UNIT TESTING AND MOCKING

Peter Veres
Senior Software Engineer
peter_veres2@epam.com

May 3, 2018


UNIT TESTING AND MOCKING
INTRODUCTION

Agenda

1. Software testing in general
2. Unit Testing basics
3. Unit Testing in practice
4. Mocking
5. Code Coverage
6. Testable code
7. Test Driven Development

SOFTWARE TESTING TRENDS

SOFTWARE EVERYWHERE

- Internet of Things
- WEB
- Mobile, etc.

INCREASING DEMANDS

- Quality
- Reliability
- Security
- Performance

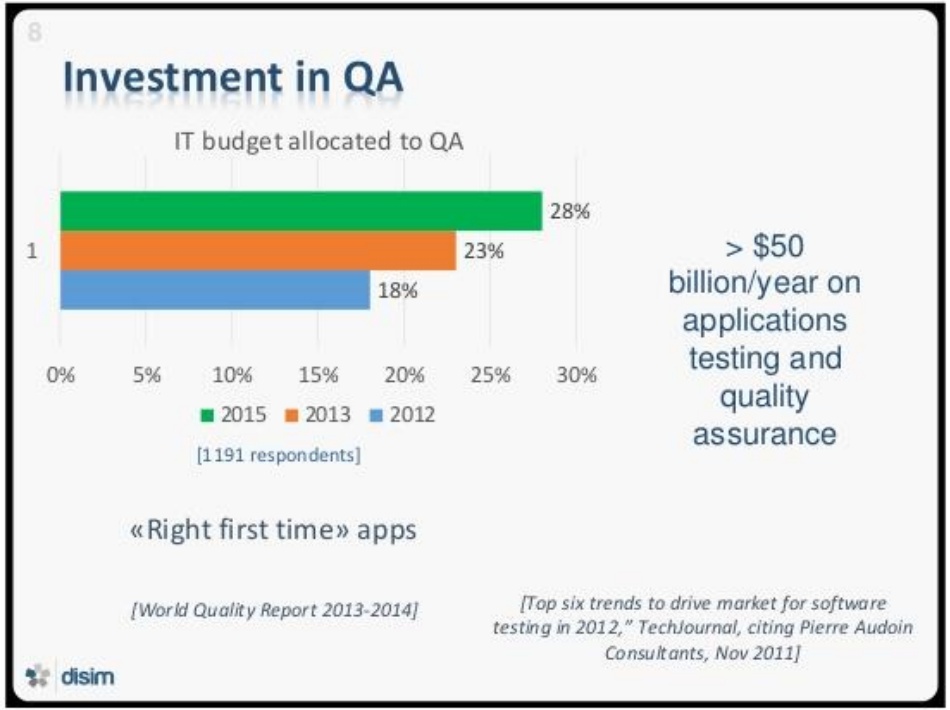
SOFTWARE TESTING TRENDS

MORE INVESTMENTS ON TESTING

- Increasing budget on software testing
- More QA jobs

COST OPTIMIZATION

- Automation
- Process improvements



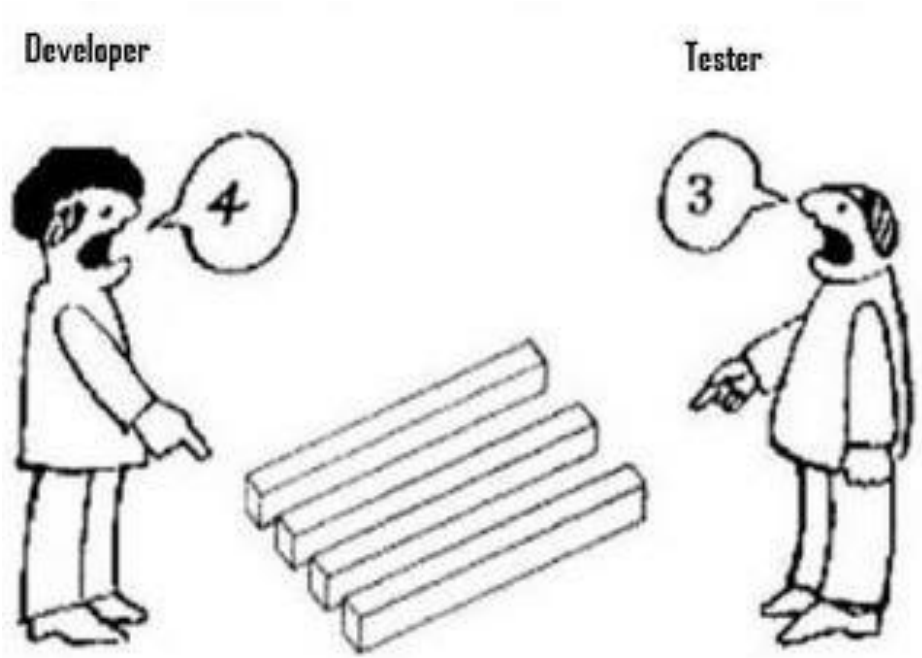
DEVELOPERS VS TESTERS

NARROWING THE GAP

- More developer knowledge on QA side
- More QA thinking on developer side

COLLABORATION

- Dev and QA on the same side, for the same goal
- Knowledge share
- Open mindedness



What is software testing?

