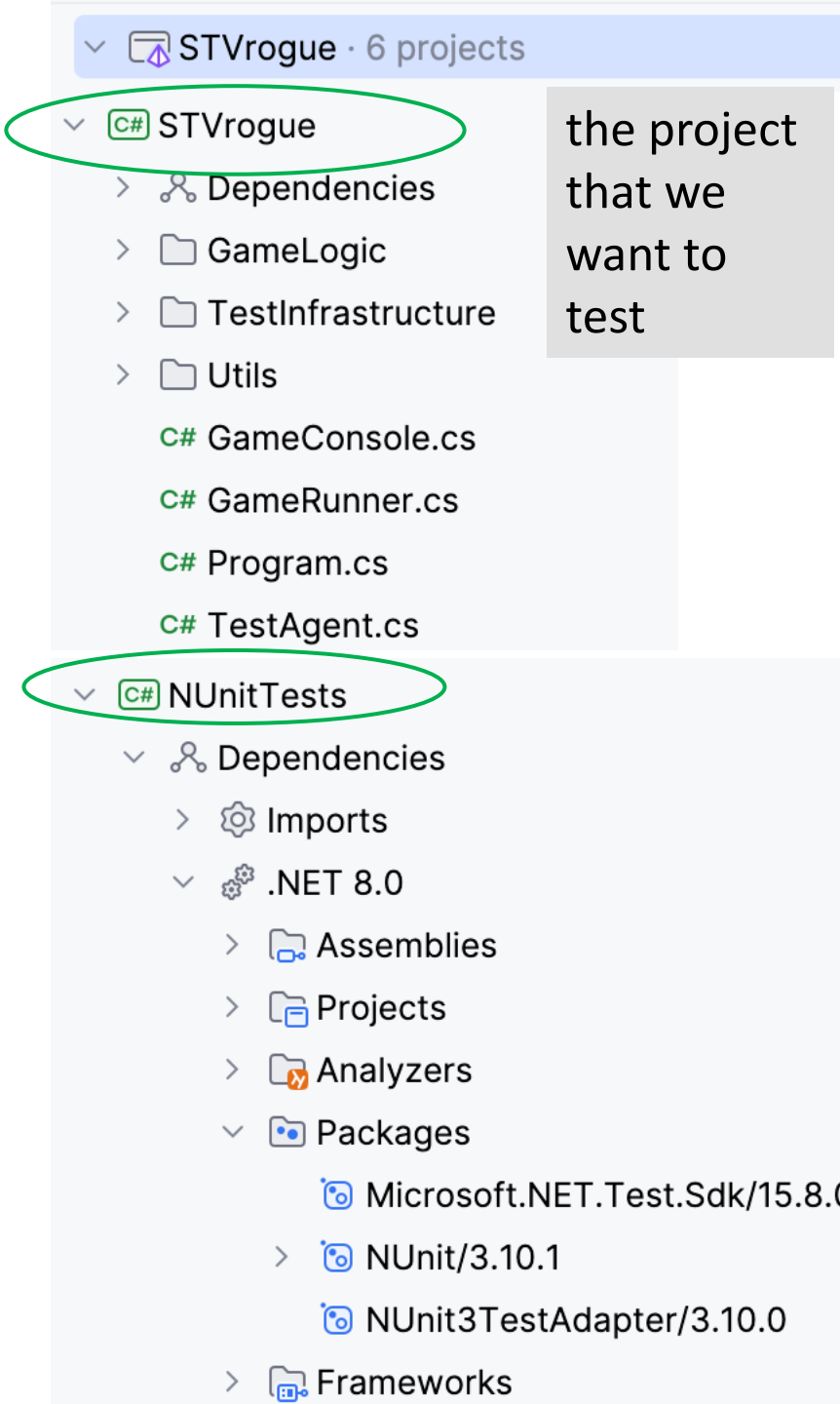
Unit testing in C#

- Unit Testing Framework: MSUnit, **NUnit**, xUnit. We will be using NUnit.
- Coverage tool: both Rider and VS Enterprise have it.
- **Check related tutorials/docs**:
  - NUnit Quick Start (older version, but will do for a tutorial): https://nunit.org/docs/2.5.9/quickStart.html
  - NUnit doc: https://github.com/nunit/docs/wiki/NUnit-Documentation
  - Testing from your IDE (Rider):
    - https://www.jetbrains.com/help/rider/Introduction.html, check the entry on "Get Started with Unit Testing".
    - Obtaining test coverage information: https://www.jetbrains.com/help/dotcover/Getting_Started_with_dotCover.html
- In this lecture we will just go through the underlying concepts.

