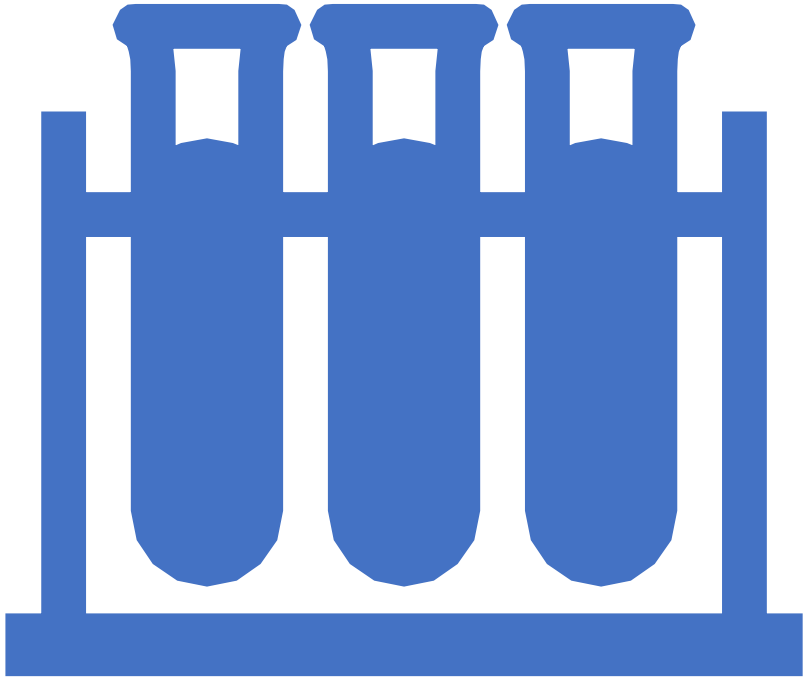


Software Testing

Software Testing

- Tests are typically binary with software: either it performs as you expect or it doesn't perform as you expect…
- Tests will take a variety of formats, from manual observation to automation, but usually adhere to the following ruleset:
  - Tests pass if the behaviour is as you expect
  - Tests fail if the behaviour differs from that expected

BUCKINGHAMSHIRE
NEW UNIVERSITY
EST. 1891

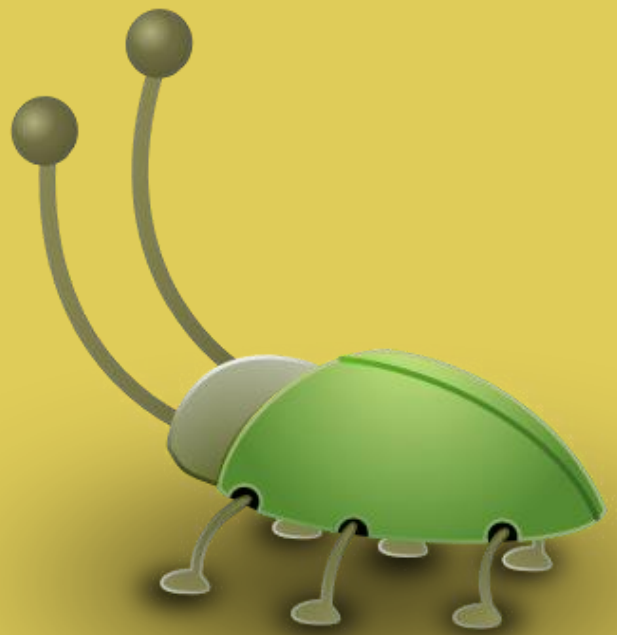
Program bugs

If the behaviour of the program does not match the behaviour that you expect, then this means that there are bugs in your program that need to be fixed.

There are two causes of program bugs:

- ***Programming errors*** - You have accidentally included faults in your program code. For example, a common programming error is an ‘off-by-1’ error where you make a mistake with the upper bound of a sequence and fail to process the last element in that sequence.

- ***Understanding errors*** - You have misunderstood or have been unaware of some of the details of what the program is supposed to do. For example, if your program processes data from a file, you may not be aware that some of this data is in the wrong format, so your program doesn’t include code to handle this.



Sommerville, I. (2018) Engineering Software Products,
An Introduction to Modern Software Engineering

BUCKINGHAMSHIRE
NEW UNIVERSITY
EST. 1891

Types of testing

Functional testing

Test the functionality of the overall system. The goals of functional testing are to discover as many bugs as possible in the implementation of the system and to provide convincing evidence that the system is fit for its intended purpose.

User testing

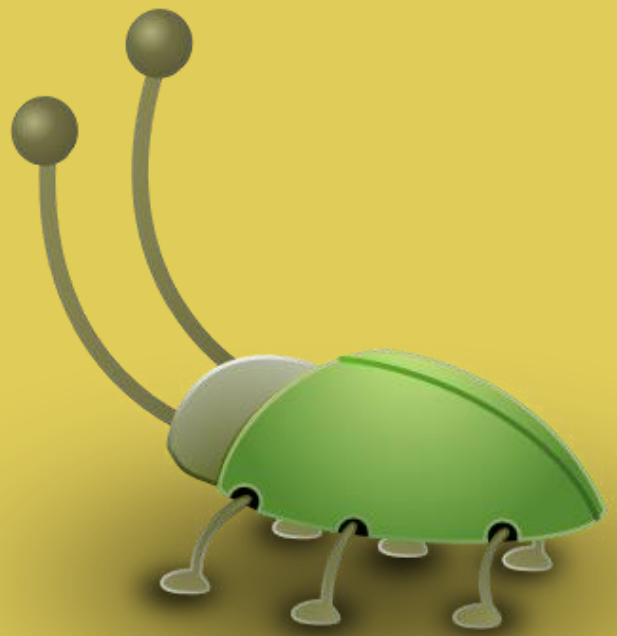
Test that the software product is useful to and usable by end-users. You need to show that the features of the system help users do what they want to do with the software. You should also show that users understand how to access the software's features and can use these features effectively.

Performance and load testing

Test that the software works quickly and can handle the expected load placed on the system by its users. You need to show that the response and processing time of your system is acceptable to end-users. You also need to demonstrate that your system can handle different loads and scales gracefully as the load on the software increases.

Security testing

Test that the software maintains its integrity and can protect user information from theft and damage.