- **Investigation** process in order to provide information about the **quality** of a software product,
- In an way, that is
    - **Objective and**
    - **Independent.**

Aims of software testing

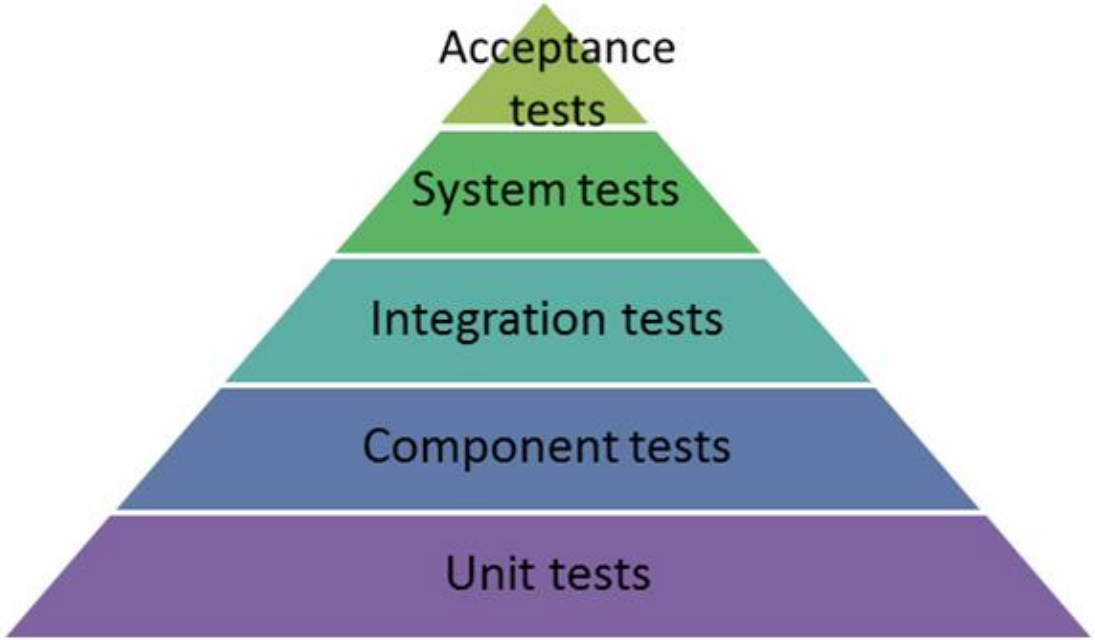
- **Validation** and/or **verification** to make sure if the product is
    - **Functionally correct**: meets the business requirements
    - **Technically correct**: meets the technical requirements
    - **Functioning**: able to run/serve
- Detect and localize software defects/failures

Software testing attributes

- Scope
- Functional or non-functional
- Static or dynamic
- Verification and/or validation

Testing pyramid

IT IS NOT ENOUGH TO HAVE ONLY UNIT TESTS



Non-functional testing

- Performance
- Stability
- Usability
- Security
- I18n and localization
- Destructive


UNIT TESTING AND MOCKING
UNIT TESTING

UNIT TESTING – WHY?

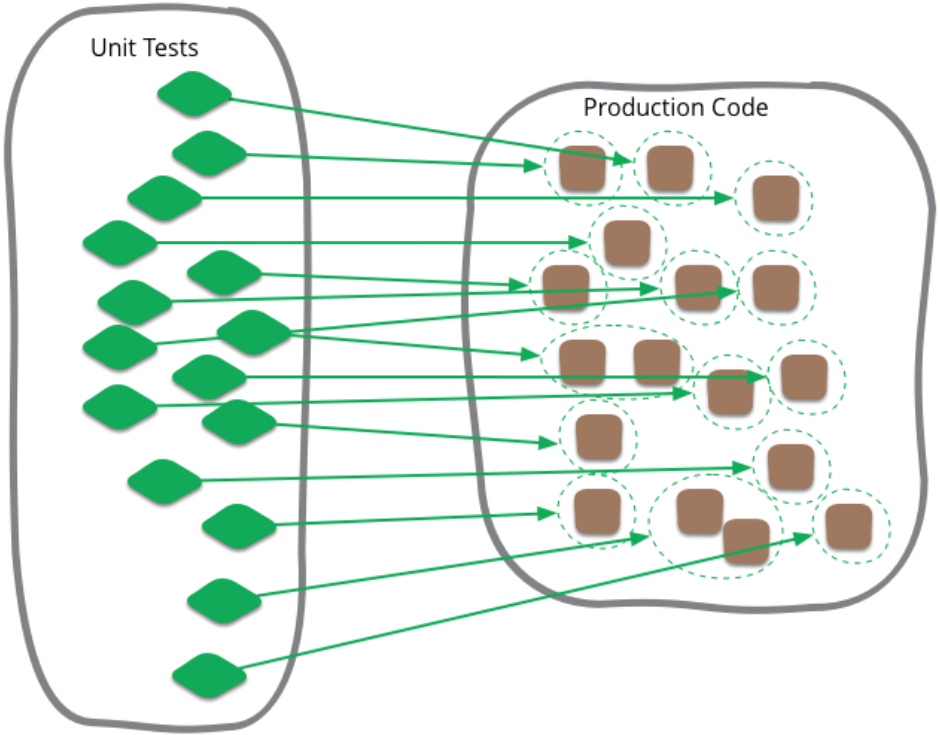
MAKES DEVELOPMENT EASIER

- Developers can become more confident
- Immediate feedback about code changes

LOWER MAINTENANCE COST

- Saves effort when one needs to identify the root cause of broken code
- Documents use cases at low level
- Points out bugs much earlier than they could cause bigger issues

UNIT TESTING – HOW?



