CONTENTS INCLUDE:

- Introduction to Unit Testing
- Configuring Mockito in a Project
- Creating Mock
- Stubbing Method's Returned Value
- Argument Matching
- And more...

Mockito

A Simple, Intuitive Mocking Framework

By Marcin Zajączkowski

INTRODUCTION TO UNIT TESTING

A *unit test* is a test related to a single responsibility of a single class, often referred to as the *System Under Test* (SUT). The purpose of unit tests is to verify that the code in an SUT works. A tested object usually talks to other objects known as collaborators. These collaborators need to be created so the tested object can be assigned to them in the test. To make unit testing simpler and allow control of all aspects of the execution context, it is useful to replace the real cooperating objects with their fake replacements called test doubles. They look like the originals, but do not have any dependencies to other objects. *Test doubles* can also be easily programmed with specific expectations, such as recording any interactions they've had.

To make it clearer, try to imagine code for a typical enterprise system. Now here's a service with some logic that needs two classes to fulfill its responsibility. Both classes then require a few other classes. One of these other classes could be a DAO, which needs access to a database, while yet another requires a message queue. It would be quite an effort to create that hierarchy and provide required resources. There could also be problems while running that kind of test, e.g., long startup times or the inability to test multiple developer stations simultaneously. Using Mocks, though, the same test could be much cleaner and faster.

Test doubles can be divided into a few groups:

- Dummy - an empty object passed in an invocation (usually only to satisfy a compiler when a method ar- gument is required)
- Fake - an object having a functional implementation, but usually in a simplified form, just to satisfy the test (e.g., an in-memory database)
- Stub - an object with hardcoded behavior suitable for a given test (or a group of tests)
- Mock - an object with the ability to a) have a programmed expected behavior, and b) verify the interactions occurring in its lifetime (this object is usually created with the help of mocking framework)
- Spy - a mock created as a proxy to an existing real object; some methods can be stubbed, while the un- stubbed ones are for- warded to the covered object

Mockito is a mocking framework helpful in creating mocks and spies in a simple and intuitive way, while at the same time providing great control of the whole process.

CONFIGURING MOCKITO IN A PROJECT

Mockito artifacts are available in the Maven Central Repository (MCR). The easiest way to make MCR available in your project is to put the following configuration in your dependency manager:

Maven:

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <version>1.9.0</version>
  <scope>test</scope>
</dependency>
```

Gradle:

```
testCompile "org.mockito:mockito-core:1.9.0"
```

Ivy:

```
<dependency org="org.mockito" name="mockito-core" rev="1.9.0" conf="test->default"/>
```

It will add JAR with Mockito classes as well as all required dependencies. Change 1.9.0 with the latest released version of Mockito.

This Refcard is based on the latest stable version 1.9.0. Some things are about to change in the further Mockito versions.

CREATING MOCK

A mock can be created with the help of a static method mock():

```
Flower flowerMock = Mockito.mock(Flower.class);
```

But there's another option: use of @Mock annotation:

```
@Mock
private Flower flowerMock;
```

Warning: If you want to use @Mock or any other Mockito annotations, it is required to call `MockitoAnnotations.initMocks(testClass)` or use `MockitoJUnit4Runner` as a JUnit runner (see the annotation section below for more information).

Hot Tip You can use Mockito to create mocks of a regular (not final) class not only of an interface.

STUBBING METHOD'S RETURNED VALUE

One of the basic functions of mocking frameworks is an ability to return a given value when a specific method is called. It can be done using Mockito.when() in conjunction with thenReturn () . This process of defining how a given mock method should behave is called stubbing.

Warning: Note that the examples in this Refcardwere created to demonstrate behaviors of Mockito in a specific context. Of course, when writing the test for your codebase, there is no need to ensure that mocks are stubbed correctly.