Sommerville, I. (2018) Engineering Software Products,
An Introduction to Modern Software Engineering

BUCKINGHAMSHIRE
NEW UNIVERSITY
EST. 1891

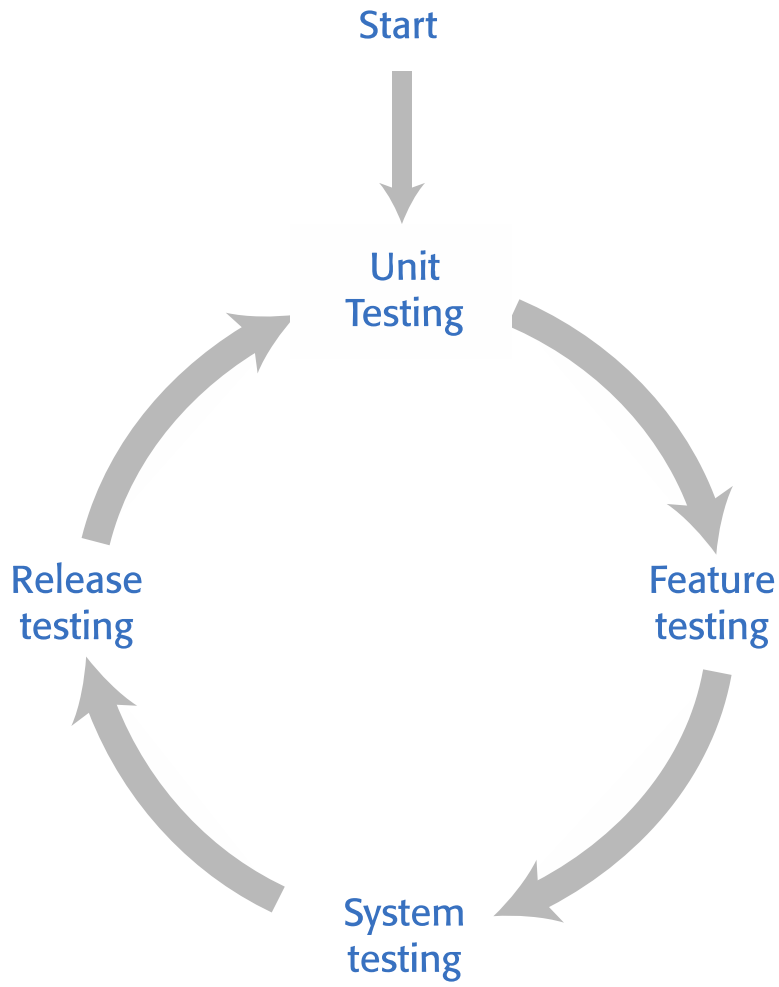
Functional testing processes (1)

Unit testing

- The aim of unit testing is to test program units in isolation.
- Tests should be designed to execute all of the code in a unit at least once.
- Individual code units are tested by the programmer as they are developed.

Feature testing

- Code units are integrated to create features.
- Feature tests should test all aspects of a feature.
- All of the programmers who contribute code units to a feature should be involved in its testing.

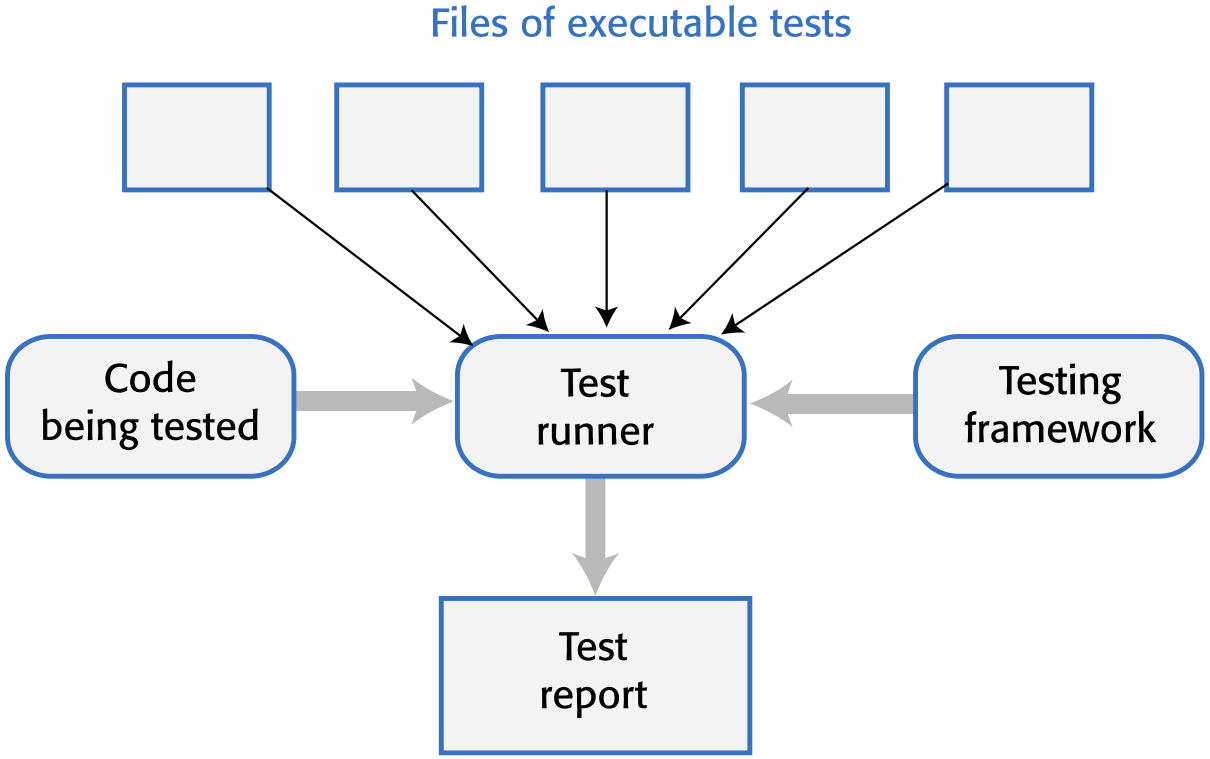


Sommerville, I. (2018) Engineering Software Products, An Introduction to Modern Software Engineering

Test Automation

Test automation

- Automated testing is based on the idea that tests should be executable.
- An executable test includes the input data to the unit that is being tested, the expected result and a check that the unit returns the expected result.
- You run the test and the test passes if the unit returns the expected result.
- Normally, you should develop hundreds or thousands of executable tests for a software product.



