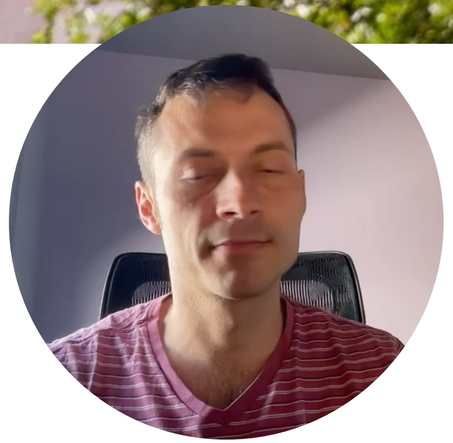
Unit testing



Testing

- Process to identify the correctness, completeness, and quality of developed computer software.
  - *Finding bugs/errors!*

- Process of executing a program under **positive** & **negative** conditions by manual/automated means
  - *Checks:*
    - Specification
      - What is this?
    - Functionality
    - Performance
    - other..

Objective

- Uncover as many bugs as possible in a given product
- Demonstrate a given software matches the requirement specifications
- Validate the quality of software with testing
  - Using the minimum cost and efforts
- High-quality test cases ~ effective tests ~ reports (do not repeat yourself)

Black-, Gray-, & White-box Testing

Input determined by...

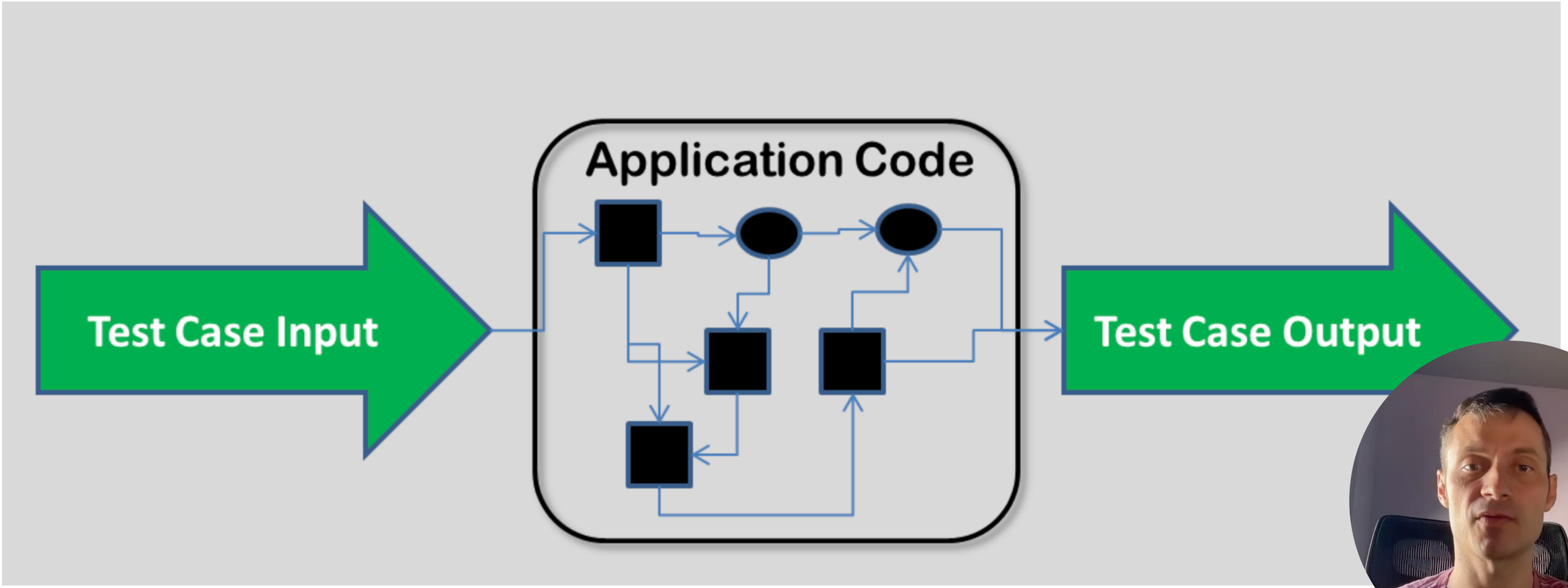
Result

… requirements → **Black box** → *Actual* output compared with *required* output

… requirements & key design elements → **Gray box** → *As for black- and white box testing*

… design elements → **White box** → Confirmation of expected behavior

White box



Test granularity

- Unit Testing
- (Component Testing)
- Integration Testing
- System Testing
- Acceptance Testing

Unit Testing

Goal:

- Confirm that a component or subsystem is correctly coded and carries out the intended functionality

- Test each module individually

- White box
- Follow the program logic

- Done by developers
- JUnit/TestNG frameworks

Integration Testing

Goal:

- Test the interfaces among the subsystems.

- Once all the modules are tested, integration is performed next
- Tests address connected interfaces
- Performed by developers

System Testing

Goal: Determine if the system meets the requirements (functional and non-functional)

- The system as a whole is tested to uncover requirements errors
- Carried out by developers

- Verifies that all system elements work properly and that overall system function and performance has been achieved

- Alpha Testing
- Beta Testing
- Performance Testing
- (detailed on next slide)

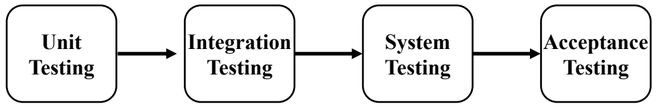
Cont..