The structure of a solution with “test projects”

A *test project* is a just a project in your solution that contains your *test-classes*.

dependencies



the project that we want to test

The structure of a “test project”

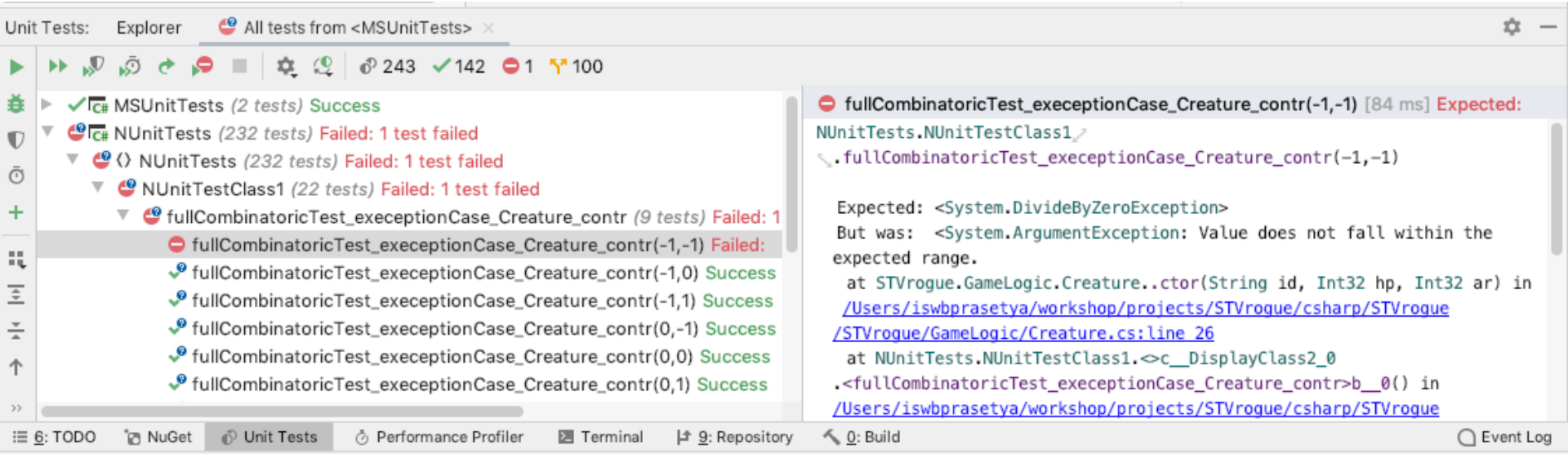
- A *solution* may contain multiple projects; including multiple test projects.
- A *test project* is used to group related *test classes*. You decide what “related” means; e.g. you may decide to put all test-cases for package/namespace in its own test project.
- A *test class* is used to group related *test methods*.
- A *test method* does the actual testing work, it usually encodes a single test-case.

Test Class and Test Method (NUnit)