BUCKINGHAMSHIRE
NEW UNIVERSITY
EST. 1891

Automated tests

It is good practice to structure automated tests into three parts:

Arrange - You set up the system to run the test. This involves defining the test parameters and, if necessary, mock objects that emulate the functionality of code that has not yet been developed.

Action - You call the unit that is being tested with the test parameters.

Assert - You make an assertion about what should hold if the unit being tested has executed successfully.

If you use equivalence partitions to identify test inputs, you should have several automated tests based on correct and incorrect inputs from each partition.

```
# TestInterestCalculator inherits attributes and
# methods from the class
# TestCase in the testing framework unittest

class TestInterestCalculator (unittest.TestCase):
    # Define a set of unit tests where each test tests
    # one thing only
    # Tests should start with test_ and the name should
    # explain what is being tested

    def test_zeroprincipal (self):
          #Arrange - set up the test parameters
          p = 0; r = 3; n = 31
          result_should_be = 0
          #Action - Call the method to be tested
          interest = interest_calculator (p, r, n)
          #Assert - test what should be true
          self.assertEqual (result_should_be, interest)

    def test_yearly_interest (self):
          #Arrange - set up the test parameters
          p = 17000; r = 3; n = 365
          #Action - Call the method to be tested
          result_should_be = 270.36
          interest = interest_calculator (p, r, n)
          #Assert - test what should be true
          self.assertEqual (result_should_be, interest)
```

BUCKINGHAMSHIRE
NEW UNIVERSITY
EST. 1891

Test name_check function

```
import unittest

from RE_checker import namecheck

class TestNameCheck(unittest.TestCase):

   def test_alphaname(self):
           self.assertTrue(namecheck('Sommerville'))

   def test_doublequote(self):
           self.assertFalse(namecheck("Thisis'maliciouscode'"))

   def test_namestartswithhyphen(self):
           self.assertFalse(namecheck('-Sommerville'))

   def test_namestartswithquote(self):
           self.assertFalse(namecheck("'Reilly"))

   def test_nametoolong(self):
           self.assertFalse(namecheck
   ('Thisisalongstringwithmorethen40charactersfrombeginningtoend'))

   def test_nametooshort(self):
           self.assertFalse(namecheck('S'))

   def test_namewithdigit(self):
           self.assertFalse(namecheck('C-3PO'))

   def test_namewithdoublehyphen (self):
           self.assertFalse (namecheck ('--badcode'))

   def test_namewithhyphen(self):
           self.assertTrue(namecheck('Washington-Wilson'))
```

```
import unittest

loader = unittest.TestLoader()

# Find the test files in the current directory

tests = loader.discover('.')

# Specify the level of information provided
# by the test runner

testRunner =
unittest.runner.TextTestRunner(verbosity=2)
testRunner.run(tests)
```

Unit testing guidelines

Test edge cases
If your partition has upper and lower bounds (e.g. length of strings, numbers, etc.) choose inputs at the edges of the range.

Force errors
Choose test inputs that force the system to generate all error messages. Choose test inputs that should generate invalid outputs.

Fill buffers
Choose test inputs that cause all input buffers to overflow.

Repeat yourself
Repeat the same test input or series of inputs several times.

Overflow and underflow
If your program does numeric calculations, choose test inputs that cause it to calculate very large or very small numbers.

Don't forget null and zero
If your program uses pointers or strings, always test with null pointers and strings. If you use sequences, test with an empty sequence. For numeric inputs, always test with zero.

Keep count
When dealing with lists and list transformation, keep count of the number of elements in each list and check that these are consistent after each transformation.

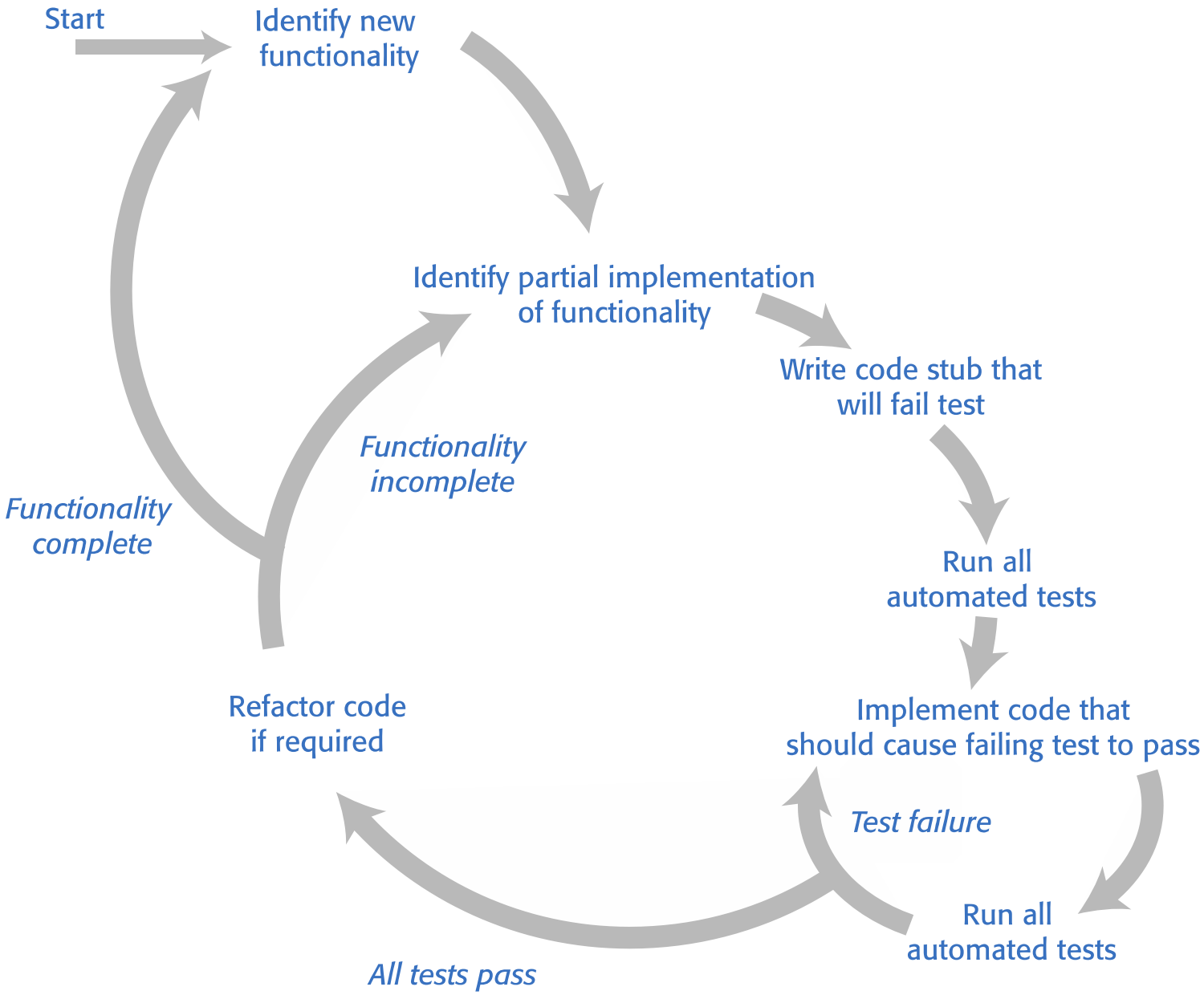
One is different
If your program deals with sequences, always test with sequences that have a single value.

```
def namecheck(s):
  # checks that a name only includes
  # alphabetic characters, -, or single quote
  # names must be between 2 and 40
  #   characters long
  # quoted strings and -- are disallowed

  namex = r"^[a-zA-Z][a-zA-Z-']{1,39}$"
  if re.match (namex, s):
    if (re.search ("'.*'", s) or
        re.search ("--", s)):
      return False
    else:
      return True
  return False
```