```
package info.solidsoft.blog.refcard.mockito;

import org.testng.annotations.Test;

import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;
import static org.testng.Assert.assertEquals;

public class SimpleStubbingTest {
        public static final int TEST_NUMBER_OF_LEAFS = 5;

        @Test
        public void shouldReturnGivenValue() {
                Flower flowerMock = mock(Flower.class);
                when(flowerMock.getNumberOfLeafs()).thenReturn(TEST_NUMBER_OF_LEAFS);

                int numberOfLeafs = flowerMock.getNumberOfLeafs();

                assertEquals(numberOfLeafs, TEST_NUMBER_OF_LEAFS);
        }
}
```

Hot Tip Mockito makes heavy use of static methods. It is good to use static imports to make code shorter and more readable. IDE can be used to automatize adding static imports.

Mockito provides a family of functions for requesting specific behaviors.

Method	Description
thenReturn(T valueToBeReturned)	returns given value
thenThrow(Throwable toBeThrown) thenThrow(Class<? extends Throwable> toBeThrown)	throws given exception
then(Answer answer) thenAnswer(Answer answer)	uses user created code to answer
thenCallRealMethod()	calls real method when working with partial mock/spy

Hot Tip Non void methods return by default an "empty" value appropriate for its type (e.g.: null, 0, false, empty collection).

Following an arrange-act-assert pattern (similar to given-when-then from Behavior Driven Development) a test should be split into three parts (blocks), each with a specified responsibility.

Section name	Responsibility
arrange (given)	SUT and mocks initialization and configuration
act (when)	An operation which is a subject to testing; preferably only one operation on an SUT
assert (then)	The assertion and verification phase

This way, what is being tested, is clearly separated from the setup and verification parts. To integrate cleanly with Behavior Driven Development semantics Mockito contains BDDMockito class which introduces an alias given() which can be used instead of when() method while stubbing. Here's the previous example, but now using the BDD semantics:

```
import static org.mockito.BDDMockito.given;
import static org.mockito.Mockito.mock;
import static org.testng.Assert.assertEquals;

@Test
public void shouldReturnGivenValueUsingBDDSemantics() {
  //given
  Flower flowerMock = mock(Flower.class);
  given(flowerMock.getNumberOfLeafs()).willReturn(TEST_NUMBER_OF_LEAFS);

  //when
  int numberOfLeafs = flowerMock.getNumberOfLeafs();

  //then
  assertEquals(numberOfLeafs, TEST_NUMBER_OF_LEAFS);
}
```

given-when-then comments make intentions of tests clearer.

ARGUMENT MATCHING

Mockito, by default, compares arguments using equals () methods. Sometimes it's convenient to know exactly what parameter the method will be called with.

```
@Test
public void shouldMatchSimpleArgument() {
                WateringScheduler schedulerMock = mock(WateringScheduler.class);
                given(schedulerMock.getNumberOfPlantsScheduledOnDate(WANTED_DATE)).willReturn(VALUE_
FOR_WANTED_ARGUMENT);

                int numberForWantedArgument = schedulerMock.getNumberOfPlantsScheduledOnDate(WANT
ED_DATE);
                int numberForAnyOtherArgument = schedulerMock.getNumberOfPlantsScheduledOnDate(A
NY_OTHER_DATE);

                assertEquals(numberForWantedArgument, VALUE_FOR_WANTED_ARGUMENT);
                assertEquals(numberForAnyOtherArgument, 0); //default value for int
}
```

Very often we needed to define a wider matching range. Mockito provides a set of build-in matchers defined in Matchers and AdditionalMatchers classes (see corresponding table).

Hot Tip If an argument matcher is used for at least one argument, all arguments must be provided by matchers.

```
given(plantSearcherMock.smellyMethod(anyInt(), contains("asparag"), eq("red"))).willReturn(true);
//given(plantSearcherMock.smellyMethod(anyInt(), contains("asparag"), "red")).willReturn(true);
//incorrect - would throw an exception
```

Luckily, in the last case, Mockito will protest by throwing a meaningful exception:

```
org.mockito.exceptions.misusing.InvalidUseOfMatchersException:
Invalid use of argument matchers!
3 matchers expected, 2 recorded.
This exception may occur if matchers are combined with raw values:
  //incorrect:
  someMethod(anyObject(), "raw String");
When using matchers, all arguments have to be provided by matchers.
For example:
  //correct:
  someMethod(anyObject(), eq("String by matcher"));

For more info see javadoc for Matchers class.
```

Hot Tip The methods from the any() family don't do any type checks. Various variants were created to avoid casting. To perform type checks, method isA(Class) should be used.

It is also possible to create a custom matcher by extending the ArgumentMatcher class and using it together with argThat ()

