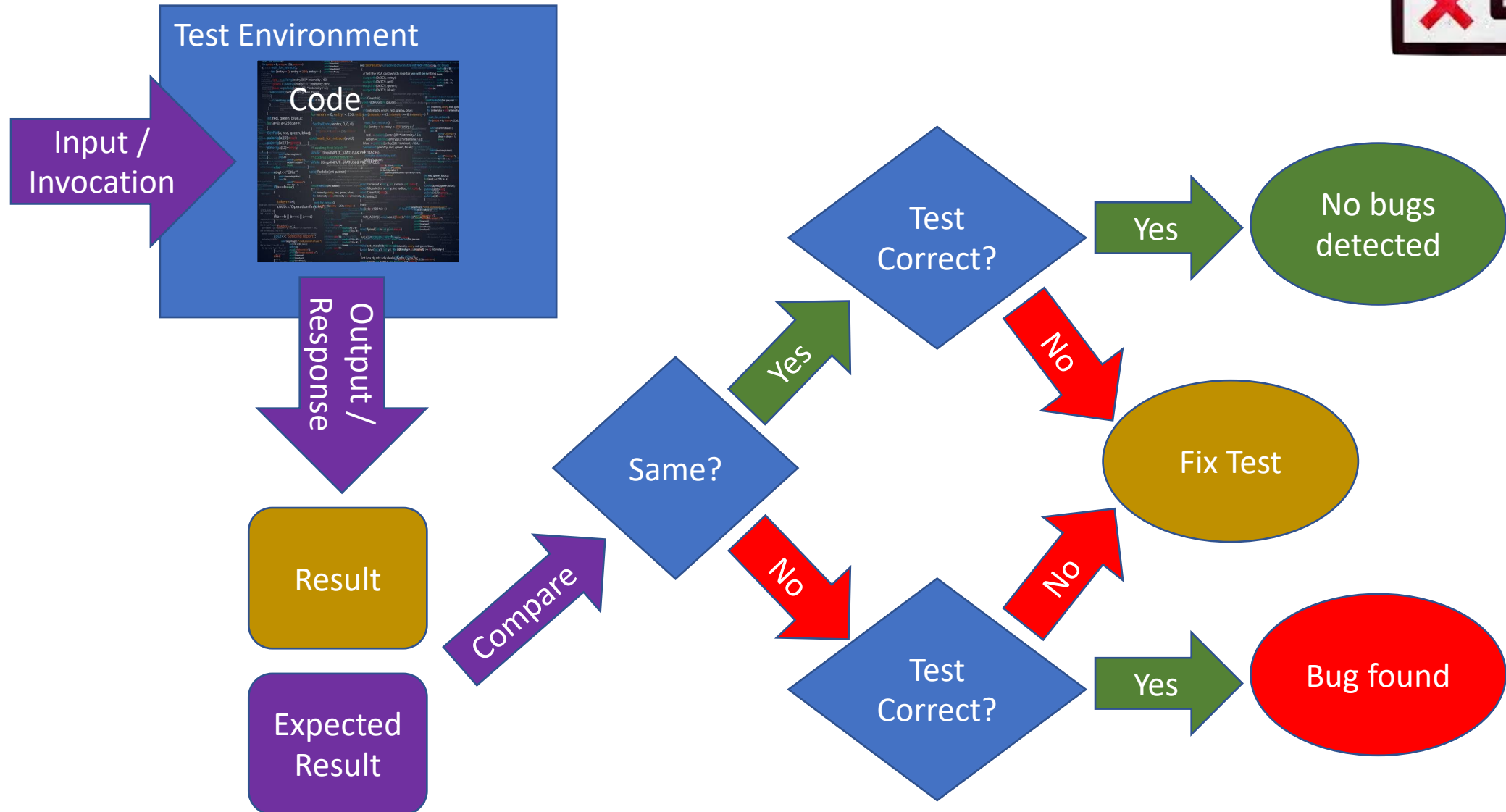CommitStrip.com

Testing 101
Introduction to Unit Testing and JUnit

CSCI 2134: Software Development

Agenda

- Lecture Contents
  - What is a Test
  - What is Unit Testing
  - Introduction to JUnit
  - Whitebox and Blackbox Testing
- Brightspace Quiz
- Readings:
  - This Lecture: Chapter 22
  - Next Lecture: Chapter 22

What Is a Test?

Test Environment

Code

Input / Invocation

Output / Response

Result

Expected Result

Compare

Same?