What shall be called Unit Test?

Classic definition of Unit Test:

A unit test is a piece of code (usually a method) that invokes another piece of code and checks the correctness of some assumptions afterward.

If the assumptions turn out to be wrong, the unit test has failed.

A “unit” is a method or function.

What shall be called Unit Test?

Is that the definition of a good unit test?

What shall be called Unit Test?

What are the characteristics of a good Unit Test?

UNIT TESTING – PRINCIPLES

Fast

Independent

Repeatable

Self-validating

Timely

Questions to always be answered

- Can I run and get results of a unit test I wrote two weeks/months/years ago?
- Can any member of my team run and get the results from unit tests I wrote two months ago?
- Can it take me no more than a few minutes to run all the unit tests I've written so far?
- Can I run all the unit tests I've written at the push of a button?
- Can I write a basic unit test in no more than a few minutes?

What is a good Unit Test?

A useful definition:

A unit test is an **automated** piece of code that invokes a different method and then checks some **assumptions** about the **logical behavior** of that method or class under test.

A unit test is written using a unit testing framework. It can be **written easily** and **runs quickly**. It can be **executed**, **repeatedly**, **by anyone** on the development team.

How does Unit testing fit into the software testing definition?

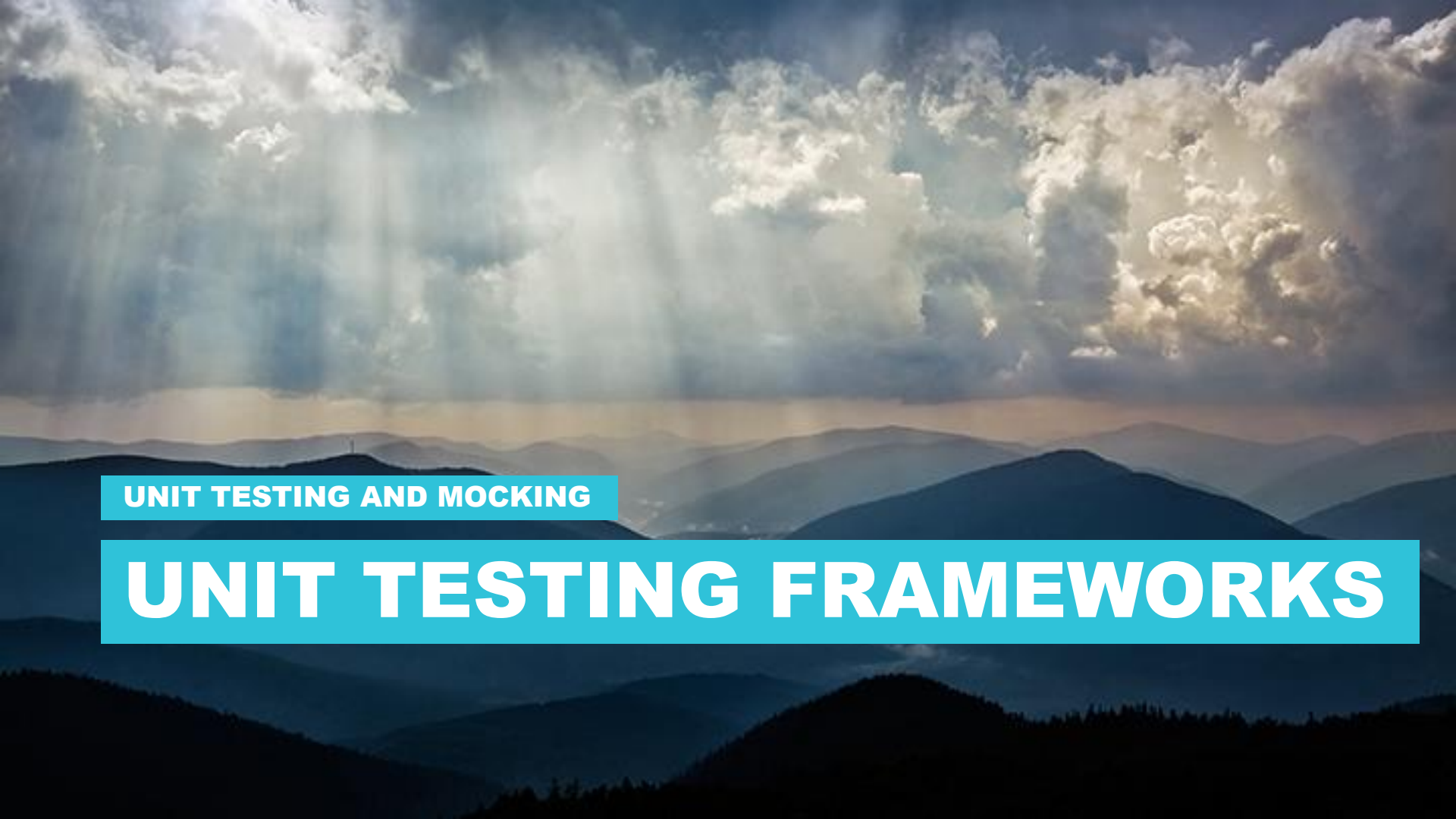
- Is it an investigation process?
- Does it provide information about quality?
- Is it objective?
- Is it independent?