```
given(schedulerMock.getNumberOfPlantsScheduledOnDate(
                argThat(haveHourFieldEqualTo(7)))).willReturn(1);

//with the util method to create a matcher
private ArgumentMatcher<Date> haveHourFieldEqualTo(final int hour) {
        return new ArgumentMatcher<Date>() {
                @Override
                public boolean matches(Object argument) {
                        return ((Date) argument).getHours() == hour;
                }
        };
}
```

Name	Matching rules
any(), any(Class<T> clazz)	any object or null, the same in a form which allows to avoid casting
anyBoolean(), anyByte(), anyChar(), anyDouble(), anyFloat(), anyInt(), anyLong(), anyShort(), anyString()	any object of the given type or null - preferred over generic any(Class<T> } clazz) for supported types
anyCollection(), anyList(),anyMap(), anySet()	respectively any collection type
anyCollectionOf(Class<T> clazz), anyListOf(Class<T> clazz), anyMapOf(Class<T> clazz), anySetOf(Class<T> clazz)	respectively any collection type in a generic friendly way
anyVararg()	Any vararg
eq(T value)	Any object that is equal to the given using equals() method
isNull, isNull(Class<T> clazz)	Null value
isNotNull, isNotNull(Class<T> clazz)	Not null value
isA(Class<T> clazz)	Any object that implements the given class
refEq(T value, String... excludeFields)	Any object that is equal to the given using reflection; some fields can be excluded
matches(String regex)	String that matches the given regular expression
startsWith(string), endsWith(string), contains(string) for a String class	string that starts with, ends with or contains the given string
aryEq(PrimitiveType value[]), aryEq(T[] value)	an array that is equal to the given array (has the same length and each element is equal)
cmpEq(Comparable<T> value)	any object that is equal to the given using compareTo() method
gt(value), geq(value), lt(value), leq(value) for primitive types and Comparable<T>	any argument greater, greater or equal, less, less or equal than the given value
argThat(org.hamcrest.Matcher<T> matcher)	any object that satisfies the custom matching
booleanThat(Matcher<Boolean> matcher), byteThat(matcher), charThat(matcher), doubleThat(matcher), floatThat(matcher), intThat(matcher), longThat(matcher), shortThat(matcher)	any object of the given type that satisfies the custom matching - preferred overgeneric argThat(matcher) for primitivesspy
and(first, second), or(first, second), not(first) for primitive types and T extending Object	an ability to combine results of the other matchers

Table 1: Selected Mockito matchers

STUBBING MULTIPLE CALLS TO THE SAME METHOD

Sometimes you want to return different values for subsequent calls of the same method. Returned values can be mixed with exceptions. The last value/behavior is used for all following calls.

```
@Test
public void shouldReturnLastDefinedValueConsistently() {
                WaterSource waterSource = mock(WaterSource.class);
                given(waterSource.getWaterPressure()).willReturn(3, 5);

                assertEquals(waterSource.getWaterPressure(), 3);
                assertEquals(waterSource.getWaterPressure(), 5);
                assertEquals(waterSource.getWaterPressure(), 5);
}
```

STUBBING VOID METHODS

As we've seen before, the stubbed method is passed as a parameter to a given / when method. This obviously means that you cannot use this construct for void methods. Instead, you should use willXXX..given or doXXX..when. See here:

```
@Test(expectedExceptions = WaterException.class)
public void shouldStubVoidMethod() {
                WaterSource waterSourceMock = mock(WaterSource.class);
                doThrow(WaterException.class).when(waterSourceMock).doSelfCheck();
                //the same with BDD semantics
                //willThrow(WaterException.class).given(waterSourceMock).doSelfCheck();

                waterSourceMock.doSelfCheck();

                //exception expected
}
```

do/willXXX methods family:

Method	Description
doThrow(Throwable toBeThrown) doThrow(Class<? extends Throwable> toBeThrown)	Throws given exception
doAnswer(Answer answer)	Uses user-created code to answer
doCallRealMethod()	Working with spy
doNothing()	Does nothing
doReturn(Object toBeReturned)	Returns given value (not for void methods)

will/doXXX methods are also handy when working with spy objects, as will be seen in the following section.

A given / when construct allows Mockito to internally use the type returned by a stubbed method to provide a typed argument in the will/thenReturn methods. Void is not a valid type of it causes a compilation error. Will/doReturn does not use that trick. Will/doReturn can be also used for stubbing non-void methods, though it is not recommended because it cannot detect wrong return method types at a compilation time (only an exception at runtime will be thrown).

Hot Tip It is not recommended to use do/ willReturn for stubbing non-void methods.