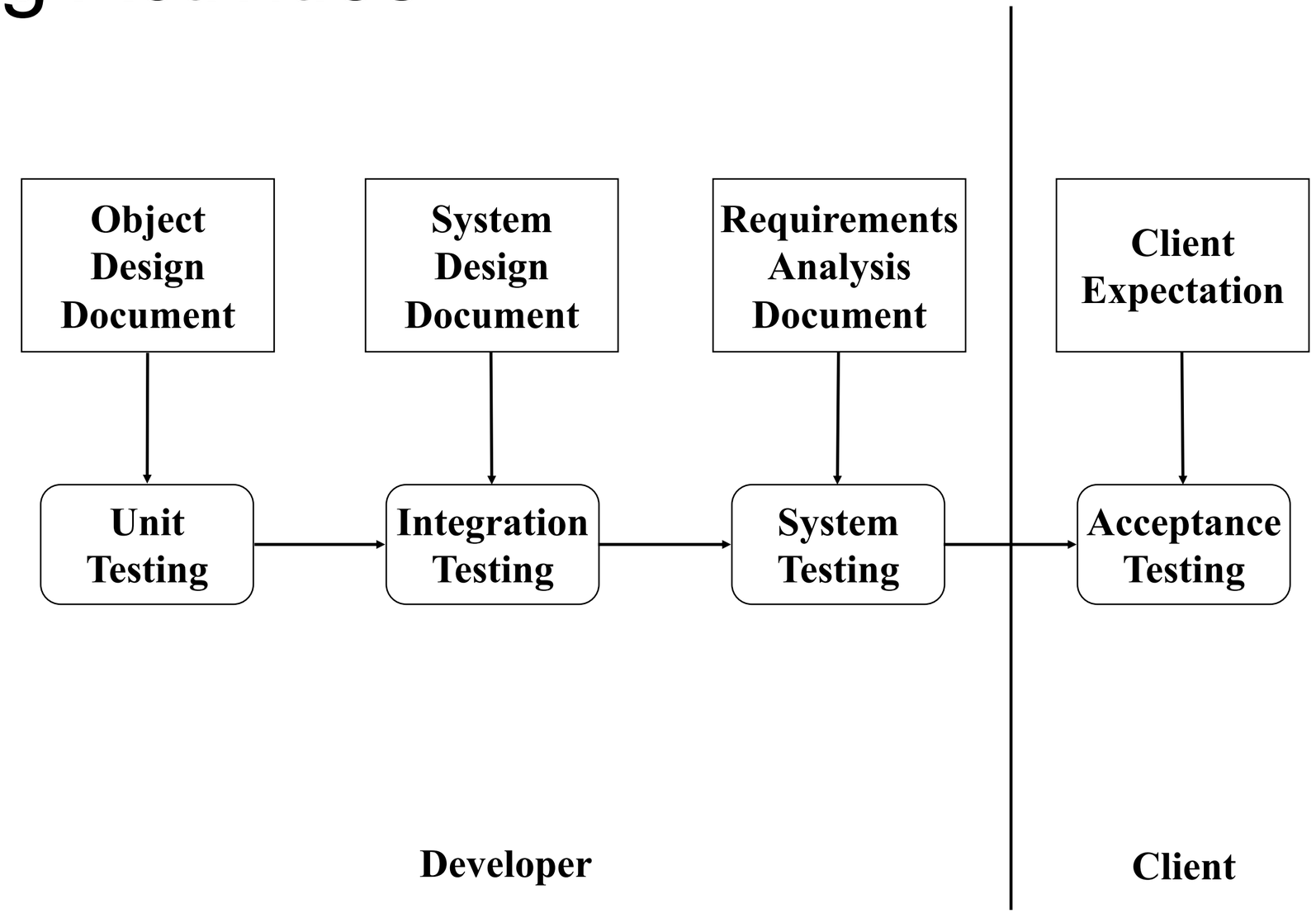
- Alpha Testing
  - *Performed by test team within the developing organization*
- Beta Testing
  - *Performed by a selected group of brave customers*

- Performance Testing
  - *Checks whether the system meats non-functional requirements*

    - Many other tests! (security, availability, leaks..)

Testing Activities





When should you write a test?

- Traditionally after the source code is written
- In eXtremeProgramming (XP) before the source code written
  - *Test-Driven Development Cycle*
    - Add a test
    - Run the automated tests
  - *=> see the new one fail*
    - Write some code
    - Run the automated tests
  - *=> see them succeed*
    - Refactor code.

Unit Testing I

- Is a level of the software testing process where
  - *individual units/components of a software/system are tested.*

- The purpose
  - *to verify that each unit of the software performs as designed.*

Unit Testing II

- A method by which individual units of source code are tested to determine if they are fit for use
- Concerned with **functional correctness and completeness** of individual program units
- Typically written and run by **software developers** to ensure that code meets its design and behaves as intended.
- Its goal is to **isolate each part of the program** and show that the individual parts are correct.

Unit Testing III - keywords

- Concerned with
  - *Functional correctness and completeness*
  - *Error handling*
  - *Checking input values (parameter)*
  - *Correctness of output data (return values)*
  - *Optimizing algorithm and performance*

Unit Testing IV

- Each part tested individually
- All components tested at least once
  - Traceability matrix?
- Errors picked up earlier
- Scope is smaller, easier to fix errors

Unit Testing VI – Benefits..

- Unit testing allows the programmer to **refactor** code at a later date, and make sure the module still works correctly.
- By testing the parts of a program first and then testing the sum of its parts, integration testing becomes much easier.
- Unit testing provides a sort of living documentation of the system.
  - Evolution of the system = tests break!

Unit Testing - Guidelines

- Keep unit tests **small and fast**
  - *Entire test suite to be executed before every code check in.*
  - *Keeping the tests fast to reduce the development turnaround time.*
  -
- Unit tests should be **fully automated** and non-interactive!!!
  - *The test suite is normally executed on a regular basis and must be fully automated to be useful.*
  - *If the results require manual inspection to get results the tests are not proper unit tests.*