What are the attributes of Unit Testing?

- What is the scope of a Unit Test?
- Is it functional or non-functional?
- Is it static or dynamic?
- Is it a verification or a validation process or both?

Place Unit Testing into development process

- When?
- Where?
- How?


UNIT TESTING AND MOCKING
UNIT TESTING FRAMEWORKS

Unit testing frameworks

Why Unit Testing frameworks are essential in Unit Testing?

Unit testing frameworks

- ✓ Can I run and get results of a unit test I wrote two weeks/months/years ago?
- ✓ Can any member of my team run and get the results from unit tests I wrote two months ago?
- ✓ Can it take me no more than a few minutes to run all the unit tests I've written so far?
- ✓ Can I run all the unit tests I've written at the push of a button?
- ✓ Can I write a basic unit test in no more than a few minutes?

Expectations to fulfill by Unit Testing frameworks

- Help in writing test easily and in a structured manner
- Provide a way to execute one or all of the unit tests
- Present the result of the test runs somehow

JUnit demo

- git@gitbud.epam.com:peter_veres2/unit-testing.git
- Create as new projects in Eclipse

- Demo: JUnit

DEMO

How does a Unit Testing framework fulfill those expectations

- Framework provides a Test Runner - console or GUI tool that
  - Identifies tests in your code
  - Runs them automatically
  - Gives you status while running
  - Can be automated by command line

How does a Unit Testing framework fulfill those expectations

- The test runners will usually let you know
  - How many tests ran
  - How many did not run
  - How many failed
  - Which tests failed
  - Why they failed
  - Assert message you wrote
  - The code location that failed
  - Possibly a stack trace


UNIT TESTING AND MOCKING
JUNIT 5

What is JUnit 5?

- JUnit 5 contains a number of exciting innovations, with the goal to support new features in Java 8 and above
- **It is important to note, that this version requires Java 8 to work**
- JUnit 5 is composed of several different modules from three different sub-projects
- JUnit 5 = JUnit Plarform + JUnit Jupiter + JUnit Vintage

Basic annotations

- `@BeforeAll`
- `@BeforeEach`
- `@Test`
- `@AfterEach`
- `@AfterAll`
- `@Disabled`
- `@DisplayName`
- `@RepeatedTest`
- `@ParameterizedTest`

Assertions - Examples

```
private Person person;
@BeforeEach
void setup() { person = new Person("John", "Doe"); }
@Test
void standardAssertions() {
    assertEquals(2, 2);
    assertEquals(4, 4, "Optional assertion message.");
    assertTrue('a' < 'b', () -> "Assertion messages can be lazily evaluated.");
}
@Test
void groupedAssertions() {
    assertAll("personAssert",
        () -> assertEquals("John", person.getFirstName()),
        () -> assertEquals("Doe", person.getLastName()));
}
```

Assertions - Examples

```
@Test
void iteterableAssertions() {
    List<Person> people = Arrays.asList(new Person("John", "Doe"), new
        Person("Jane", "Doe"));
    assertNotNull(people);
    assertIterableEquals(people, findPeople());
}

@Test
void linesMatchAssertions() {
    List<String> expectedLines = Collections.singletonList("(.*)@(.*)");
    List<String> emails = Arrays.asList("john@gmail.com");
    assertLinesMatch(expectedLines, emails);
}
```

Exception handling

- @Test annotation does not accept arguments in Junit 5 to check exceptions

```
@Test
void testException() {
    Throwable exception =
    assertThrows(IllegalArgumentException.class, () -> {
        throw new IllegalArgumentException("message");
    });
    assertEquals("message", exception.getMessage());
}
```

Assumptions

JUnit 5

- An assumption defines the conditions which have to be met so that a test will be run.
- A failing assumption does not mean a test is failing, but simply that the test won't provide any relevant information, so it doesn't need to run.
  - assumeTrue()
  - assumeFalse()
  - assumingThat()
- If an assumption fails, a TestAbortedException is thrown and the test is simply skipped.