Test Driven Development (TDD)

Test Driven Development

- Software Requirements are converted into test cases, before the software is fully developed.
- Test cases are designed to be run repeatedly as development progresses. It can be a useful measure of how well the requirements are fulfilled.
- Kent Beck is credited with having ‘rediscovered’ this process after promoting a ‘test-first’ concept in Extreme programming (1999) – where pair programming comes from.
- In 2003, Kent stated that “TDD encourages simple designs and inspires confidence”



Sommerville, I. (2018) Engineering Software Products,
An Introduction to Modern Software Engineering

Test Driven Development (TDD) Cycle

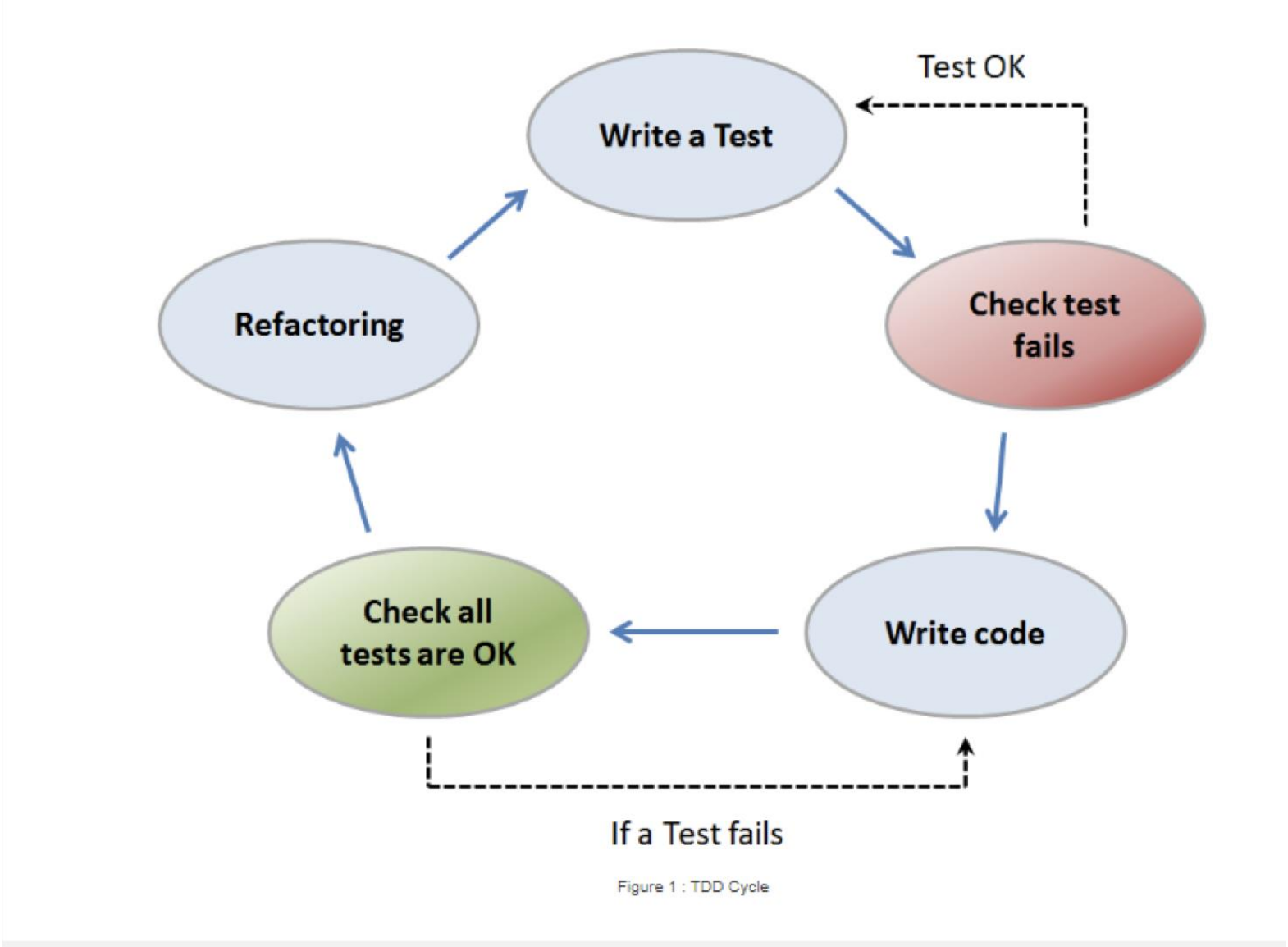
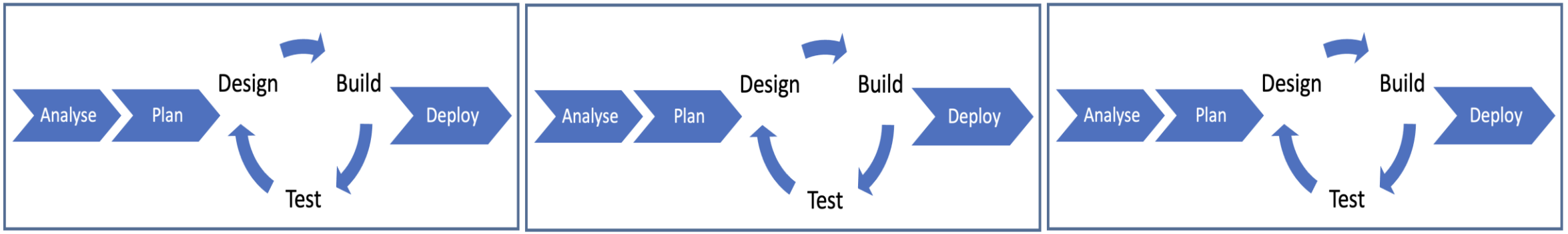