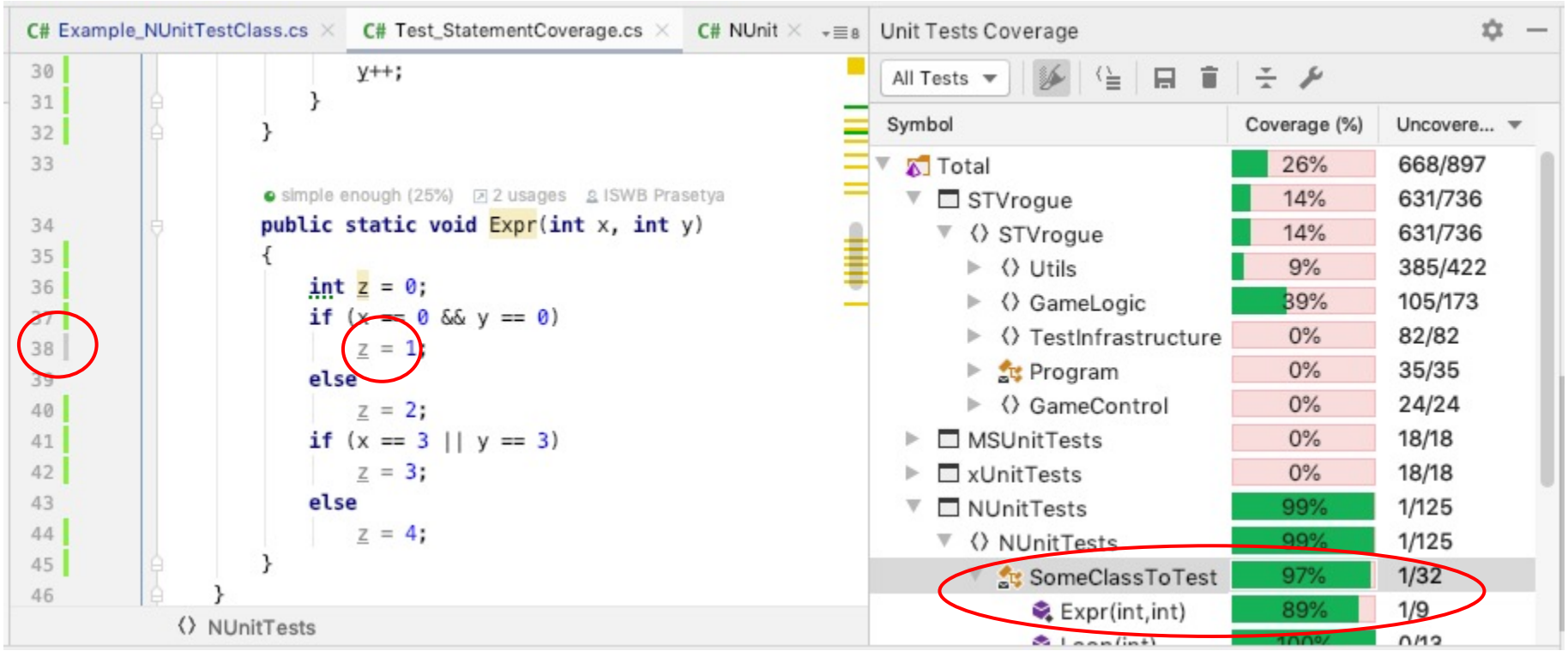
```
[TestFixture]
public class TriangleTest  {
    [ SetUp]
    public static void Init() ...
    //[TearDown]
    //public static void Cleanup() ...

    [Test]
     public void Test1_Triangle()  ...
    [Test]
    public void Test2_Triangle()  ....
}
```

```
Test1_Triangle() {
        var ty = TriType(4,4,1) ;
        Assert.AreEqual(Isosceles, ty)
 }
```

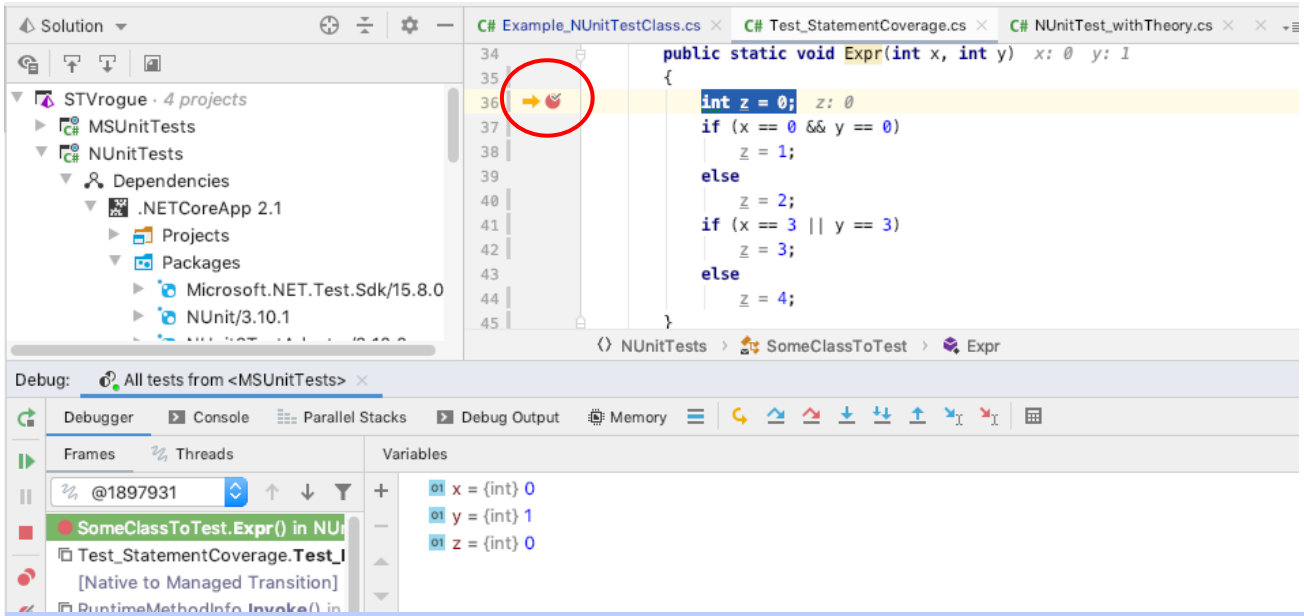
Inspecting Test Result (Rider)