Assumptions - Examples

JUnit 5

```
@Test
void trueAssumption() {
    assumeTrue(5 > 10);
    assertEquals(5 + 2, 7);
}                              //test is skipped, because assumption failed

@Test
void assumptionThat() {
    String someString = "Just a string";
    assumingThat(
        someString.equals("Just a string"),
        () -> assertEquals(2 + 2, 4)
    );
}
```

Dependency Injection

JUnit 5

- Currently, there are only 3 built-in resolvers for parameters of type TestInfo, RepetitionInfo and TestReporter

```
@Test
@DisplayName("Test getUsers")
void testCaseB(TestInfo info) {
   assertEquals(2, userDao.findAll().size());
   assertEquals("Test getUsers", info.getDisplayName());
   assertEquals(UsersTest.class, info.getTestClass().get());

   logger.info("Running test method: " +
   info.getTestMethod().get().getName());
}
```

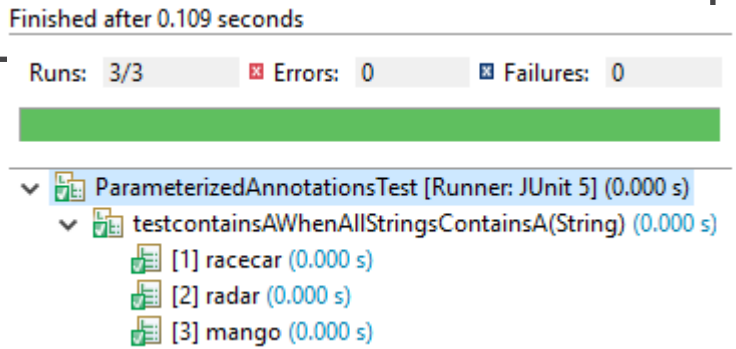
Parameterized Tests Annotations

- @ValueSource
- @EnumSource
- @MethodSource
- @CsvSource and @CsvFileSource
- @ArgumentsSource

Parameterized Tests - Example

- Parameterized tests allow running the same test multiple times, but with different arguments.

```
@ParameterizedTest
@ValueSource(strings = { "racecar", "radar", "mango" })
void testcontainsAWhenAllStringsContainsA(String candidate) {
   assertTrue(containsA(candidate));
}
```



Annotations in interfaces

- JUnit 5 also allows several annotations to be added to test interfaces:
  - @Test, @RepeatedTest, @ParameterizedTest, @TestFactory, @BeforeEach and @AfterEach can be added to default methods in interfaces
  - @BeforeAll and @AfterAll can be added to static methods in interfaces
  - @ExtendsWith and @Tag can be declared on interfaces

Annotations in interfaces - Example

JUnit 5

- We can declare tests in an interface:

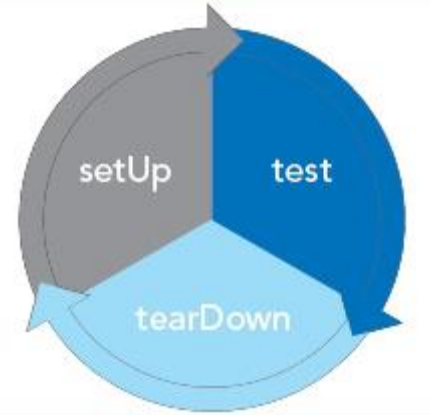
```
public interface DatabaseConnectionTest {

    @Test
    default void testDatabaseConnection() {
        Connection con = ConnectionUtil.getConnection();
        assertNotNull(con);
    }
}
```

- Which are going to be executed within the Implemetor:

```
public class UsersTest implements DatabaseConnectionTest {...}
```

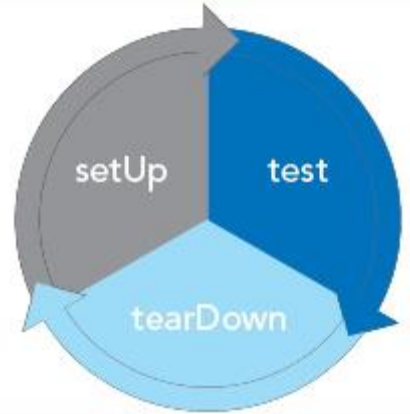
Basic structure

- Setup methods (create fixture, set up initial state)
- Test methods
- Tear down methods



Lifecycle

- Load Unit Test class
- Invoke BeforeAll methods
- Instantiate the Unit Test class
- Do tests (while non-executed test method exists):
  - Invoke Before methods
  - Invoke Test method
  - Invoke After method
- Invoke AfterAll methods



How does a Unit Testing framework fulfill those expectations

- Supplies the developer with a class library that holds
  - Attributes to place in your code to note your tests to run
  - Base classes or interfaces to inherit
  - Assert classes that help in verifying your code

DEMO

Exercise: First unit test

Please write a Unit Test for an existing class

<UltimateKnowledge.java>

EXERCISE

How did You write your unit test?

- How did You name your test class?
- How did You name your test methods?
- How did You name the System Under Test?
- How did You structure your test methods?
- What did You have in the Before and After methods?
- How many test methods did You write?

Basic Rules to follow when writing Unit Tests

- Name the test class like `[SystemUnderTest]Test`
- Name the test methods like `test[TestedMethod]Should[DoSomething]When[Condition]()`
- Name the tested object's variable conventional, like `underTest`

Basic Rules to follow when writing Unit Tests

- Structure test methods like:
  - GIVEN: initialize a sate the tested method should run in
  - WHEN: call the tested method
  - THEN: verify the new state
- Never initialize a state in the Before method that are not needed for ALL your test methods!

Basic Rules to follow when writing Unit Tests

- How many test methods should you write?
  - At least 1 test method for all method that contains any kind of logic
  - However the good approach is to have as many test method as the cyclomatic complexity of the tested method

Basic Rules to follow when writing Unit Tests

- Keep it simple
- Keep it easily understandable
- Keep it conventional
- The unit test should also describe the logic the production class implements: helps in understanding the code

Naming conventions

- JUnit3 naming conventions are usually followed:
  - `@Test` methods named like `test...()`
  - `@Before` methods called `setUp()`
  - `@After` methods called `tearDown()`

What should be tested?