Yes — Test Correct? — Yes — No bugs detected

No — Fix Test

No — Test Correct? — Yes — Bug found

No — Fix Test

Testing: A Blast from the Past

- CSCI 1105 and CSCI 1110
  - Input/Output Tests
    - Codio tests
    - Mimir tests
  - Runner / Class tests
- CSCI 2110 (Winter)
  - Input/Output Tests
    - Mimir tests
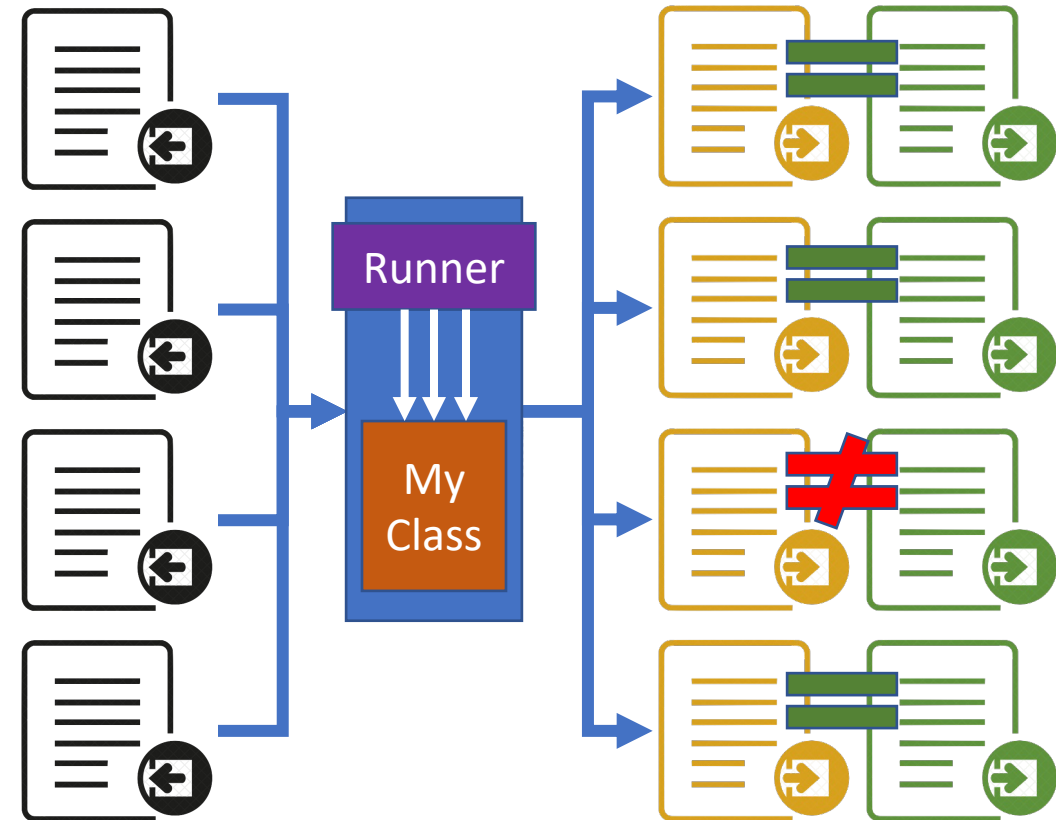
Example from a CSCI 2110 Assignment

Input

```
502788 32
503612 33
503775 34
```

```java
public ArrayList computeTraffic(Scanner stream) {
  ArrayList list = new ArrayList();

  if( stream.hasNextLong() ) {
    curTime = stream.nextLong();
    curSpeed = stream.nextInt();

    while( stream.hasNextLong() ) {
      list.add( nextVehicle( stream ) );
    }
  }
  return list;
}

...
```

Actual Output

```
502: heavy truck, 1 2
```

Expected Output

```
502: Heavy truck, 1 2
```

Compare

Same?

No

Bug found

6

Unit Testing and JUnit

- Unit testing involves testing small parts of a program written by a single developer
  - Methods
  - Classes (small)
- Unit testing is made easier using the JUnit framework
- JUnit provides a way to
  - Test each method of a class
  - Run the tests at a click of a button
  - Debug failed tests
  - Write tests as you develop
- We will be using JUnit 5

JUnit in a Nutshell

```
public MyClassTest {
  ...
  public int method1Test1(…) { ... }
✖public int method1Test2(…) { ... }
  public int method1Test3(…) { ... }

✖public int method2Test1(…) { ... }
  public int method2Test2(…) { ... }

  public int method3Test1(…) { ... }
  public int method3Test2(…) { ... }

✖public int method4Test(…) { ... }

✖public int method5Test1(…) { ... }
  public int methoD5Test2(…) { ... }
  ...
}
```

JUnit

```
public MyClass {
  ...
  public int method1(…) { ... }

  public int method2(…) { ... }

  public int method3(…) { ... }

  public int method4(…) { ... }

  public int method5(…) { ... }

  public int method6(…) { ... }
}
```

Example of Using JUnit in IntelliJ

MatrixTest.java

```java
MatrixTest.java
1   import ...
7   class MatrixTest {
8       private final static String simpleMatrix = "2 2 1 2 3 4"; // [
9                                                               // [
10      private final static String nonSqMatrix = "3 2 1 2 3 4 5 6"; //
11      // [ 3 4 ]
12
13      @Test
14      void getElem() {
15          Matrix m = new Matrix(new Scanner(simpleMatrix));
16          assertEquals( expected: 2, m.getElem( i: 1, j: 2), message: "getE
17      }
18
19      @Test
20      void setElem() {
21          Matrix m = new Matrix(new Scanner(simpleMatrix));
22          m.setElem( i: 2, j: 1, v: 5);
23          assertEquals( expected: 5, m.getElem( i: 2, j: 1), message: "se
24      }
25
26      @Test
27      void negate() {
28          Matrix m = new Matrix(new Scanner(simpleMatrix));
29          m.negate();
30          assertEquals( expected: -1, m.getElem( i: 1, j: 1), message: "negate() did not negate element (1,1
31          assertEquals( expected: -2, m.getElem( i: 1, j: 2), message: "negate() did not negate element (1,2
```
MatrixTest