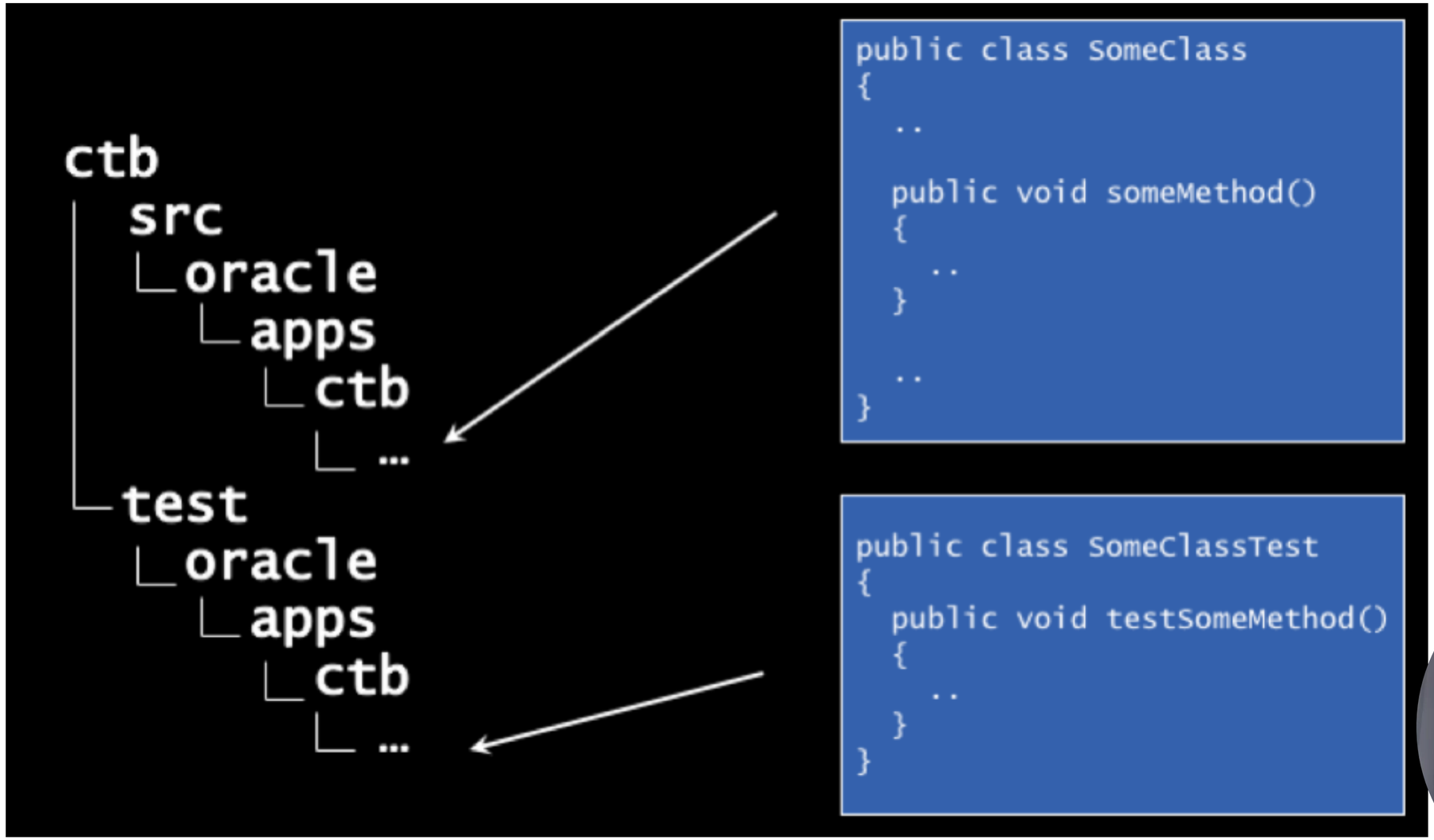
Unit Testing - Guidelines

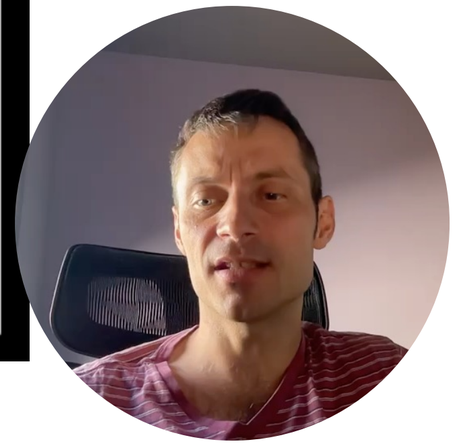
- Keep tests independent
  - *To ensure testing **robustness** and simplify **maintenance**, tests should never rely on other tests nor should they depend on the ordering in which tests are executed.*
- **Name** tests **properly**
  - *Make sure each test method test one distinct feature of the class being tested and name the test methods accordingly.*
  - ***Method** naming convention is test[whatItDoes] such as*
    - testSaveAs(), testAddListener(), testDeleteProperty() etc

Unit Testing - Guidelines

- Keep tests close to the **class** being tested
  - *If the class to test is **Foo** the test class should be called **FooTest** (not TestFoo) and kept in the same package as Foo.*
  - *In Java **reflect the packages in test folder**!*
  - *Make sure the build environment is configured so that the test classes doesn't make its way into production libraries or executables.*
  - *Maven src/test folders!!!*

Test folder


ctb
src
oracle
apps
ctb
…
test
oracle
apps
ctb
…
public class SomeClass
{
..
public void someMethod()
{
..
}
..
}
public class SomeClassTest
{
public void testSomeMethod()
{
..
}
}



Unit Testing - Guidelines

- Focus on execution coverage first
  - *All code tested/executed even trivial case*
  - *Then move to actual test coverage*
- Cover boundary cases
  - *-1,0,1, null*
  - *February 29*
  - *December 31/January 1*
- Provide a random generator for input
- Test each feature once

Unit Testing - Guidelines

- Use explicit asserts! - code constructs
  - *Always prefer assertEquals(a, b) to assertTrue(a == b)*
  - *Often you can accompany a message assertEquals(a, b, msg)*
- Provide negative tests
- Design code with testing in mind
- Do not connect to external resources!
  - *Mocking (fake components e.g., always return false)*
- Cost of testing (execution coverage 80%)
- Prioritize tests

Unit Testing - Guidelines

- Prepare test code for failures
- Write tests to reproduce bugs
- Know the limitations

Black-box testing: Test case selection

a) Input is valid across range of values

- *Developer selects test cases from 3 equivalence classes:*
  - Below the range
  - Within the range
  - Above the range

b) Input is only valid, if it is a member of a discrete set

- *Developer selects test cases from 2 equivalence classes:*
  - Valid discrete values
  - Invalid discrete values

- No rules, only guidelines.