- Business Requirements:
  - Do some calculations
  - Deal with persistent state
  - Manage history (business logging)
  - ...
- Technical requirements
  - Log activities (system logging)
  - Alert system failures
  - ...

What should be tested?

- Return value
- Side effects:
  - Side effects on value holders (DTOs)
  - Behavior:
    - All interaction with dependencies
- Exceptional cases

Exercise: Second unit test

Please write a Unit Test for existing classes

Calculator.java, Logger.java, Sum.java

EXERCISE

Questions about your test

What problems did you face with?

What have you tested?

Keeping real dependencies of the SUT makes our Unit Test

- To be not independent
  - which means it is no longer a Unit Test
  - but Integration test

What problems caused by integration tests?

UNIT TESTING – PRINCIPLES

UNIT TEST RULES BY MICHEAL FEATHERS

A test is ***not*** a unit test if

1. It talks to the database
2. It communicates across the network
3. It touches the file system
4. It can't run correctly at the same time as any of your other unit tests
5. You have to do special things to your environment (such as editing config files) to run it

UNIT TEST VS INTEGRATION TEST

UNIT TESTS ARE NOT INTEGRATION TESTS

UNIT TEST

- Only one unit in scope
- Test runs quickly
- Specific errors
- Only the unit must be initialized
- Change in a unit affects only one test

INTEGRATION TEST

- Multiple participants in an interaction
- Test can run longer
- Hard to localize cause of failure
- Additional configuration and setup
- Changes in dependencies affect more than one test

UNIT TESTING AND MOCKING

MOCKING AND MOCKING FRAMEWORKS

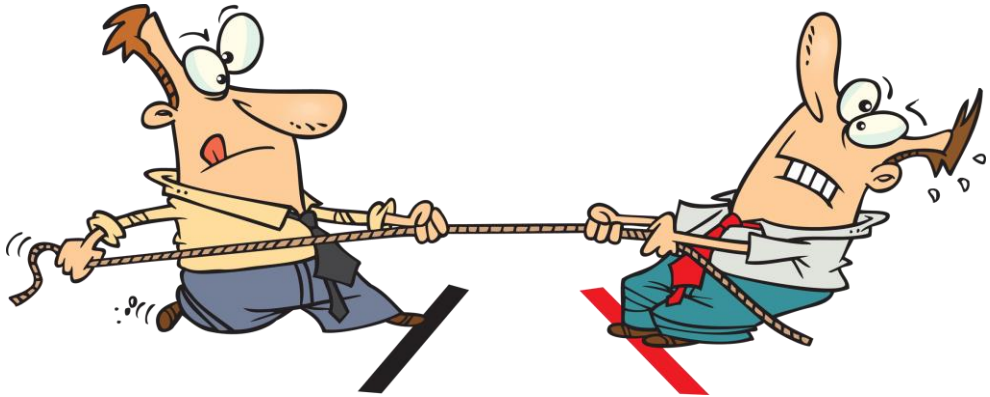
How to deal with dependencies?

- Stubs/Mocks:
  - Replacement for an existing dependency in the system
  - The Unit Test can have control over
  - The SUT interacts with stubs instead of real dependencies.

UNIT TESTING – VERIFICATION STEP

TWO GROUPS OF UNIT TEST WRITERS

CLASSICISTS **MOCKISTS**



UNIT TESTING – VERIFICATION STEP

TWO GROUPS OF UNIT TEST WRITERS

CLASSICISTS

- After the exercise phase they check the collected results and the state of the tested object
- Use assertions

MOCKISTS

- Before the exercise phase they expects some specific behavior and after the exercise phase they verify if it happened
- Use mock objects (see in later section)

Stubs

Problems arising when using stubs:

- Stubs are silent contributors in testing
- We cannot record interactions between SUT and Stubs

Mocks

- Also replaces a real object
- Allows verifying the calls (interactions)
- Implements the same interface as the replaced object
- Can be controlled, created and injected into the system under test by the unit test.

Mocking frameworks

- A set of programmable APIs
- Allow creating Mock and Stub Objects in an easy way
- Prevents creating Mocks and Stubs manually

EasyMock demo

EASYMOCK

- EasyMock demo: ClientTest.java, ClientInOrderTest.java

DEMO

EasyMock

EASYMOCK

- A Mocking framework for Java
- Enables creating dynamic mocks
- Only a .jar file is needed to be placed in the classpath
- *Some disadvantage:*
  - *Before v3 classmocking is available only with class-extension.jar*
  - *Now Cglib and Objenesis are needed but no class-extension.jar*

Simple flow of using Mock Objects

EASYMOCK

1) Create a Mock Object for the specified interface or class
2) Record the expected behavior (recoding status)
3) Switch the Mock Object to replay state
4) Call the tested method
5) Verify the state, make assertions

Verifying behavior

EASYMOCK

- When some behavior is expected, but there are no interactions between the SUT and the mock, the test still passes when no call performed to `EasyMock.verify(mocks);`

Recording state

EASYMOCK

- Expecting void method calls
- Specifying return values
- Throwing exceptions
- Changing method behavior
- Using Answer callbacks
- Specifying number of calls
- Calls “in order” (with one and two mocks)

Recording state

EASYMOCK

- Enable/disable order checking
- Mock behavior when it is:

	Default order checking	Calls to unexpected methods
Default mock	disabled	cause test fail
Strict mock	enabled	cause test fail
Nice mock	disabled	allowed

- `andStubReturn(...)` calls

Flexible Expectations with argument matchers

EASYMOCK

- Defines different behavior for the mocked method according to the arguments provided by the caller
- Limitations: when at least argument matcher provided then all the arguments have to be a matcher instance.