Run: ◀▷ MatrixTest ✕

| Test Results	140 ms
▼ MatrixTest	140 ms
✔ equal2s()	43 ms
❗ equals()	19 ms
✖ negate()	22 ms
❗ multiplyWithMatrix()	13 ms
❗ getElem()	7 ms
❗ add()	18 ms
❗ multiplyWithScalar()	5 ms
✔ getHeight()	5 ms
✖ getWidth()	4 ms
❗ setElem()	4 ms

Terminal   ▤ 0: Messages   ▶ 4: Run   ☰ 6: TODO

Matrix.java

```
2D array stores matrix of size height x width
number of rows in the matrix
number of columns in the matrix

atrix of size m x n.
matrix
matrix

initialize height and width fields.

assed matrix
be cloned

to be cloned and instantiate array.

41      height = mtx.getHeight();
42      width = mtx.getWidth();
43      matrix = new double[height][width];
```
Matrix

How to Create JUnit Test Class in IntelliJ 1

- Create a **test** directory for the project (if one does not exist)
  - Create a directory called **test** in the main project directory
  - Map the the `test` directory to be Test Sources Directory
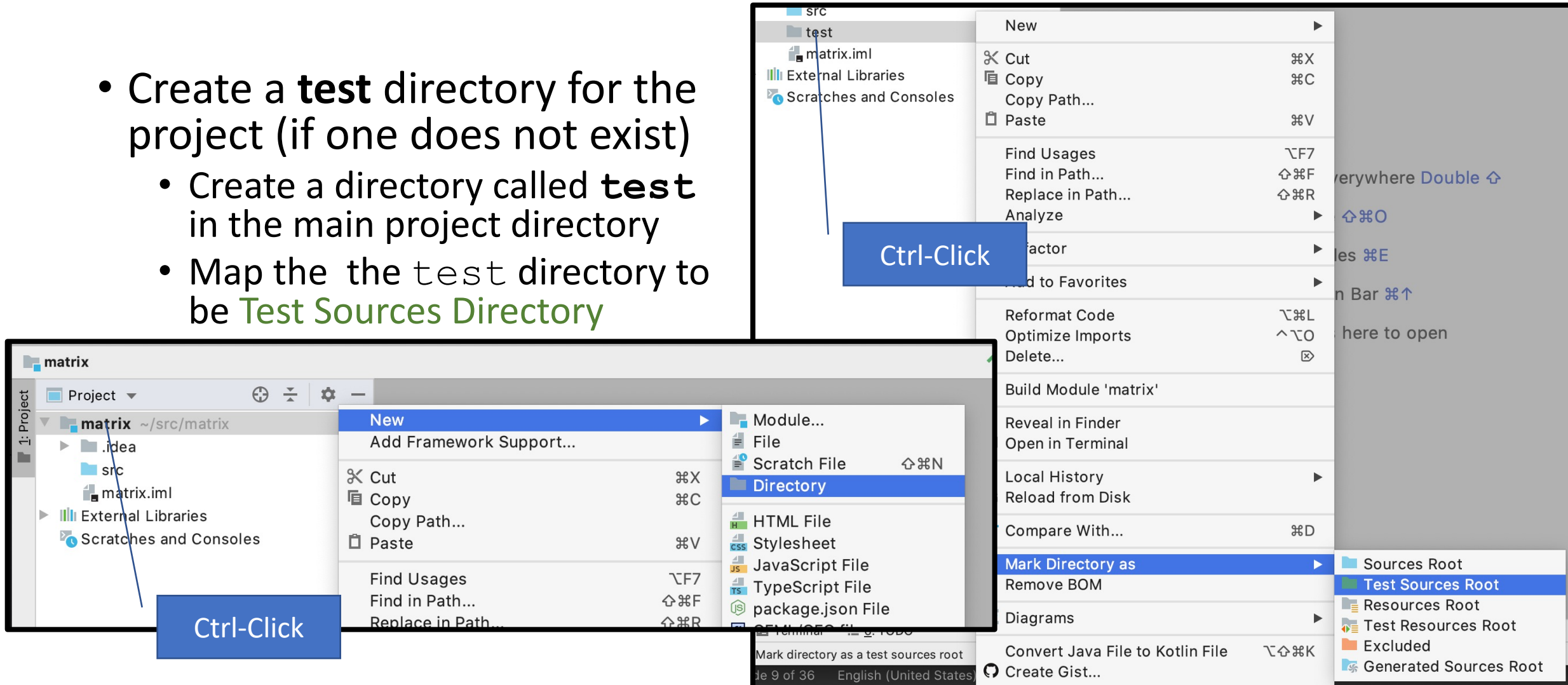
How to Create JUnit Test Class in IntelliJ 2

```
16        * is easy to manipulate.
17        💡✔️
18     public class Matrix {
19          private double [][] matrix;   // 2D array stores matrix of size height x width
20          int height;                    // number of rows in the matrix
21          int width;                     // number of columns in the matrix
22
```

View	Navigate	Code	Analyze	Refacto

Class...	⌘O
File...	⇧⌘O
Symbol...	⌥⌘O
Custom Folding...	⌥⌘.
Line/Column...	⌘L
← Back	⌘[
→ Forward	⌘]
Last Edit Location	⇧⌘⌫
Next Edit Location	
Bookmarks	▶
Select In...	⌥F1
Jump to Navigation Bar	⌘↑
Declaration or Usages	⌘B
Implementation(s)	⌥⌘B
Type Declaration	⇧⌘B
Super Class	⌘U
Test	⇧⌘T
Related Symbol...	^⌘↑
File Structure	⌘F12
Type Hierarchy	^H
Method Hierarchy	⇧⌘H
Call Hierarchy	^⌥H
Next Highlighted Error	F2
Previous Highlighted Error	⇧F2
Next Method	^↓
Previous Method	^↑