Black box testing: An example

```
public class MyCalendar {

   public int getNumDaysInMonth(int month, int year)
     throws InvalidMonthException
   { … }
}
```

Representation for the month:
1: January, 2: February, …., 12: December

Representation for the year:
1904, … 1999, 2000,…, 2006, …

How many test cases do we need for the black box testing of getNumDaysInMonth()?

Example test data

Depends on the calendar: Gregorian calendar	Month parameter equivalence classes	year: parameter equivalence classes
	30 days 31 days February No month 0, 13, -1	Normal year Leap year: •*/4* •*/100* •*/400* Before christ/ After chri Before and after 200 **=> 12 test cases**

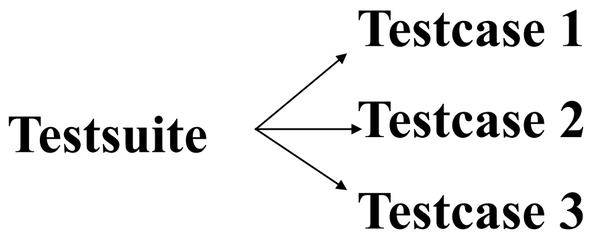
JUnit

- Framework – package of classes that lets you write tests for each method and easily run those tests.
- Instruments simplifying testing
  - *Assert, Report, TestRunner, IDE integration*

- Current version 5

Framework elements (JUnit 4)

- **TestCase**
  - *Base class for classes that contain tests*
- **assert*()**
  - *Method family to **check conditions***
- **TestSuite**
  - *Enables grouping several test cases*



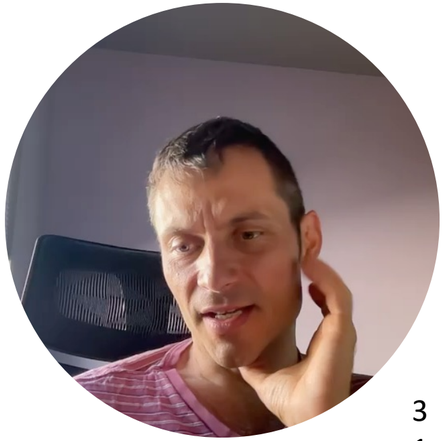
FOR BASICS AND LEARNING PURPOSES WE CONSIDER JUNIT 4

To understand the concepts

but.. we will use JUnit 5

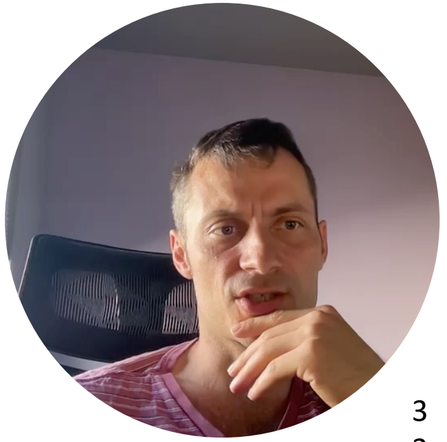
Assert*()

- Methods defined in the base class TestCase
- Names begin with “assert” and *are used in test methods*
  - *es.* ***assertTrue(“stack should be empty”, aStack.empty());***
- If the condition is false:
  - *test fails*
  - *execution skips the rest of the test method*
  - *the message (if any) is printed*
- If the condition is true:
  - *execution continues normally*



Assert*()

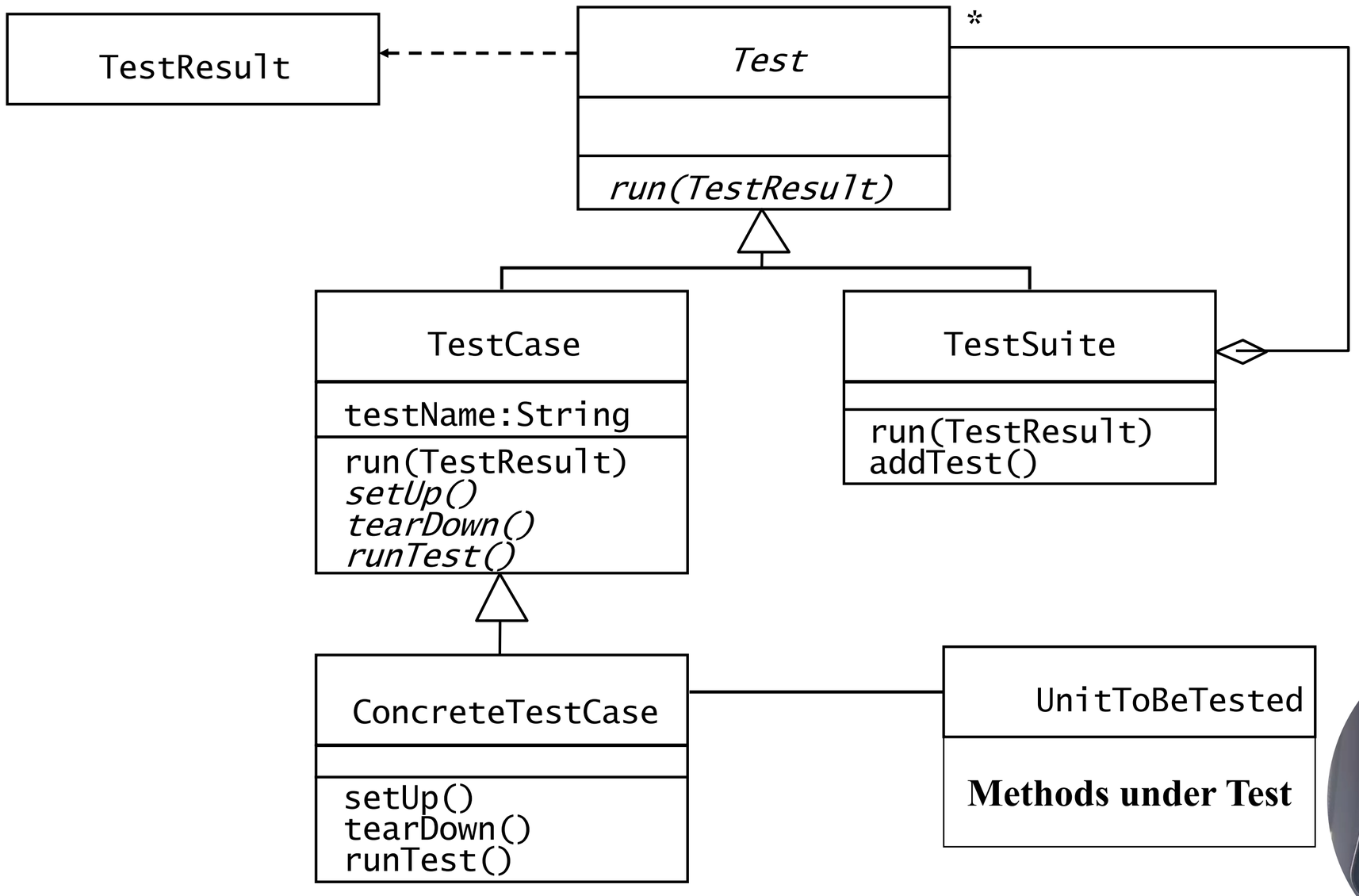
- for a boolean condition
  - ***assertTrue(“message for fail”, condition);***
  - ***assertFalse(“message”, condition);***
- for object, int, long, and byte values
  - ***assertEquals(expected_value, expression);***
- for float and double values
  - ***assertEquals(expected, expression, error);***
- for objects references
  - ***assertNull(reference)***
  - ***assertNotNull(reference)***
- **…**