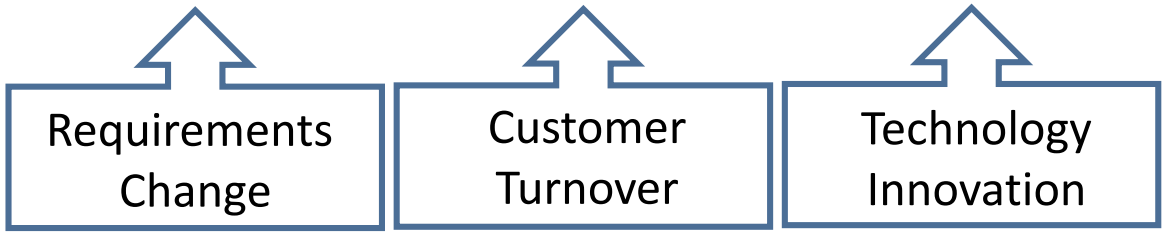
Figure 1 : TDD Cycle

Waterfall

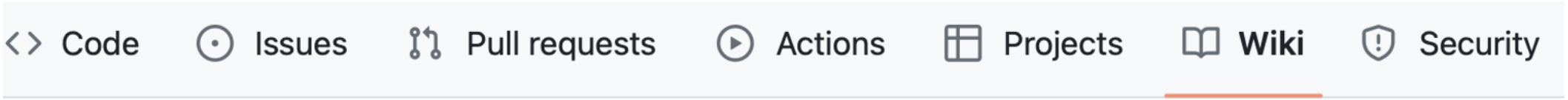
Agile



Project Timeline



Requirements – map to tests!



Space Invaders

Nicholas Day edited this page now · 1 revision

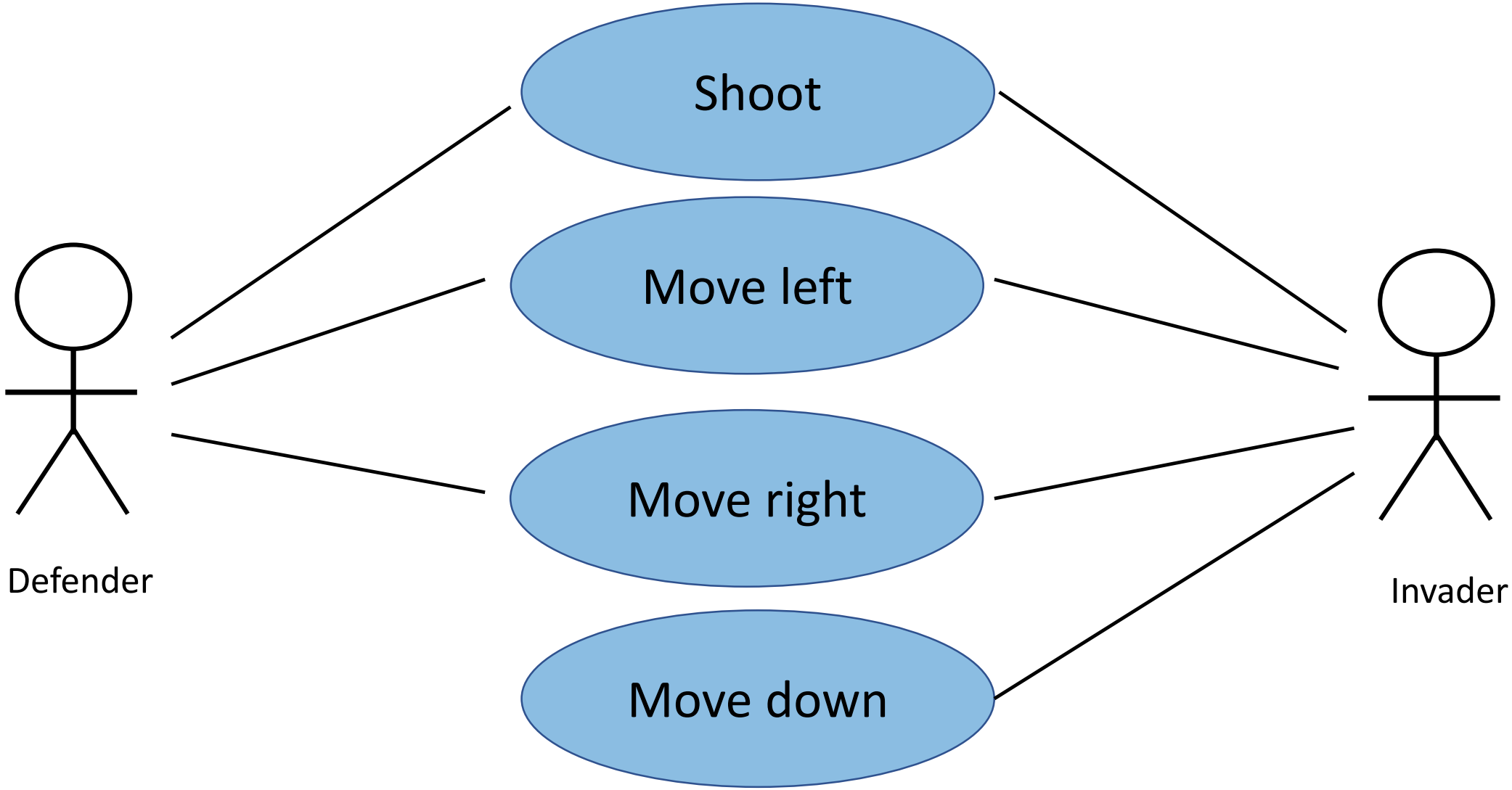
Requirements

- The Defender should move left and right and be able to fire bullets
- Invaders must alternate direction as they move down the screen
- Invaders speed of travel should increase as fewer remain and levels progress
- Shields should crumble as they shot by both the Invader and the Defender