- Select the class you wish to test
  - Open the desired `.java` file in the **src** directory
  - Highlight the public class in the file and click on **Navigate → Test**
  - Click on **Create New Test** in the pop-up menu

```
do    Choose Test for Matrix (1 found)      🗖  o
ht    © MatrixTest                Lab4 📂   ow
h;    💡 Create New Test...                  o
      Press ^⇧R to run selected tests
```
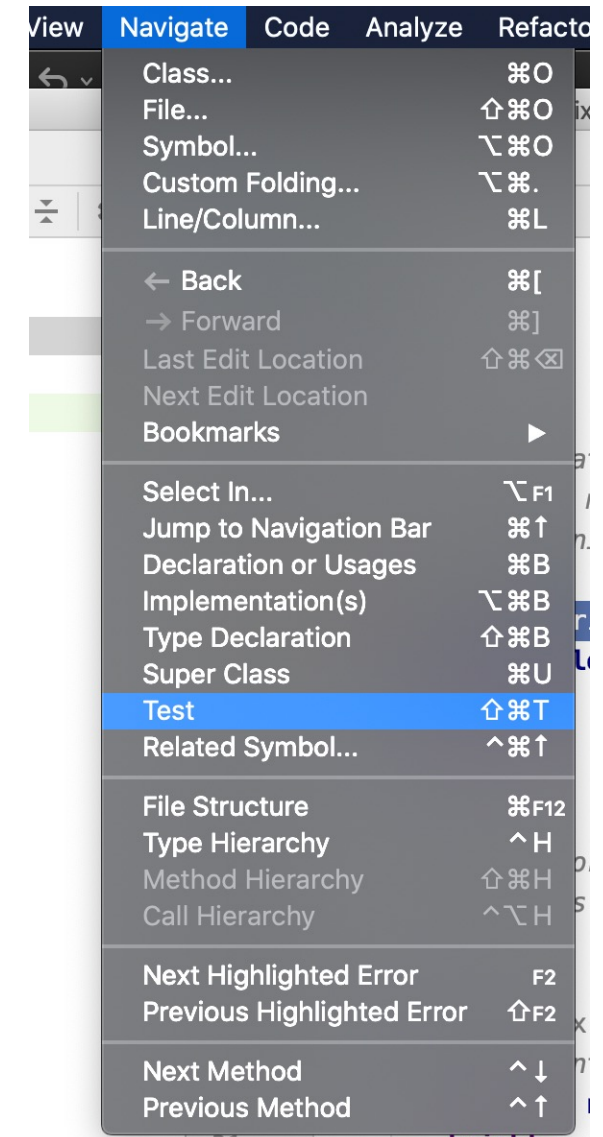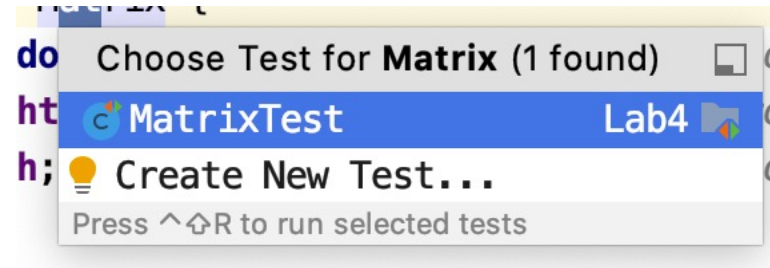
How to Create JUnit Test Class in IntelliJ 3

Select

Fix

- Create the JUnit class
  - When the dialog box appears, ensure **JUnit5** is selected
  - If the JUnit library is not found, click on the **Fix** button right below the selector
    - Check the **Download to: …** box in the **Download Library** dialog
    - Click on **OK**
  - The test class name is the class' name, suffixed with `Test`
  - Select all the methods that you wish to test
    - Typically all the public and some private
  - Click on **OK**

Select

Select all

Create Test

Testing library: ◀▶ JUnit5

💡 JUnit5 library not found in the module   Fix

Class name: MatrixTest

Superclass: ▼ ...

Destination package: ▼ ...

Generate: ☐ setUp/@Before
          ☐ tearDown/@After

Generate test methods for: ☐ Show inherited methods

	Member
☐ m 🔓	equals(a:Matrix):boolean
☐ m 🔓	negate():void
☐ m 🔓	add(b:Matrix, res:Matrix):Matrix
☐ m 🔓	multiplyWithScalar(s:double, res:Matrix):Matrix
☐ m 🔓	multiplyWithMatrix(b:Matrix, res:Matrix):Matrix
☐ m 🔓	write(out:PrintStream):void
☐ m 🔓	getElem(i:int, j:int):double
☐ m 🔓	setElem(i:int, j:int, v:double):void

?   Cancel   OK

Download Library from Maven Repository

org.junit.jupiter:junit-jupiter:5.4.2 ▼ 🔍 ⟳ Found: 169
                                              Showing: 169

keyword or class name to search by or exact Maven coordinates, i.e. 'spring', 'Logger' or 'ant:ant-junit:1.6.5'