Assert: example

```
public void testStack() {
     Stack aStack = new Stack();
     assertTrue("Stack should be empty!", aStack.isEmpty());
     aStack.push(10);
     assertTrue("Stack should not be empty!",!aStack.isEmpty());
     aStack.push(4);
     assertEquals(4, aStack.pop());
     assertEquals(10, aStack.pop());
}
```

```
class Stack {
 public boolean isEmpty(
 public void push(int i
 public int pop(){ ...
 …
}
```


TestResult
Test
*
run(TestResult)
TestCase
testName:String
run(TestResult)
setUp()
tearDown()
runTest()
TestSuite
run(TestResult)
addTest()
ConcreteTestCase
setUp()
tearDown()
runTest()
UnitToBeTested
Methods under Test

We expect a normal behavior …

```
class TheClass {
 public void method(String p)
    throws PossibleException
  { /*... */ }
}
```

We expect a normal behavior …

```
try {
      // We call the method with correct parameters
      object.method("Parameter");
      assertTrue(true); // OK
} catch(PossibleException e){
      fail("method should not fail !!!");
}
```

```
class TheClass {
 public void method(String p)
     throws PossibleException
  { /*... */ }
}
```

We expect an exception …

```
try {
   // we call the method with wrong parameters
   object.method(null);
   fail("method should fail!!");
} catch(PossibleException e){
     assertTrue(true); // OK
}
```

```
class TheClass {
 public void method(String p)
     throws PossibleException
  { /*... */ }
}
```

SetUp() and tearDown()

setUp() method initialize object(s) under test.

- *called before every test method*

tearDown() method release object(s) under test

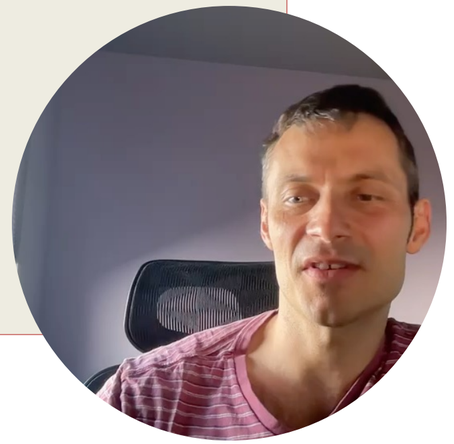
- *called after every test case method.*

```
ShoppingCart cart;
Book book;

protected void setUp() {
 cart = new ShoppingCart();
 book = new Book("JUnit", 29.95);
 cart.addItem(book);
}
…
```

JUnit 4 is Easy

```
…
public void testInvalidPersonName() {
   person.setFirstName(null);
   person.setLastName(“Smith”);
   try {
      personService.createPerson(person);
      fail(“An invalid person name error should be
   thrown”);
   } catch (InvalidPersonName ipn) {
      // Exception expected
   }
}
```