Inspecting Coverage (Rider)



Finding the source of an error: use a debugger!



- Add *break points*; execution is stopped at every BP.
- You can inspect the values of every variable
- You can proceed to the next BP, or execute one step at a time: *step-into*, *step-over*, *step-out*.

Test Oracle

```
Test1_Triangle() {
        var ty = TriType(4,4,1) ;
        Assert.AreEqual(Isosceles, ty)
}
```

An *oracle* specifies your expectation on the program's responses.

- Check NUnit doc on Assertions: https://github.com/nunit/docs/wiki/Assertions
  - Classic way: Assert.IsTrue(x == 0) or Assert.AreEqual(0,x)
  - Constraint model: Assert.That(x, Is.EqualTo(0))

A test needs oracles

```
Test1_Triangle() {
        var  ty = TriType(4,4,1) ;
        Assert.AreEqual(Isosceles, ty)
}
```

- How do we determine what the expected responses of the program under test?
- Ideally, there exists a specification (or you have to elicit that, somehow).

Informal or formal specification?

```
TriType(Float a, Float b, Float c) { ... }
```

Informal: "*If a, b , c represent the sides of a triangle, this methods determines the type of the triangle.*"

Formal specification, pros and cons

- Pros:
  - Precise
  - Can be turned to “executable” specifications.
  - When the program is changed, only its specification needs to be adapted; we don’t have to re-program the test cases.
  - Allow you to “**generate**” the test sequences/inputs rather than writing them manually.
- Cons:
  - Capturing the intended specification is not always easy.
  - Additional work.

Example

- Formalize the specification of TriType(a,b,c).

Informal: "*If a, b , c represent the sides of a triangle, this methods determines the type of the triangle.*"

Formalizing specification with a pre- and post-conditions

Pre-condition

$$\{\ a>0 \wedge b>0 \wedge c>0 \wedge a+b>c \wedge a+c>b \wedge b+c>a\ \}$$

TriType(a,b,c)

$$\{\ (a=b \wedge b=c \wedge a=c) \Leftrightarrow \text{retval=Equilateral}$$
$$\wedge$$
$$(a\neq b \wedge b\neq c \wedge a\neq c) \Leftrightarrow \text{retval=Scalene}$$
$$\wedge$$
$$\ldots \Leftrightarrow \text{retval=Isosceles}\ \}$$

Post-condition

- Formal, but not yet "executable". We can't invoke it from our test cases.
- "…" above means "information not shown" (it does not mean "else")

Turning it to an in-code specification

(here, encoded as a parameterized NUnit-test)

```
void MethodSpec(x) {
    if (...pre-cond...) {
        var retval = Method(x)
        Assert.IsTrue( ...post cond... )
    }
    else Assert.Throws<expected exception>(() => Method(x));
}
```

Pre-condition

Post-condition

Check that the method throws the right exception when the pre-cond is violated.

In-code specifications are specifications expressed in a **programming language**. It is less clean, but it is **executable**, so you can invoke them from your tests.