☑ Download to: /Users/abrodsky/Teach/2110/assignments/assn3/code/lib 📁

    ☑ Transitive dependencies ☐ Sources ☐ JavaDocs ☐ Annotations

                          Cancel   **OK**

How to Create JUnit Test Class in IntelliJ

- Create a **test** directory for the project (if one does not exist)
  - Create a directory called `test` in the main project directory
  - Map the the `test` directory to be Test Sources Directory
- Select the class you wish to test
  - Open the desired `.java` file in the **src** directory
  - Highlight the public class in the file and click on **Navigate → Test**
  - Click on **Create Test** in the pop-up menu
- Create the JUnit class
  - When the dialog box appears, ensure **JUnit5** is selected
  - If the JUnit library is not found, click on the **Fix** button right below the selector
    - Check the **Download to: …** box in the **Download Library** dialog
    - Click on **OK**
  - The test class name is the class' name, suffixed with `Test`
  - Select all the methods that you wish to test
    - Typically all the public and some private
  - Click on **OK**
- The test class will be created and opened in IntelliJ

```
import ...

class MatrixTest {

    @Test
    void testEquals() {
    }


    @Test
    void negate() {
    }


    @Test
    void add() {
    }


    @Test
    void multiplyWithScalar() {
    }


    @Test
    void multiplyWithMatrix() {
    }


    @Test
    void write() {
    }

    @Test

MatrixTest  ›  getWidth()
```

Creating JUnit5 Tests in IntelliJ
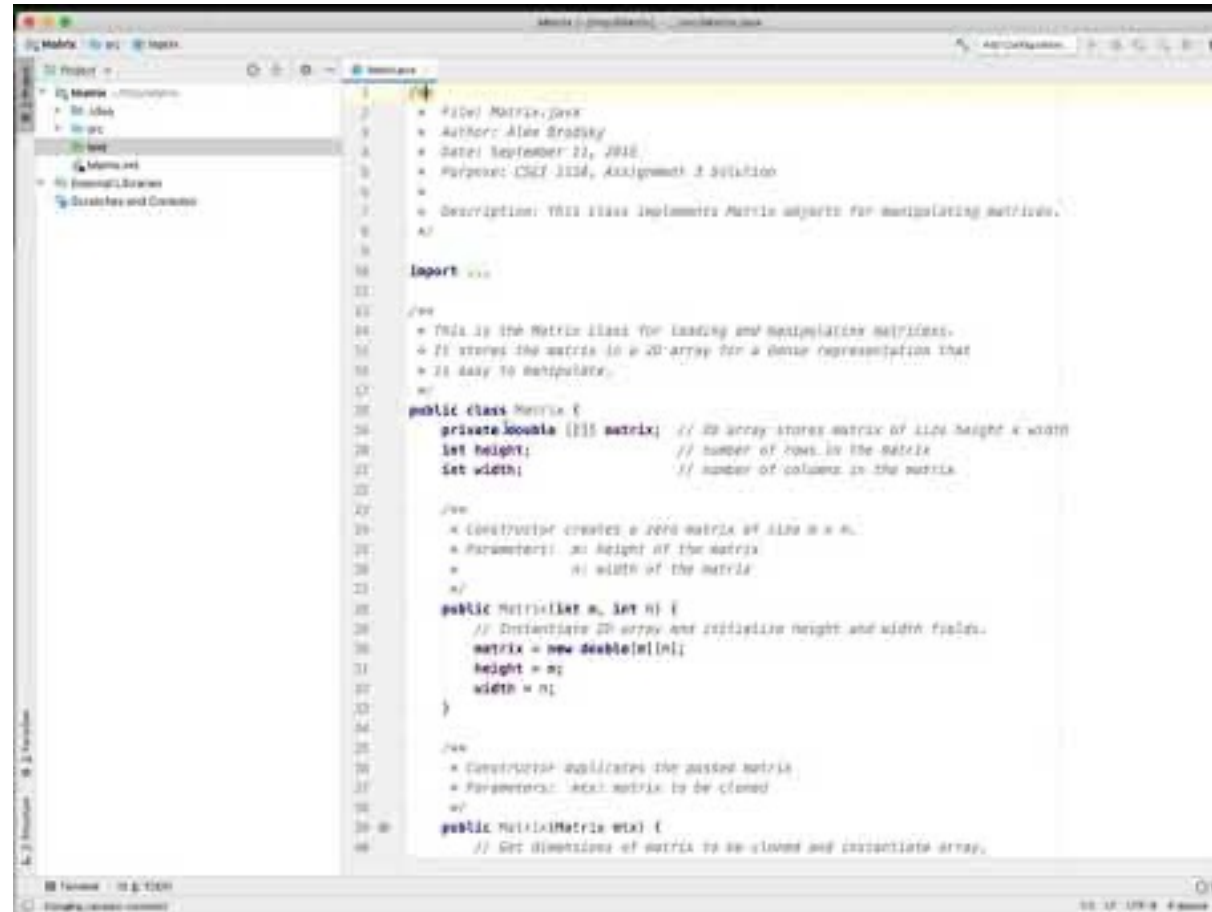
Anatomy of a Test Method in JUnit 5

