

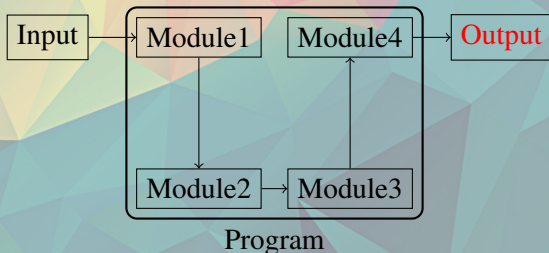
UNIT TESTING WITH JUNIT

Bernardo Cuteri

- Testing
 - An empiric method for verifying the correctness of a software
 - An automated process aimed at showing the behavior of a software on a give input
- Two categories: **black box** v.s. **white box**

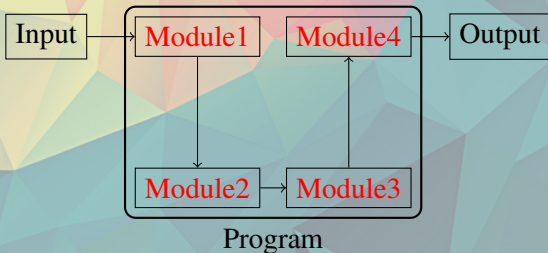
BLACK BOX TEST

- **Black box:** Given an input, tests whether the software outputs the expected result, ignoring how the software really works



WHITE BOX TEST

- **White box:** Single portions of source code are tested



UNIT TESTING

- **Unit Testing** is a white box testing methodology that tests the **unit of a source code**
- A **unit** is the smallest portion of code that may be tested
 - In procedural programming it may be a single program, or a function
 - In Java it may be a class, an interface, or even a method
- Unit Testing is the testing of a specific unit

WHY UNIT TESTING?

- A source code cannot be considered correct without being verified
- Divide-et-impera approach
 - Subdivide the system into unit
 - Each unit is debugged separately
 - Reduce probability of presenting bugs
 - Errors are not propagated among units
- Support regression testing
 - Verify that the application works as specified even after the changes/additions/modification were made to it
 - The original functionality continues to work as specified even after changes/additions/modification to the software application
 - The changes/additions/modification to the software application have not introduced any new bug

- Unit testing can be performed by a software
- **JUnit** is a Java Unit Testing framework
 - API for easily creating tests
 - Comprehensive assertion facilities (expected vs actual result)
 - Test runner for running tests
 - Test aggregation facilities

JUNIT: BASIC CONCEPTS

- **Test Case**: a method that verifies a specific functionality of a unit
- **Test Suite**: a collection of Unit Tests

JUNIT: CONVENTIONS

- The name of a test case method should indicate the expected behavior
 - good: `sqrtWorks`, `minWorks`, etc.
 - bad: `test1`, `myTest`, etc.
- Test classes usually end their name with “Test”
 - good: `MathTest`, `PersistenceTest`, etc.
 - bad: `MyClass`, `Test1`, etc.

JUNIT:TEST CASE

- JUnit is annotation driven
- There is no need to extend any special class
- Test cases are annotated with `@Test`
 - Test methods are void and take no parameters
 - Extra infos suggests specific behaviors
 - `@Test(timeout = 10)`: test succeeds if terminate within 10 seconds
 - `@Test(expected = IllegalArgumentException.class)`: test succeeds if `IllegalArgumentException` is thrown
 - `@Ignore("reason")` ignore a test

JUNIT: ANNOTATIONS

- **@Before:** mark a method for being invoked before each test case
- **@After:** mark a method for being invoked after each test case
- @BeforeClass: mark a method for being invoked at the beginning of the test
- @AfterClass: mark a method for being invoked at the end of the test
- @Before and @After are meant to prepare/release the test fixture for each test case
- @BeforeClass @AfterClass are static method and must appear at most once in each test

JUNIT: ANNOTATIONS

- **@Before**: mark a method for being invoked before each test case
- **@After**: mark a method for being invoked after each test case
- **@BeforeClass**: mark a method for being invoked at the beginning of the test
- **@AfterClass**: mark a method for being invoked at the end of the test
- **@Before** and **@After** are meant to prepare/release the test fixture for each test case
- **@BeforeClass** **@AfterClass** are static method and must appear at most once in each test

- `@Before`: mark a method for being invoked before each test case
- `@After`: mark a method for being invoked after each test case
- `@BeforeClass`: mark a method for being invoked at the beginning of the test
- `@AfterClass`: mark a method for being invoked at the end of the test
- `@Before` and `@After` are meant to prepare/release the test fixture for each test case
- `@BeforeClass` `@AfterClass` are **static** method and must appear at most **once** in each test

JUNIT: PARAMETRIZED TESTS

- Using `@RunWith(Parameterized.class)` and a parameter marked with `@Parameters`, we can execute a test over multiple values of the parameter

```
@RunWith(value=Parameterized.class)
public class FactorialTest {
    private long expected;
    private int value;
    @Parameters
    public static Collection<Object[]> data() {
        return
            Arrays.asList(new Object[][]{{1,0},{1,1},{2,2},{120,5}});
    }
    public FactorialTest(long expected, int value) { // constructor
        this.expected = expected;
        this.value = value;
    }
    @Test
    public void factorial() {
        assertEquals(expected, new Calculator().factorial(value));
    }
}
```

- Test Suites group tests into hierarchies

```
@RunWith(value=Suite.class)
@SuiteClasses(value={MyProgramTest.class, AnotherTest.class})
public class AllTests{
    ...
}
```

JUNIT: ASSERTS

- assertEquals (expected, actual)
 - Works with object, int, long, byte, string, . . . , etc.
 - Object: it invokes object.equals(object) to check for equality
- assertEquals (expected, actual, **epsilon**)
 - For float and double
- assertTrue / assertFalse (bool)
- assertNull / assertNotNull (object)
- assertEquals / assertEquals (object, object)
- assertEquals (object[], object[])