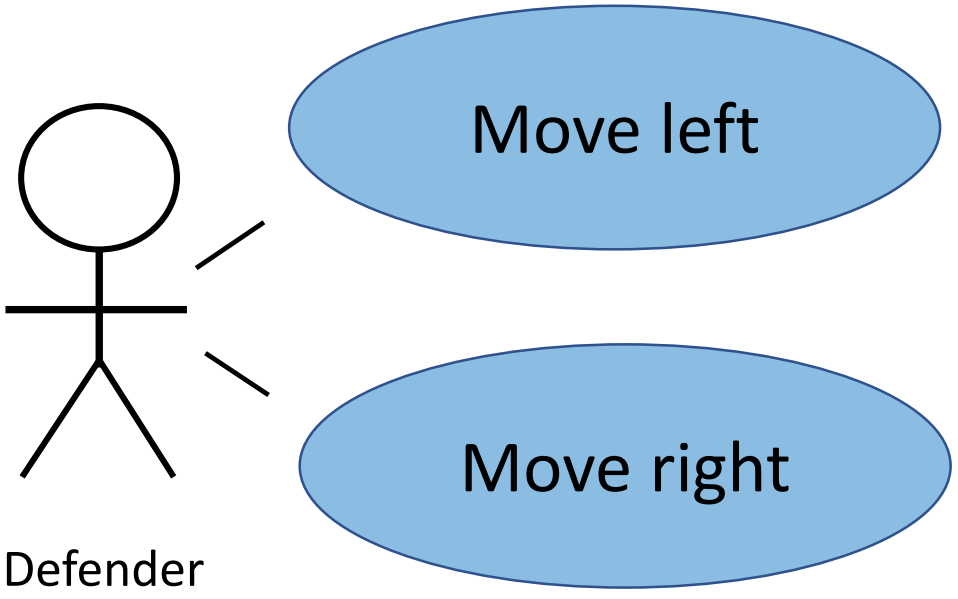
Requirements – map to tests!

- The defender should be able to move left and right:
  - def test_move_left()
  - def test_prevent_offside_left()
  - def test_move_right()
  - def test_prevent_offside_right()
- The defender should be able to fire bullets:
  - def test_shoot_bullet()
  - def test_remove_enemy_upon_collision()
- Invaders must move in an alternative pattern:
  - def test_detect_edge()
  - def test_reverse_direction()

Use Case Diagram – Specialist movement



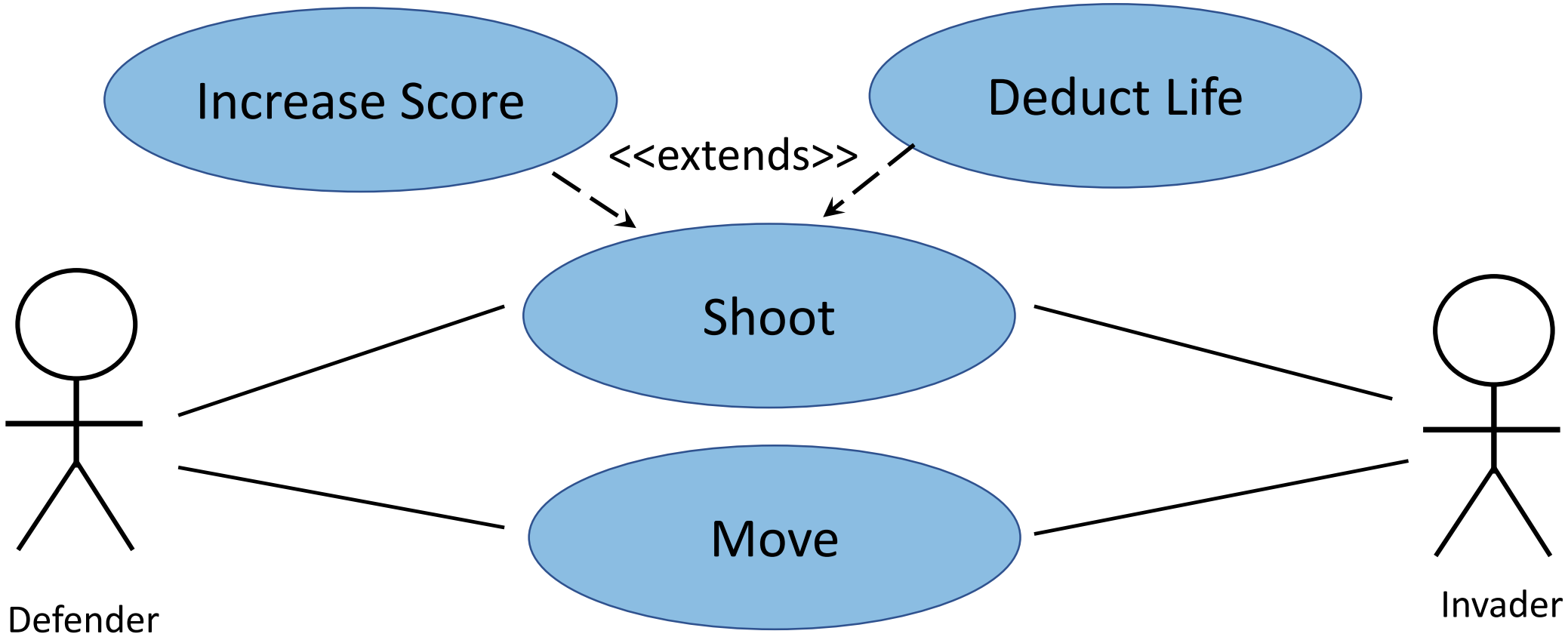
Designing Unit Tests alongside your design



- Does move_left() return 1 or 0 when called?
- When player.x <= 0, does move_left() stop moving the player offscreen?

- Does move_right() return 1 or 0 when called?
- When player.x >= SCREEN_WIDTH, does move_right() stop moving the player offscreen?