EasyMock is not able to mock…

EASYMOCK

- Final classes
- Final methods
- Static methods

Most of the mocking frameworks do not provide options to mock the constructs above.

Differences between mocks and stubs

Mock

- Can fail test
- Records interactions
- Unit Test makes assertions on mock

Stub

- Cannot fail test
- Does not record interactions
- Unit Test makes assertions on the SUT instead

Homework: Third unit test

Please write a Unit Test for existing classes

<com.epam.alltogether.*>

EXERCISE

Mockito

- A Mocking framework for Java (just like EasyMock)
- Everything is in one .jar file
- Easy Mock creation (Annotation support)

Simple flow of using Mock Objects

- Annotate the member properties to be mocked
- Call `MockitoAnnotations.initMocks(this)`
- Or use `@MockitoJUnitRunner`
- Mock behavior
- Call tested method
- Verify behavior, make assertions

Mockito features

- Defining expectations:
  - Void calls, return values, throwing exceptions, argument matchers etc.
- Verifying behavior:
  - Verifying method calls, specifying number of calls etc.

Mockito demo

- Mockito demo: ClientMockitoTest.java, ClientMockitoInOrderTest.java

DEMO


UNIT TESTING AND MOCKING
OTHER FEATURES

Maven integration

- Surefire Plugin
  - used during the test phase of the build lifecycle to execute the unit tests

- Failsafe Plugin
  - designed to run integration tests
  - usage
    - `mvn test`
    - `mvn integration-test`
    - `mvn test -Dtest.categories=SlowTests`
    - `mvn test -Dtest=SlowTestSuite`

Tagging and filtering

JUnit 5

- @Tag annotation can be used on class or method level tests
- These allow developers to group and filter tests

```
@Tag("math")
public class TaggedTest {

    @Test
    @Tag("arithmetic")
    void testCaseA() {
        assertTrue(1 == 1);
    }
}
```

Hamcrest matchers

- Framework for writing matcher objects allowing 'match' rules to be defined declaratively
- In unit tests make your (JUnit) tests as readable as possible

```
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.*;

public class Test extends TestCase {
    public void testEquals() {
        Biscuit theBiscuit = new Biscuit("Ginger");
        Biscuit myBiscuit = new Biscuit("Ginger");
        assertThat(theBiscuit, equalTo(myBiscuit));
    }

    public void testSquareRootOfMinusOneIsNotANumber() {
        asserThat(Math.sqrt(-1), is(notANumber()));
    }
}
```

Hamcrest matchers

- Lots of common matchers
  - Core – `anything`, `is`
  - Logical – `allOf`, `anyOf`, `not`
  - Object – `instanceOf`, `equalTo`
  - Beans - `hasProperty`
  - Collections – `hasEntry`, `hasItemInArray`
  - Number – `greaterThan`, `greaterThanOrEqualTo`
  - Text – `equalToIgnoringCase`, `containsString`, …
- Writing custom matchers is possible

- More information at https://github.com/hamcrest/JavaHamcrest

TestNG

- Another Unit testing framework
- Provides more opportunity to influence test runs than JUnit3-4

JUnit 5 vs. TestNG

	Annotation Support	Exception Test	Ignore Test	Timeout Test	Suite Test	Group Test	Parameterized Test	Dependency Test
TestNG	✓	✓	✓	✓	✓	✓	✓	✓
JUnit 5	✓	✓	✓	✓	✓	✓	✓	✗

Putting all together

- JUnit/TestNG
- EasyMock/Mockito
- Emma
- Eclipse/IDEA
- Maven/Gradle


UNIT TESTING AND MOCKING
CODE COVERAGE

CODE COVERAGE – SOFTWARE METRIC

CODE COVERAGE IS A GROUP OF SOFTWARE METRICS

- It measures the proportion of source code that is executed by the test suites

```
public boolean addAll(int index, Collection c) {
    if(c.isEmpty()) {
        return false;
    } else if( size == index ||  size == 0) {
        return addAll(c);
    } else {
        Listable succ = getListableAt(index);
        Listable pred = (null == succ) ? null : succ.prev();
        Iterator it = c.iterator();
        while(it.hasNext()) {
            pred = insertListable(pred,succ,it.next());
        }
        return true;
    }
}
```

Coverage Units

- Class
- Method
- Line
- Basic Block (base unit)

Emma can detect if a coverage unit was covered or not covered or partially covered (excluding block)

Examples

- IF statement
- Iteration
- Exception
- Hidden logic

CODE COVERAGE – PROS & CONS

- Automated
- Can be aggregated
- Easy to use (just some numbers), understable by business

CODE COVERAGE CAN PROVIDE MORE THAN YOU EXPECT

- Can identify dead/unused functionality
- Is a great help when you need to decide what to be covered in a legacy project

BUT IT CAN BE EASILY MISLEADING

- It is very easy to write test that covers 100% of code but does not operate as a test harness

CODE COVERAGE – COVERAGE GOAL?