- A JUnit *test method* is intended to test a method in the target class
  - Typically with the same name
- A test method contains several parts:
  - **Setup**: instantiation of necessary objects and data structures
    - Typically, an object of the class being tested is instantiated
  - **Call**: call to the target method
  - **Evaluation**: the result of the call to the target method is evaluated.
    - This involves a call to one or more assertion functions
- JUnit provides other mechanisms for instantiating and setting up classes and objects once for an entire set of tests

Example of *MatrixTest* Class

```
1    ⊞import ...
7 ⓒ  class MatrixTest {
8        private final static String simpleMatrix = "2 2 1 2 3 4"; // [ 1 2 ]
9                                                                  // [ 3 4 ]
10       private final static String nonSqMatrix = "3 2 1 2 3 4 5 6"; // [ 1 2 3 ]
11                                                                      // [ 4 5 6 ]
12
13       @Test
14 ⓒ    void getElem() {
15         Matrix m = new Matrix(new Scanner(simpleMatrix));
16         assertEquals( expected: 2, m.getElem( i: 1, j: 2),
17                 message: "getElem() did not return correct value");
18       }
19
20       @Test
21 ⓒ    void setElem() {
22         Matrix m = new Matrix(new Scanner(simpleMatrix));
23         m.setElem( i: 2,  j: 1,  v: 5);
24         assertEquals( expected: 5, m.getElem( i: 2, j: 1),
25                 message: "setElem() may not have set correct value");
26       }
27
```

Data used by the tests (a 2x2 and 2x3 matrices

Setup: Instantiate a *Matrix* object

Call: the `getElem()` method on *Matrix*

Evaluation: compare return value of call

Setup: Instantiate a *Matrix* object

Call: the `setElem()` method on *Matrix*

Evaluation: compare return value of call

Aside on Assert

- Idea: JUnit provides assert functions to allow the test method to communicated the success of the test

- The *org.junit.jupiter.api.Assertions* class provides hundreds of assertions to use (static methods).

- See online documentation for list of all the possible assertion

Assertion	Description
assertTrue(boolean b)	Test fails if **b** is false
assertFalse(boolean b)	Test fails if **b** is true
assertNotNull(Object o)	Test fails if **o** is null
assertNull(Object o)	Test fails if **o** is not null
assertEquals(x,y)	Test fails if x is not equal to y
assertNotEquals(x,y)	Test fails if x equals y
assertArrayEquals(a, b)	Test fails if arrays are not equal

How Do We Design a Test?

- Two General Approaches
  - White Box Testing
  - Black Box Testing

- White Box Testing
  - Test based on implementation of code
  - Typically try to test every code path through the method

- Black Box Testing
  - Test based on specification of what code should do
  - Use the description of class and method(s) to test if specification is being met.

Example: A White Box Test

```
/**
 * Return element E{i,j} of the matrix.  If i or j is less than 1
 * or greater than height or width, respectively, the behaviour is
 * undefined.
 * Parameters:  i: row of entry
 *              j: column of entry
 */
public double getElem(int i, int j) {
    // return corresponding entry in matrix.
    return matrix[i][j];
}
```

- **Observation**: Single path through method
- Proposed test:
  - Initialize *Matrix* object with known values
  - Perform **getElem()**
  - Expectation: It should return the known value
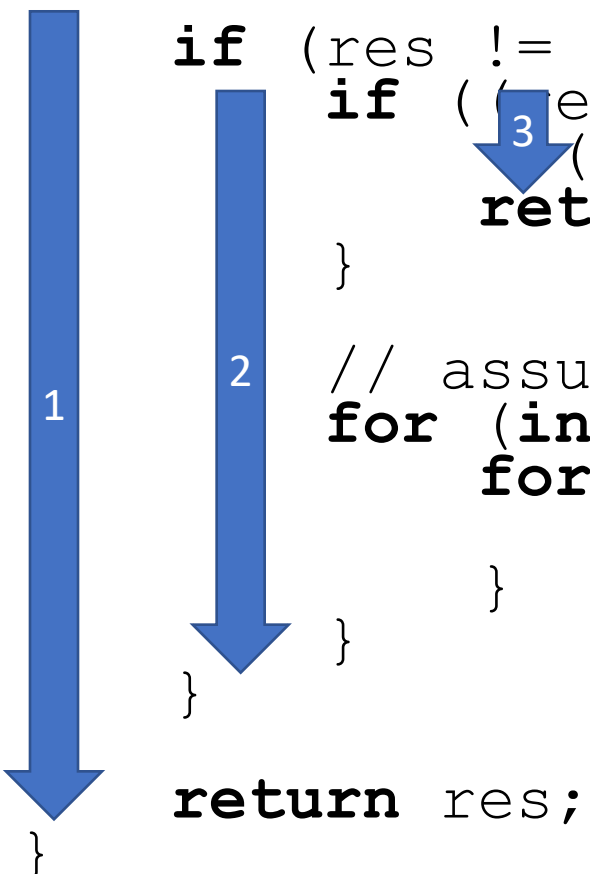
```
@Test
void getElem() {
  Matrix m = new Matrix(new Scanner(simpleMatrix));
  assertEquals( expected: 2, m.getElem( i: 1, j: 2),
          message: "getElem() did not return correct value");
}
```

An Intro to Structured Basis Testing

- Idea:
  - For each method create tests that test each part of the code in the method
  - Each part of the code needs to be tested once
  - A kind of white box testing
- Algorithm
  - Start with one test case at start of method
  - Add a test case for each
    - Loop
    - if statement
    - Part of a Boolean condition
    - Every code branch
- Observation: Smaller methods need fewer test cases