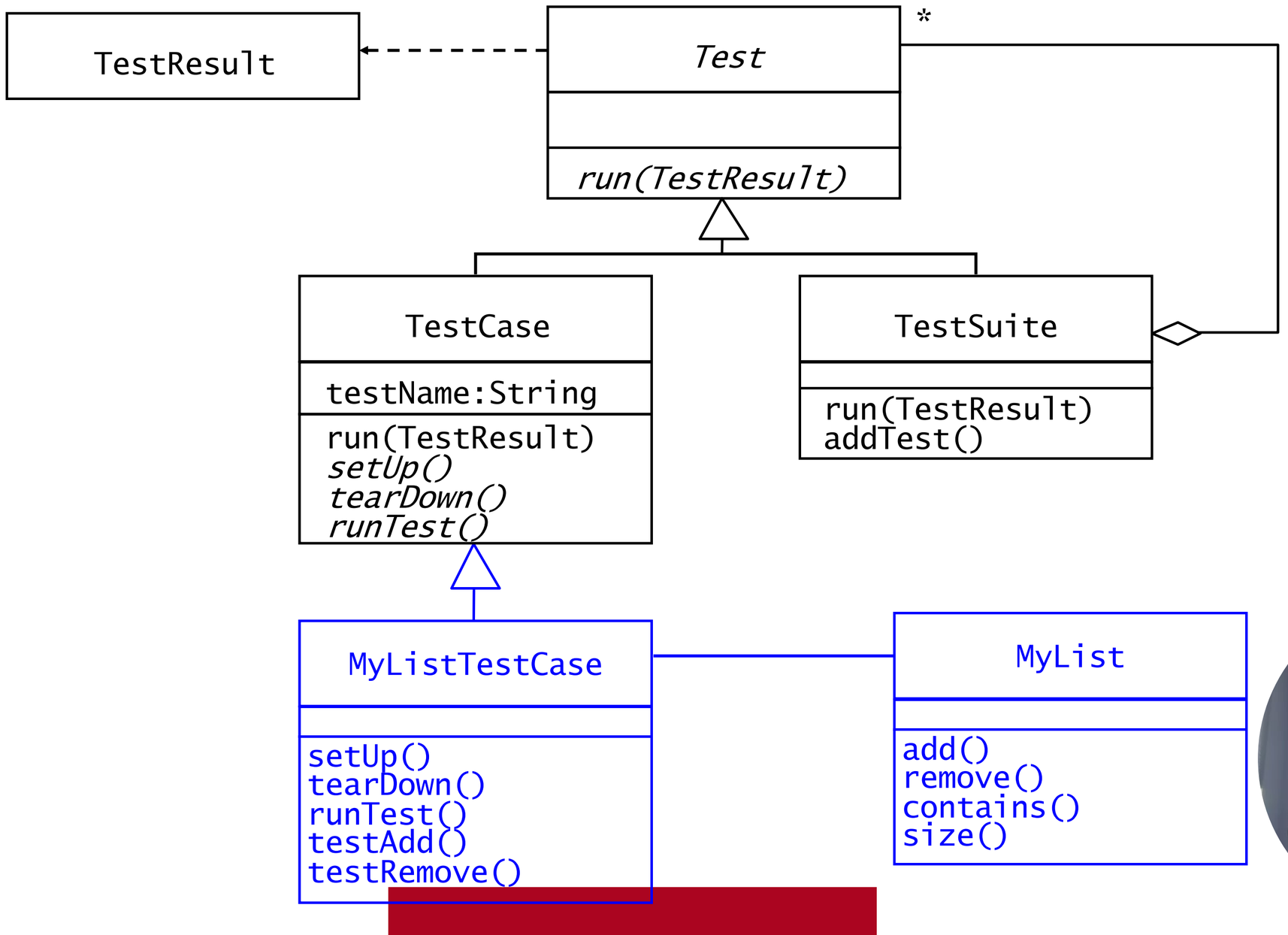
…



An example JUnit 4: Testing MyList

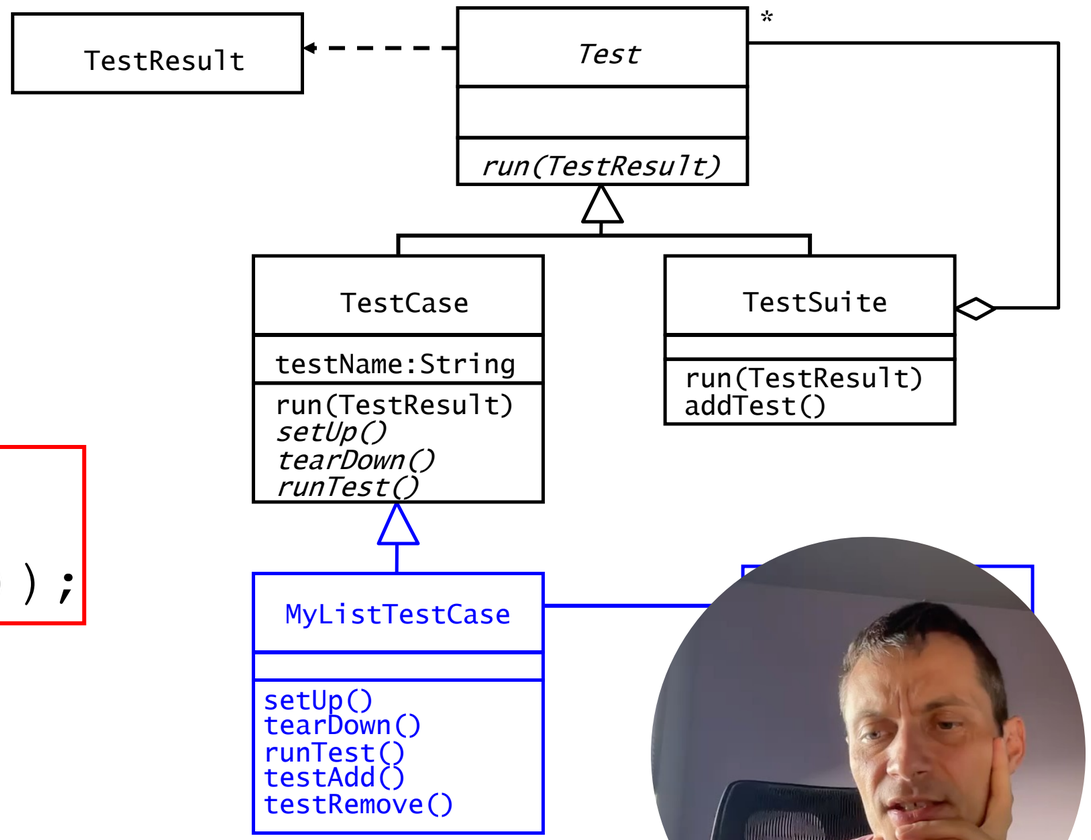
- Unit to be tested
  - *MyList*
- Methods under test
  - *add()*
  - *remove()*
  - *contains()*
  - *size()*
- Concrete Test case
  - *MyListTestCase*

JUnit 4


TestResult
Test
*
run(TestResult)
TestCase
testName:String
run(TestResult)
setUp()
tearDown()
runTest()
TestSuite
run(TestResult)
addTest()
MyListTestCase
setUp()
tearDown()
runTest()
testAdd()
testRemove()
MyList
add()
remove()
contains()
size()

Can we do better?

```
public class MyListTestCase extends TestCase {

public MyListTestCase(String name) {
   super(name);
}
public void testAdd() {
    // Set up the test
   List aList = new MyList();
   String anElement = "a string";

    // Perform the test
   aList.add(anElement);

    // Check if test succeeded
   assertTrue(aList.size() == 1);
   assertTrue(aList.contains(anElement));
}
protected void runTest() {
   testAdd();
}
}
```



Writing Fixtures & Test Cases

```
public class MyListTestCase extends TestCase {
// …
private MyList aList;
private String anElement;
public void setUp() {
   aList = new MyList();
   anElement = “a string”;
}
```