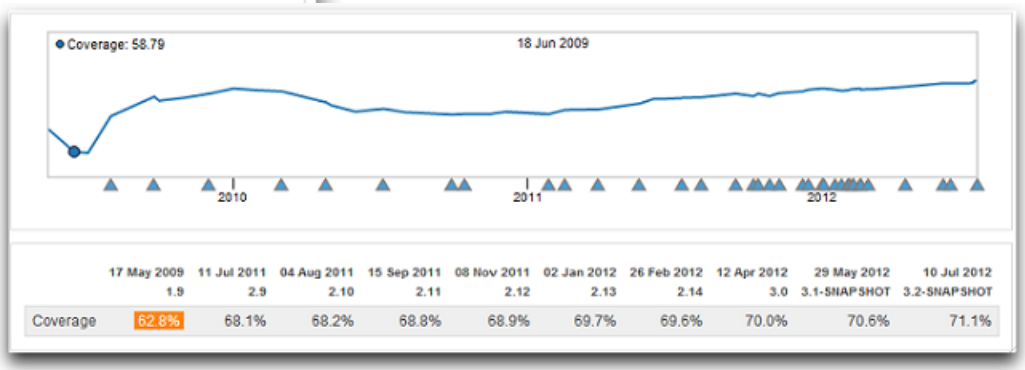
WHAT IS THE BEST OPTION? 80? 95? 100%?

- 80% could be good compromise but there is not any best limit
- Continuous integration can help to avoid declines in code coverage

Code coverage
70.6% (+0.6)
71.2% line coverage **(+0.5)**
68.9% branch coverage **(+0.7)**

On new code
78.8%
1,481 lines to cover
80.1% line coverage
74.5% branch coverage

Unit test success
100.0% (+0.0)
0 failures **(+0)**
0 errors **(+0)**
2,778 tests **(+172)**
0 skipped **(-2)**
12:34 min **(+1:12 min)**



	17 May 2009 1.9	11 Jul 2011 2.9	04 Aug 2011 2.10	15 Sep 2011 2.11	08 Nov 2011 2.12	02 Jan 2012 2.13	26 Feb 2012 2.14	12 Apr 2012 3.0	29 May 2012 3.1-SNAPSHOT	10 Jul 2012 3.2-SNAPSHOT
Coverage	62.8%	68.1%	68.2%	68.8%	68.9%	69.7%	69.6%	70.0%	70.6%	71.1%

Coverage tools

- Emma
- Clover
- Cobertura
- ...

Emma

- Coverage tool for Java
- Only one .jar file to use
- Support different output formats, like: txt, xml, html
- Can be used from command line, Maven, Eclipse (plugin)


UNIT TESTING AND MOCKING
TESTABLE CODE

Testable code

- What makes code easily testable?
  - Clean dependency hierarchy
  - Clean methods (simple, not too complex ones)
  - Keeping Test Unfriendly Features on low degree

Test Unfriendly features

- Access to database, filesystem, network
- Side effecting APIs (like GUIs)
- Lengthy computations
- Static variable usage

Test Unfriendly constructs

- Final methods, classes
- Static, private methods
- Static initialization expression or blocks
- Constructors
- Object initialization blocks
- New expressions

A best practice

- Never hide a test unfriendly **feature** within a test unfriendly **construct**.

Unit Testing's effects on production code

- Designing to testability improves code and design quality:
  - Cleans dependencies
  - Decreases complexity
  - Highlights responsibilities

How to improve these effects?

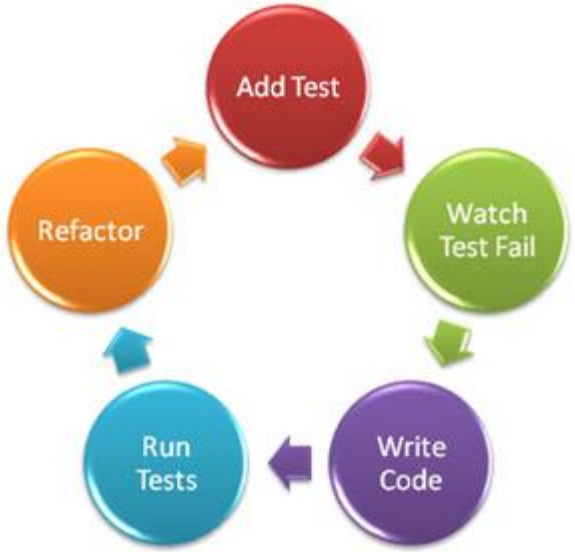

UNIT TESTING AND MOCKING
TDD

Test Driven Development

Test first, then implement production code

TDD one step

1. Create production class/method
2. Create unit test class/method
3. Test a requirement
4. Run test, see if it fails
5. Satisfy requirement
6. Runt test, see if it passes
7. Refactor




epam
THANKS FOR YOUR
KIND ATTENTION