Example: How many tests do we need?

```
public Matrix multiplyWithScalar(double s, Matrix res) {

    if (res != null) {
      if ((res.getHeight() != height) ||
            (res.getWidth() != width)) {
        return null;
      }
    }

    // assume height, width > 0
    for (int i = 0; i < height; i++) {
      for (int j = 0; i < width; j++) {
        res.matrix[i][j] = s * matrix[i][j];
      }
    }

    return res;
}
```

Test Cases:
1. res == null; expected return: null
2. res.height == height, res.width = width, expected return s*this
3. res != null, res.height != height, expected return null
4. res.height == height, res.width != width, expected return null

21

Test Cases in JUnit

Test Cases	Test code
`res == null`	`Matrix m = new Matrix(1,1)` **`assertNull`**`(m.multiplyWithScalar(1, null))`

`res != null`
`res.height == height,`
`res.width == width`
	`Matrix m = new Matrix(…) // 2 x 2 matrix`
	`Matrix n = new Matrix(2,2)`
	`n = m.multiplyWithScalar(2, n))`
	`assertNotNull``(n)`
	`assertEquals``(2 * m.getElem(1,1) == n.getElem(1,1))`
	`assertEquals``(2 * m.getElem(2,1) == n.getElem(2,1))`
	`assertEquals``(2 * m.getElem(1,2) == n.getElem(1,2))`
	`assertEquals``(2 * m.getElem(2,2) == n.getElem(2,2))`

`res != null,`
`res.height != height`
	`Matrix m = new Matrix(1,1)`
	`Matrix n = new Matrix(2,1)`
	`assertNull``(m.multiplyWithScalar(1, n))`

`res != null`
`res.height == height`
`res.width != width`
	`Matrix m = new Matrix(1,1)`
	`Matrix n = new Matrix(1,2)`
	`assertNull``(m.multiplyWithScalar(1, n))`

Black Box Testing: Specification-based Testing

- Idea:
  - For each method create tests that are based on the description of the method
- Need to create tests for
  - Common cases
  - Boundary cases
  - Possibly, exceptional cases

Example: A Black Box Test

```
/**
 * Return element E{i,j} of the matrix.  If i or j is less than 1
 * or greater than height or width, respectively, the behaviour is
 * undefined.
 * Parameters:  i: row of entry
 *              j: column of entry
 */
public double getElem(int i, int j) {




}
```

- **Cases to Consider:**
  - i and/or j on the edge of a matrix
  - Common case: i,j are not on the edge
- Nothing to test if i,j are < 1 or > sizeof(E)

```
@Test
void getElem() {
    Matrix m = new Matrix(new Scanner(simpleMatrix));
    assertEquals( expected: 2, m.getElem( i: 1, j: 2),
                 message: "getElem() did not return correct value");
}
```

Specification-based Testing

- Identify all common cases in the specification
  - Need one test-case per case
- Identify all boundary cases in the specification
  - Need one test case per boundary
- If a specification has exceptional cases include those as well

Example: How many tests do we need?
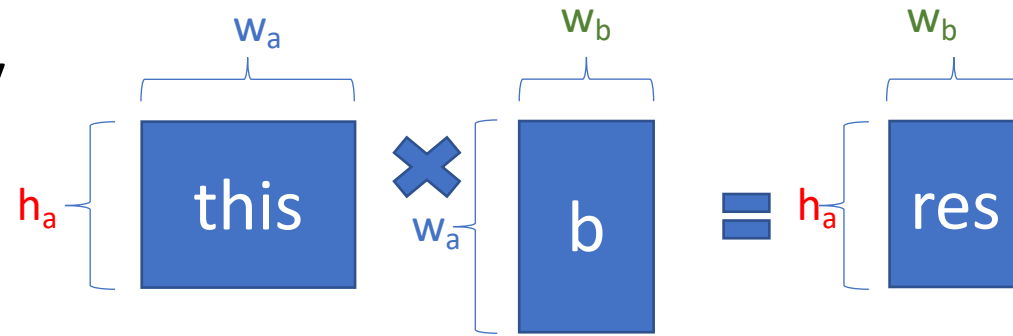

```
/**
 * Multiply the current matrix with matrix b and store the results in
 * res.  If matrices are not allocated or correctly sized, return null.
 * Parameters:  b:    matrix to be multiplied with the current matrix
 *              res: destination matrix where the result is placed.
 * Return: returns res matrix for programming convenience.
 */
public Matrix multiplyWithMatrix(Matrix b, Matrix res) { … }
```

Test Cases:

1. Null matrices: returns `null`
   - `res == null, b != null`
   - `res != null, b == null`
   - `res == null, b == null`

2. Mismatched dimensions: returns `null`
   - `this.width != b.height`
   - `res.height != this.height`
   - `res.width = b.width`

Test Cases:

3. Boundary conditions: return is `this * b`
   - `this.height = 1`
   - `this.width = 1`
   - `b.width = 1`

4. Normal cases: return is `this * b`
   - 1 < this.height == this.width
   - 1 < this.height < this.width
   - 1 < this.width < this.height
   - b.width >= 1 (with the above)