Test Fixture

```
public void testAdd() {
   aList.add(anElement);
   assertTrue(aList.size() == 1);
   assertTrue(aList.contains(anElement));
}
```

Test Case

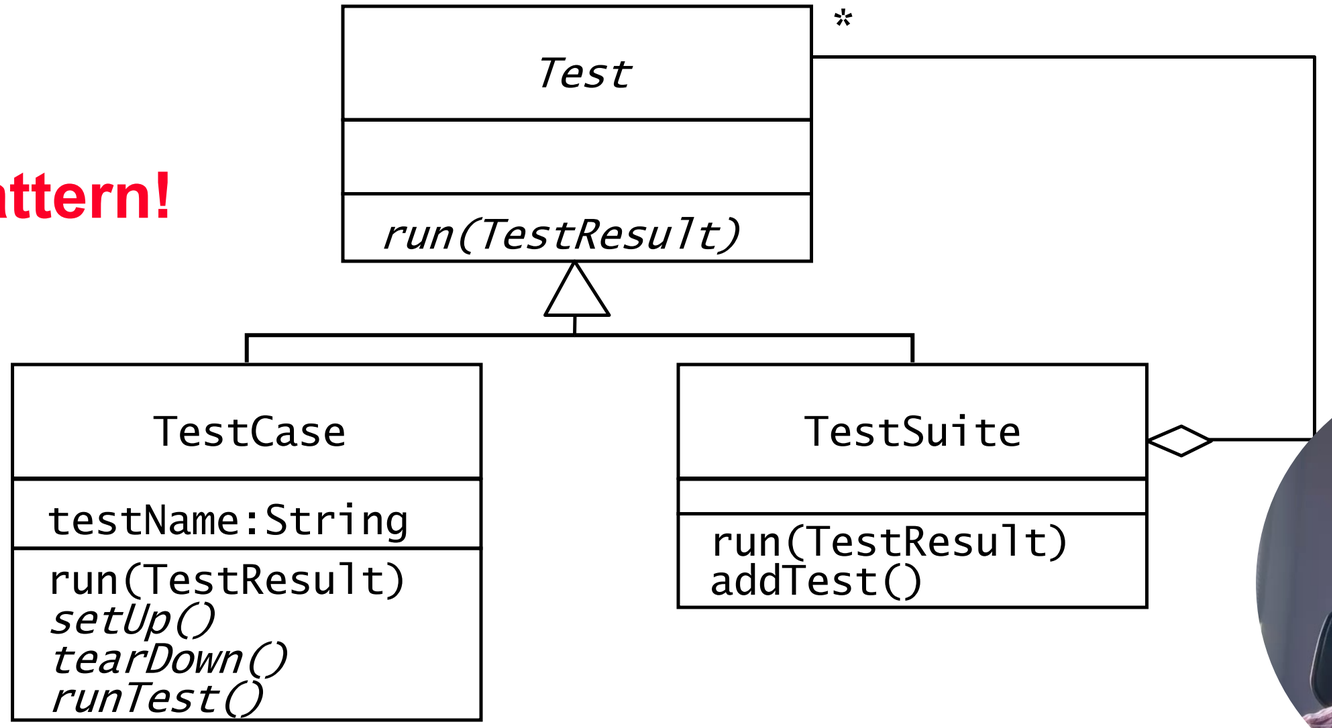
```
public void testRemove() {
   aList.add(anElement);
   aList.remove(anElement);
   assertTrue(aList.size() == 0);
   assertFalse(aList.contains(anElement));
}
```

Test Case

Collecting TestCases into TestSuites

```
public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(new MyListTestCase("testAdd"));
        suite.addTest(new MyListTestCase("testRemove"));
        return suite;
}
```

Composite Pattern!



THAT IS IT BUT

..JUnit 5

JUnit 5 Installation

- Maven!

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.0.1</version>
    <scope>test</scope>
</dependency>
```

JUnit Hello World

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

class FirstJUnit5Test {

    @Test
    void myFirstTest() {
        assertEquals(2, 1 + 1);
    }
}
```

JUnit Running - Maven

```
$ mvn clean test
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running edu.baylor.ecs.FirstJUnit5Test
This test method should be run
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.033 sec -
in edu.baylor.ecs.FirstJUnit5Test
Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

JUnit Example

```
import static org.junit.jupiter.api.Assertions.fail;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;
..
//<next slide>
```

JUnit Example

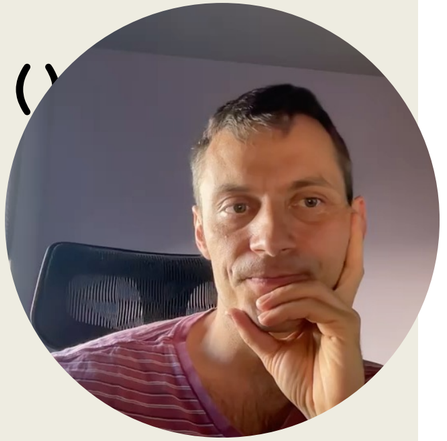
Neither test classes nor test methods need to be public.

```
class StandardTests {

 @BeforeAll
 static void initAll() {..}

 @BeforeEach
 void init() {..}

 @Test
 void succeedingTest() {..}

 @Test
 void failingTest() {
   fail("a failing test");
 }

 @Test
 @Disabled("for demonstration")
 void skippedTest() {
  // not executed
 }
 @AfterEach
 void tearDown() {..}

 @AfterAll
 static void tearDownAl
}
```

JUnit Example

Neither test classes nor test methods need to be public.