In-code Spec of TriType

(encoded as a parameterized NUnit-test)

```
bool TriTypeSpec(float a, float b, float c) {
    if (a>0 && b>0 && c>0 && ...)  {
        var retval = TriType(a,b,c)
        Assert.True((retval == Equilateral) == (a==b && b==c && a==c))
        Assert.True((retval == Scalene) == (a!=b && b!=c && a!=c))
        Assert.True((retval == Isosceles) == ...)
    }
    else Assert.Throws<ArgumentException>(() => TriType(a,b,c))
}
```

Pre-condition

Post-condition

Now you can write your tests like this

(NUnit, parameterized test)

```
[TestCase(4,4,1)]
[TestCase(4,4,4)]
[TestCase(1,2,4)]
[TestCase(0,4,1)]
[TestCase(0,0,0)]
...
bool TriTypeSpec(float a, float b, float c) {  ... }
```

Important observation: this approach also opens a way for you to "**generate**" the tests, e.g. using a QuickCheck-like tool.

Note: "TestCase" Nunit attribute can only handle simple types. See also the doc. for '**TestCaseSource**" Nunit attribute.

Example specifications for methods on arrays or collections

```
int GetIndex(x, int[ ] a) { ... }
```

Informal: *given a non-empty integer array a, and assume x occurs in a, the method returns an index k in a of an x.*

Formal spec: now we also need “quantifiers”

- **Informal**: *given a non-empty integer array a, and assume x occurs in a, the method returns an index k in a of an x.*

$$\{\ a \neq null \wedge (\exists k: 0 \leq k < a.length : a[k]=x)\ \}$$

GetIndex(x, int[] a)

$$\{\ 0 \leq retval < a.length \ \wedge \ a[retval] = x\ \}$$

Providing quantifiers as in-code

- Define (static):

```
Exists<T> ( T [ ] a,   Predicate<int> P) {
    for (int k=0; k<a.Length; k++) if (P(k)) return true  ;
    return false ;
}
```

- **Exists**(a,P) = there exists a valid index i of a, such that P(i) is true.
- Similarly you can define **Forall**(a,P).
- Similarly you can define quantifiers for collections.
- Already provided in STVRogue project.

Specifying properties of arrays (and similarly collections)

- Now we can write in-code properties, e.g. : the array a contains only positive integers:

```
Forall(a,  k =>  a[k] > 0 )
```

- Or, the array a has at least one positive integers:

```
Exists(a,  k =>  a[k] > 0 )
```

- Alternatively using built-in a.Any(..) and a.All(..)

Example: in-code spec of GetIndex

Informal: *given a non-empty integer array a, and assume x occurs in a, the method returns an index k in a of an x.*

```
GetIndexSpec(int x, params int[] a) {

   if (a!=null &&  Exists(a,  k =>  a[k]==x))  {        Pre-condition

      var retval    =   GetIndex(x,a)

      Assert.IsTrue(a[retval]) == x)  }                 Post-condition

   else   Assert.Throws<ArgumentException>(() => GetIndex(x,a))

}
```

Using GetIndexSpec for parameterized test

```
[TestCase(0)]
[TestCase(0,1)]
[TestCase(0,0,0)]
[TestCase(0,1,1,1,0)]
...
GetIndexSpec(int x, params int[] a) { ... }
```

Mock

- Consider testing a class C that uses a class D.
- In a larger project, it is possible that D is developed by a separate team, and by the time you want to test C, D is not ready/stable yet. **Solution**: we ”mock” D.
- A *mock* of a program P:
  - has the same interface as P
  - may only implement a very small subset of P’s behavior
  - fully under your control
- You would want a way to conveniently create mocks.
- Mentioned in Ch. 6.2.1 A&O (12.2.1 2nd Ed).

Example: Heater

```
class Heater
double limit
bool active
Thermometer thermometer
public AutoOff() {
    // if thermometer.value()
   // exceeds the limit,
   // deactivate the heater }
```