## 15+ Test Cases!

Adding Tests to a JUnit Test Class

- By default, IntelliJ creates one test method for each method in the target class
- If we have multiple tests for a method, we have two options
  - Put all tests in a single test method
  - Create multiple test methods for the same method

- Idea: Multiple small test methods are better:
  - Easier to understand
  - Easier to determine which tests failed
- To create another test method
  - Process make a copy of an existing test method
  - Modify name
  - Modify test
- All tests will be run when the JUnit Test Class is run

Click
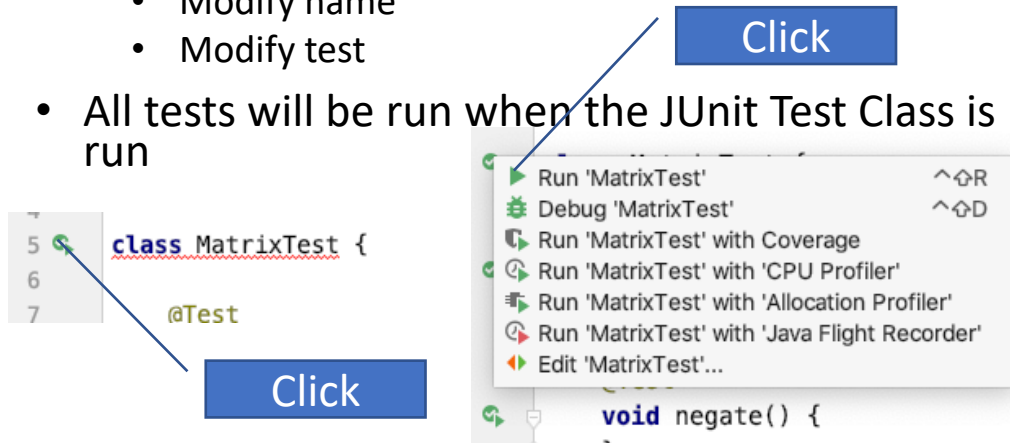
```
@Test
void multiplyWithMatrix_bIsNull() {
    ...
}


@Test
void multiplyWithMatrix_heightMismatch() {
    ...
}


@Test
void multiplyWithMatrix_bIsWidth1() {
    ...
}
```

```
5    class MatrixTest {
6
7        @Test
```

Run 'MatrixTest'	^⇧R
Debug 'MatrixTest'	^⇧D
Run 'MatrixTest' with Coverage	
Run 'MatrixTest' with 'CPU Profiler'	
Run 'MatrixTest' with 'Allocation Profiler'	
Run 'MatrixTest' with 'Java Flight Recorder'	
Edit 'MatrixTest'...	

```
    void negate() {
```

Click

Unit Testing

- **JUnit** got its name from unit testing
- Unit testing is the execution of a complete class, routine, or small program, which is tested in isolation from the more complete system
- **Scope**: Small units of code
  - Methods and functions
  - Simple classes
  - Typically written by a single developer
- **Goal**: Ensure the small simple building blocks work
  - Easiest to do because they focus on small chunks of code
- **Principle**: In most cases smaller pieces of code are easier to test
- Three types, depending on how much is known about the implementation
  Tests are based on:
  - **Black Box**: Strictly on specification or interface
  - **White Box**: Specification or interface, and the implementation
  - **Grey Box**: Specification or interface, and partial knowledge of the implementation
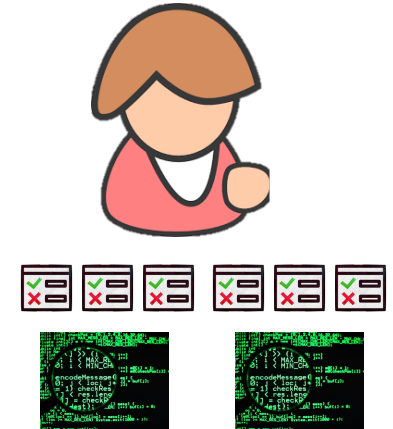
Why Start with Unit Testing

- First kind of testing done by the developer
- Serves as a basis for higher level testing
  - Unit tests are incorporated into higher level tests
- Easier to learn how to test small pieces of code
- Typically done individually
- Does not require a dedicated server

- Next Lecture will be a top-down view of testing!

Key TAKEAWAY Points

- A test is a combination of an execution framework, input, and output that runs a target piece of code to detect defects
- JUnit is a unit-testing framework that allows us to easily create, run, and debug tests
- Whitebox testing uses tests that are guided by knowledge of the implementation
- Blackbox testing uses tests that are guide by the specification for the code
- Unit testing focuses on small pieces of code that we want to test in isolation

Image References

Retrieved January 8, 2020

- http://pengetouristboard.co.uk/vote-best-takeaway-se20/
- https://i.pinimg.com/originals/b5/22/38/b52238fad11b0a3ecac36fa17604 1d98.jpg
- https://www.nbs-system.com/wp-content/uploads/2016/05/160503_Tests_boites-788x433.jpg
- https://www.pinclipart.com/picdir/middle/29-293393_challenges-peace-first-clip-art-work-teamwork-clip.png
- https://miro.medium.com/max/10000/1*6HUPtGBOSdERQkCOpL9QEg.png
- https://pngtree.com/freepng/black-repeat-icon_4841007.html

Retrieved September 26, 2020

- https://www.commitstrip.com/en/2013/10/11/mieux-et-moins-cher-que-les-tests-unitaires/?