```
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

@DisplayName("A special test case")
class DisplayNameDemo {

 @Test @DisplayName("Custom test name containing spaces")
 void testWithDisplayNameContainingSpaces() {     }

 @Test @DisplayName("ﾉ°□°）ﾉ")
 void testWithDisplayNameContainingSpecialCharacters()

 @Test @DisplayName("😱")     void
 testWithDisplayNameContainingEmoji() {     }
}
```



JUnit Example Snippet To Run

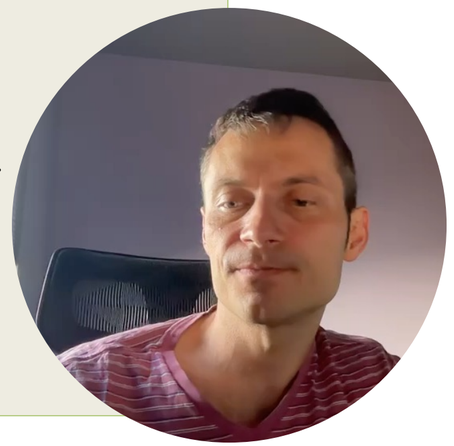
```
mvn test
mvn -Dtest=Ex1StandardTests test
mvn -Dtest=Ex1StandardTests#succeedingTest test
```

http://maven.apache.org/plugins-archives/maven-surefire-plugin-2.12.4/examples/single-tes

JUnit Example Snippet

```
@RunWith(JUnitPlatform.class)
@SelectPackages("edu.baylor.cs.junit.hooks")
@IncludeClassNamePatterns("^.*Tests?$")
public class Ex6_2Suite {}
```

```
@RunWith(JUnitPlatform.class)
@SelectClasses( {ExampleTest.class,
ExampleTest2.class, ExampleTest3.class }
public class Ex6_2Suite2 {}
```



JUnit Example Snippet

```
@Test
public void convertToIntNullParameterAssertThrows() {
  String st = null;
  assertThrows(IllegalArgumentException.class, () -> {
    StringUtils.convertToInt(st);
  });
}
```

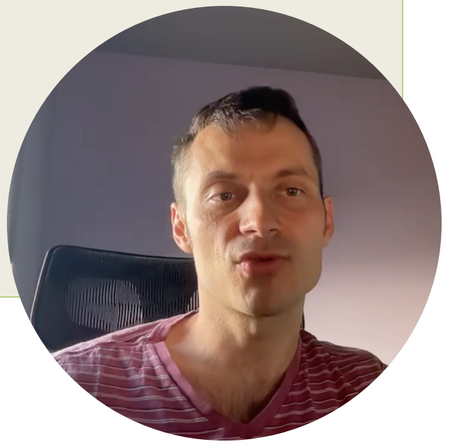
JUnit Notes

- Run test
  - *Do not expect test order execution*
  - *If you must then use*

```
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
```

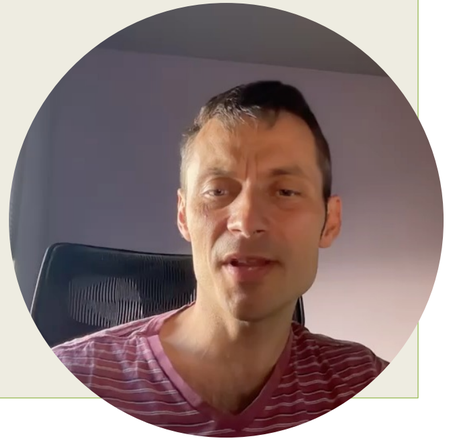
JUnit Example Snippet Params

```
public class Ex2ValueSourceExampleTest {



   @ParameterizedTest
   @ValueSource(strings = {"Hello", "World"})
   void shouldPassNonNullMessageAsMethodParameter(String m) {
         assertNotNull(m);
   }
}
```



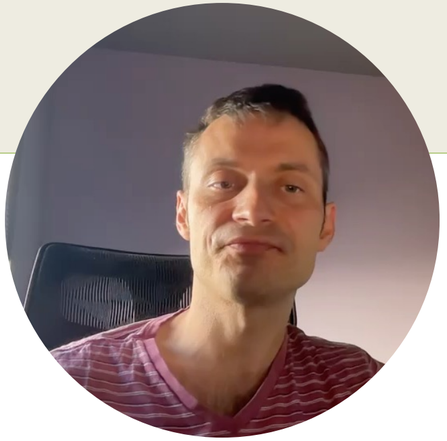
JUnit Example Snippet Params

```
@ParameterizedTest(name = "{index} => a={0}, b={1}, sum={2}")
@MethodSource("sumProvider")
void sum(int a, int b, int sum) {
	assertEquals(sum, a + b);
}

@SuppressWarnings("unused")
private static Stream<Arguments> sumProvider() {
	return Stream.of(
			Arguments.of(1, 1, 2),
			Arguments.of(2, 3, 5)
	);
}
```



JUnit Example Snippet Params

```
@DisplayName("Should calculate the correct sum")
@ParameterizedTest(name = "{index} => a={0}, b={1}, sum={2}")
@CsvSource({
                "1, 1, 2",
                "2, 3, 5"
})
void sum(int a, int b, int sum) {
        assertEquals(sum, a + b);
}
```

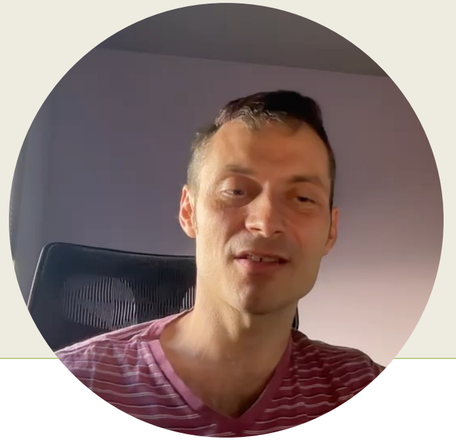


JUnit Example Snippet Params

```
@DisplayName("Should calculate the correct sum")
@ParameterizedTest(name = "{index} => a={0}, b={1}, sum={2}")
@CsvFileSource(resources = "/test-data.csv")
void sum(int a, int b, int sum) {
        assertEquals(sum, a + b);
}
```

JUnit Example Snippet Params

```
@Tag("debug")
public class Ex8TagTest {
    @Test
    @Tag("production")
    void testCaseA(TestInfo testInfo) {
      fail("production");
    }

    @Test
    void testCaseB(TestInfo testInfo) {
      fail("debug");
    }
}
```



JUnit Example Snippet Params

```
@RunWith(JUnitPlatform.class)
@SelectPackages("edu.baylor.cs.junit.tag")
@IncludeTags("debug")
@IncludeClassNamePatterns({"^.*Tests?$"})
public class Ex8_2RunTagDebug
{
}
```