Use Case Diagram – Extension actions

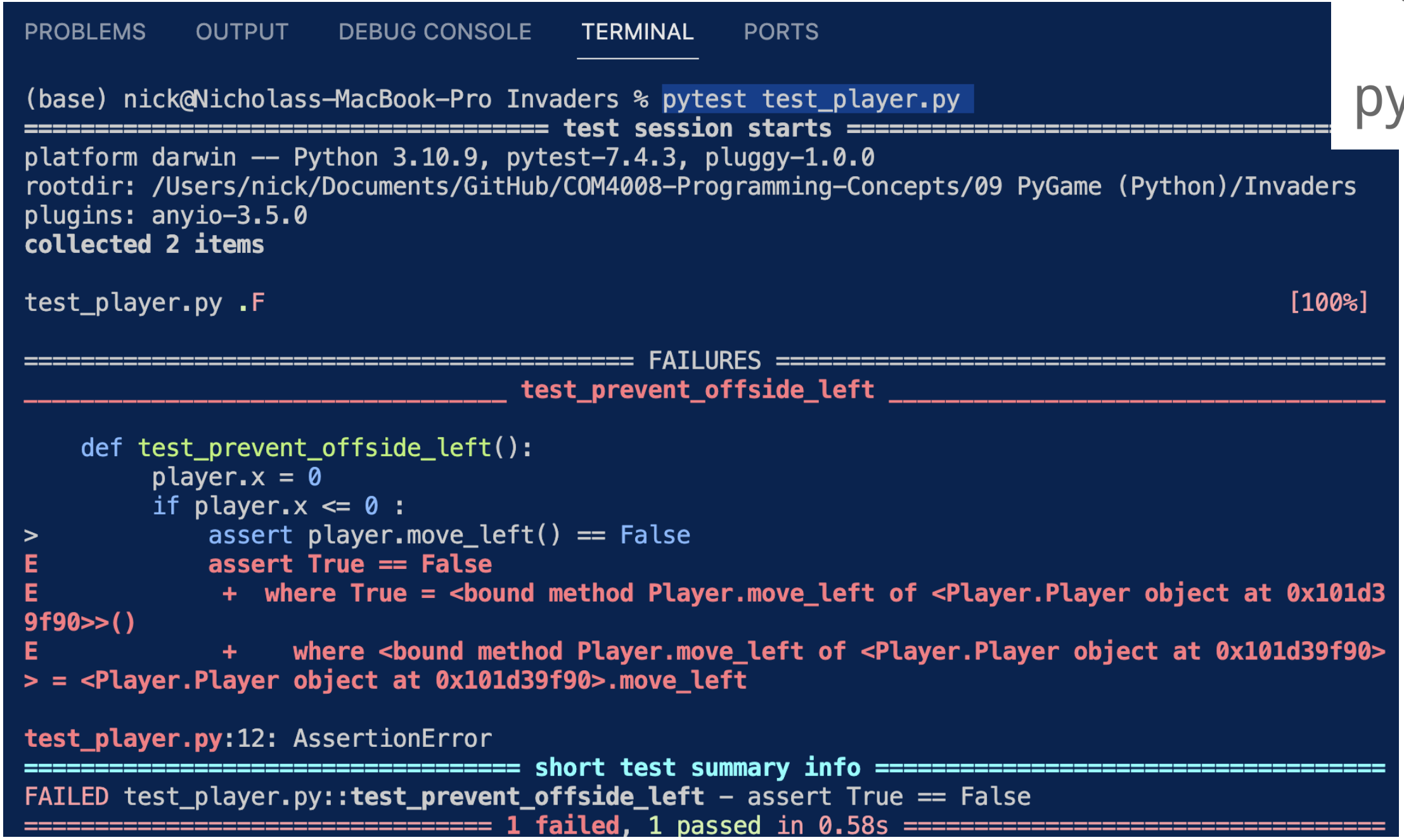


Unit Test Frameworks

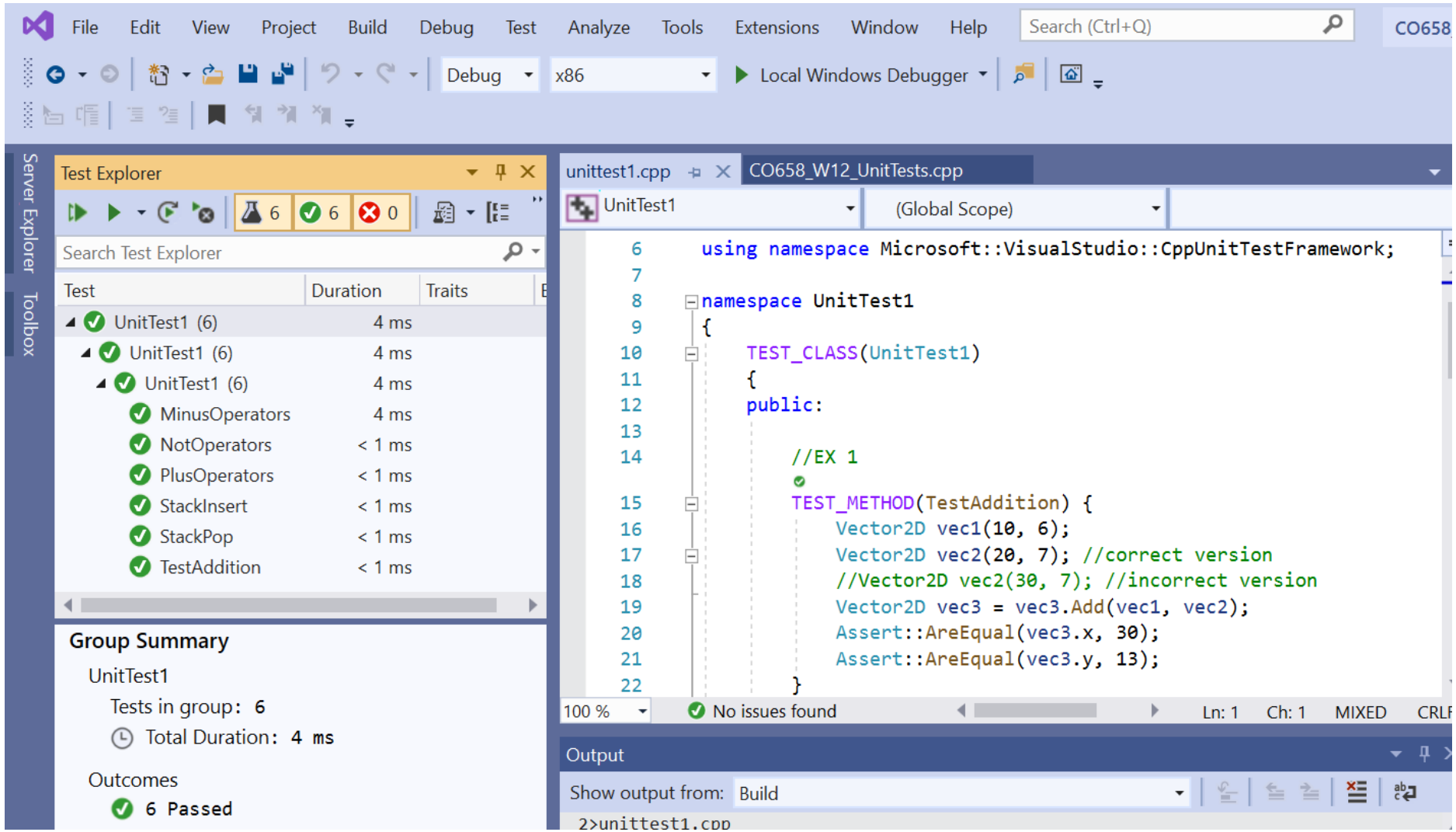
Pytest

- Pytest is an open-source package available at: https://docs.pytest.org/en/7.4.x/index.html
- It works on functions that are set up to ‘test’ a behaviour by the outcome of an ‘assertion’.
- This assertion tests to see if a condition is True. If the assertion is True, then the test has passed, if not, then the test fails.

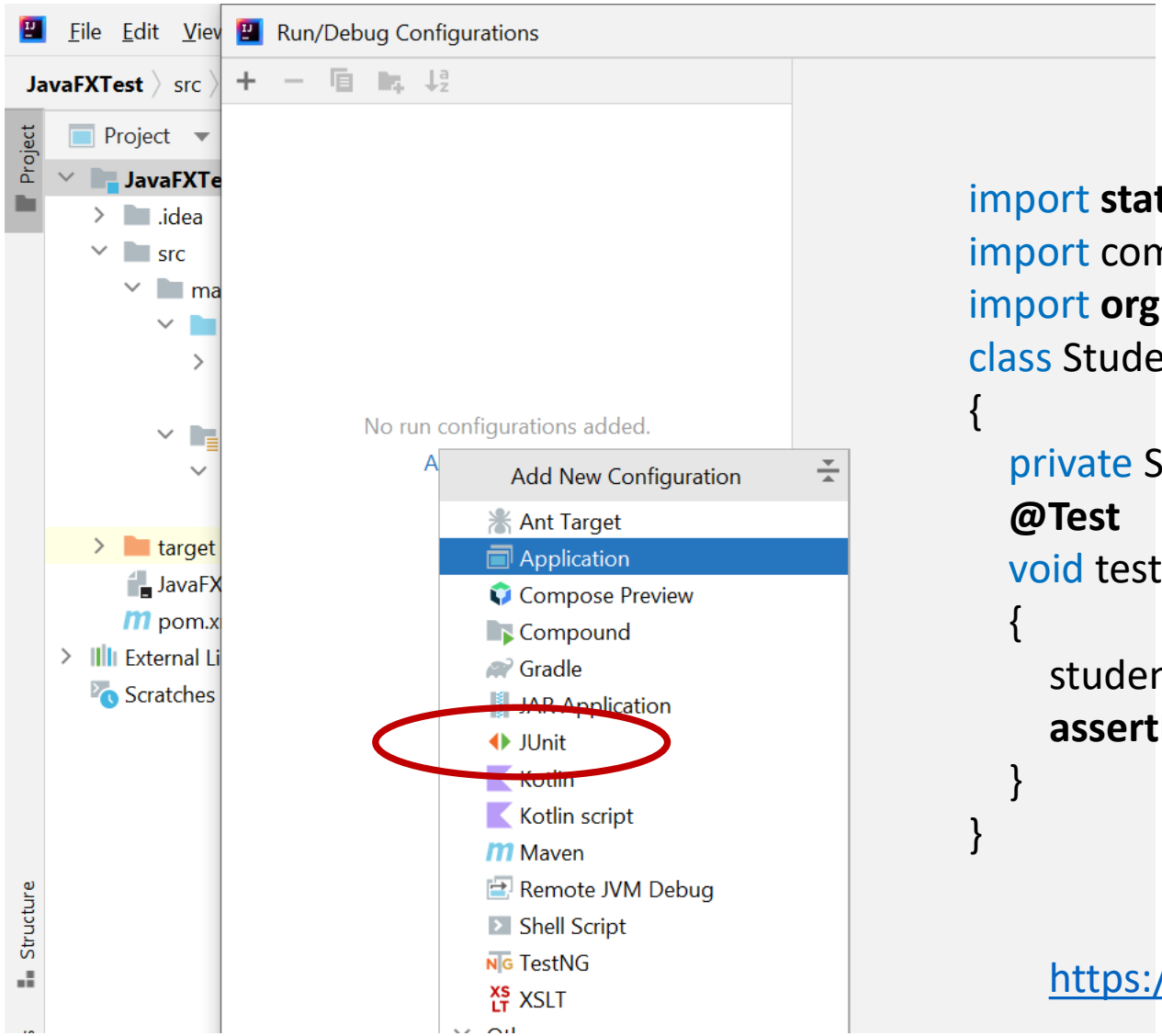
```
def add(x):
    return x + 1

def test_answer():
    assert add(3) == 5
```


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
(base) nick@Nicholass-MacBook-Pro Invaders % pytest test_player.py
test session starts
platform darwin -- Python 3.10.9, pytest-7.4.3, pluggy-1.0.0
rootdir: /Users/nick/Documents/GitHub/COM4008-Programming-Concepts/09 PyGame (Python)/Invaders
plugins: anyio-3.5.0
collected 2 items
test_player.py .F [100%]
FAILURES
test_prevent_offside_left
def test_prevent_offside_left():
player.x = 0
if player.x <= 0 :
> assert player.move_left() == False
E assert True == False
E + where True = <bound method Player.move_left of <Player.Player object at 0x101d39f90>>()
E + where <bound method Player.move_left of <Player.Player object at 0x101d39f90> = <Player.Player object at 0x101d39f90>.move_left
test_player.py:12: AssertionError
short test summary info
FAILED test_player.py::test_prevent_offside_left - assert True == False
1 failed, 1 passed in 0.58s
pytest

Visual Studio 2022's Test Explorer



JUnit5 (Java's Unit Testing)




```
import static org.junit.jupiter.api.Assertions.assertEquals;
import com.company.Student;  //Class location
import org.junit.jupiter.api.Test;
class StudentTests
{
   private Student student = new Student();
   @Test                //annotation
   void testSetID()
   {
      student.setID(1234);    //define a value to test
      assertEquals(1234, student.getID());   // compare expected vs actual
   }
}
```

https://junit.org/junit5/docs/current/user-guide/#writing-tests