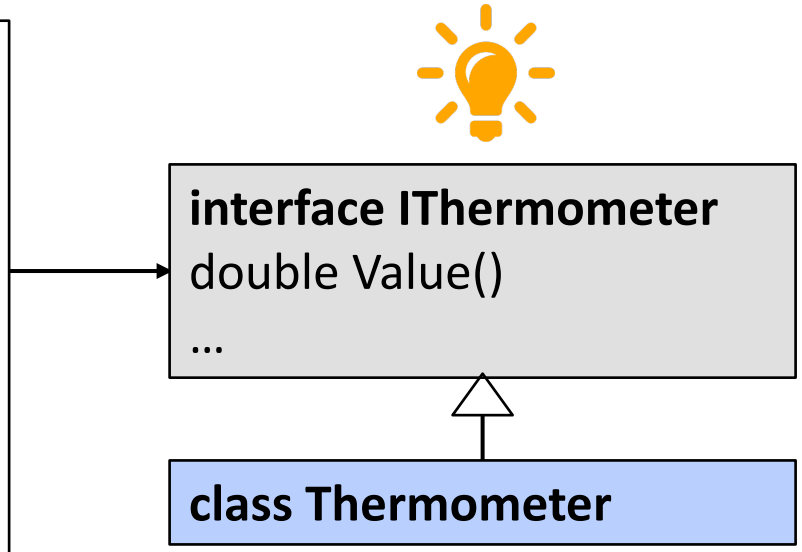
```
class Thermometer
double Value()
…
```

```
test1() {
  thermometer = .... // get some thermometer
  Heater heater = new Heater(thermometer)
  Assume.That(thermometer.Value() >= heater.limit+0.001)
  heater.AutoOff()
  Assert.IsFalse(heater.active)
}
```

Notice that for this test you need a working Thermometer class.

You need to restructure a bit

```
class Heater
double limit
bool active
IThermometer thermometer
Heater(Ithermometer t) // contr
autoOff() {
    // if thermometer.value()
   // exceeds the limit,
   // deactivate the haeter
}
```



To facilitate mocking usually you need to replace your dependency so that your class of interest depends on interfaces instead.

Creating mocks dynamically using NSubstitute

- Create an instance of an Interface, by calling the method “For” from the class ” NSubstitute.Substitute” :

```
thermometer = Substitute.For<IThermometer>()
```

- You can program the behavior of the interface’s methods on-demand. General form: mock.<method>(x1,x2,...).**Returns**(y), e.g.:

```
thermometer.Value ().Returns(100)
```

Now whenever you do thermometer.Value() this will return 100.

Test with a mock

class Heater
double limit
bool active
IThermometer **thermometer**
Heater(Ithermometer t)
autoOff()

interface IThermometer
double Value()
…

Now we can write test1() like this:

```
test1() {
  thermometer = Substitute.For<IThermometer>()
  Heater H = new Heater(thermometer)
  thermometer.Value().Returns(heater.limit + 0.001)
  H.autoOff()
  Assert.IsFalse(H.active)
}
```

Creating a mock thermometer and programming its behavior.

Specifying a “class”

- Many classes have methods that *interact with each other* (e.g. as in Stack). How to specify (and test) these interactions?
- Option-1: specify every method with pre- and post-conditions (we discussed this).
  This is expressive enough, but pre- and post-conditions do not naturally capture inter-method interactions.

Specifying a “class”

- Other options, which can be user either as alternatives or in conjunction with pre/post-conditions:
  - class invariant
  - Abstract Data Type (ADT)
  - Finite State Machine (FSM)

Specifying with class invariant

```
class Thermometer {
   double temperature // in Celcius
   Value()
   Incr(t)
   Decr(t)
}
```

- A class invariant of a class C specifies valid states of instances of C. When an instance is created, it must have a valid state. All operations/methods of C are expected to restore the state of their target instance to a valid state.
- Example, for Thermometer, its class inv could be: temperature ≥ -273.15

Specifying a class as an ADT

An *Abstract Data Type* (ADT) is a model of a (stateful) data structure. The data structure is modeled abstractly by only describing a set of *operations* (without exposing underlying data structure implementing the state).

The semantic is described in terms of "logical properties" (also called the ADT's *axioms*) over those operations.

Example : specifying my ItemStore

```
class ItemStore<T> {
    Store(T  x)
    T   Get()
     int Size()
}
```

Do these look familiar?

Axioms :

1. The Size of a new ItemStore is 0.
2. After s.Store(x), Size is 1 + the Size before.
3. If Size>0, after S.Get(), Size is one less than the Size before.
4. After S.Store(x), s.Get() gives x.

Testing ADT

Testing an ADT amounts to verifying each of its axioms, each can be formulated as a **parameterized test.**

For example, to test Ax-4:

```
Axiom4(ItemStore s, T x) {
  s.Store(x) ;  assertEqual(x, s.Get()) ; }
```

We can imagine these test cases :

1. empty *s*
2. a non-empty *s* that does not contain *x*
3. an *s* that already contains x