```
//compilation error - int expected, not boolean
//given(flowerMock.getNumberOfLeafs()).willReturn(true);

//only runtime exception
willReturn(true).given(flowerMock).getNumberOfLeafs();
```

STUBBING WITH A CUSTOM ANSWER

In some rare cases it can be useful to implement a custom logic, later used on a stubbed method invocation. Mockito contains a generic Answer interface allowing the implementation of a callback method and providing access to invocation parameters (used arguments, a called method, and a mock instance).

```
@Test
public void shouldReturnTheSameValue() {
  FlowerFilter filterMock = mock(FlowerFilter.class);
  given(filterMock.filterNoOfFlowers(anyInt())).will(returnFirstArgument());

  int filteredNoOfFlowers = filterMock.filterNoOfFlowers(TEST_NUMBER_OF_FLOWERS);

  assertEquals(filteredNoOfFlowers, TEST_NUMBER_OF_FLOWERS);
}

//reusable answer class
public class ReturnFirstArgument implements Answer<Object> {
  @Override
  public Object answer(InvocationOnMock invocation) throws Throwable {
    Object[] arguments = invocation.getArguments();
    if (arguments.length == 0) {
      throw new MockitoException("...");
    }
    return arguments[0];
  }

  public static ReturnFirstArgument returnFirstArgument() {
    return new ReturnFirstArgument();
  }
}
```

Warning: The need to use a custom answer may indicate that tested code is too complicated and should be re-factored.

VERIFYING BEHAVIOR

Once created, a mock remembers all operations performed on it. Important from the SUT perspective, these operations can easily be verified. In the basic form, use Mockito. verify (T mock) on the mocked method.

```
WaterSource waterSourceMock = mock(WaterSource.class);

waterSourceMock.doSelfCheck();

verify(waterSourceMock).doSelfCheck();
```

By default, Mockito checks if a given method (with given arguments) was called once and only once. This can be modified using a VerificationMode. Mockito provides the number of very meaningful verification modes. It is also possible to create a custom verification mode.

Name	Verifying method was...
times(int wantedNumberOfInvocations)	called exactly n times (one by default)fl
never()	never called
atLeastOnce()	called at least once
atLeast(int minNumberOfInvocations)	called at least n times
atMost(int maxNumberOfInvocations)	called at most n times
only()	the only method called on a mock
timeout(int millis)	interacted in a specified time range

```
verify(waterSourceMock,never()).doSelfCheck();
verify(waterSourceMock,times(2)).getWaterPressure();
verify(waterSourceMock,atLeast(1)).getWaterTemperature();
```

As an alternative to never (), which works only for the specified call, verifyZeroInteractions (Object ... mocks) method can be used to verify no interaction with any method of the given mock(s). Additionally, there is one more method available, called verifyNoMoreInteractions (Object ... mocks), which allows to ensure that no more interaction (except the already verified ones) was performed with the mock(s verifyNoMoreInteractions can be useful in some cases, but shouldn't be overused by using on all mocks in every test. Unlike other mocking frameworks, Mockito does not automatically verify all stubbed calls. It is possible to do it manually, but usually it is just redundant. The tested code should mainly care about values returned by stubbed methods. If a stubbed method was not called, while being important from the test perspective, something else should break in a test. Mockito's philosophy allows the test writer to focus on interesting behaviors in the test for the SUT and his collaborators.

Hot Tip Mockito does not automatically verify calls

VERIFYING CALL ORDER

Mockito enables you to verify if interactions with a mock were performed in a given order using the InOrder API. It is possible to create a group of mocks and verify the call order of all calls within that group.

```
@Test
public void shouldVerifyInOrderThroughDifferentMocks() {
  WaterSource waterSource1 = mock(WaterSource.class);
  WaterSource waterSource2 = mock(WaterSource.class);

  waterSource1.doSelfCheck();
  waterSource2.getWaterPressure();
  waterSource1.getWaterTemperature();

  InOrder inOrder = inOrder(waterSource1, waterSource2);
  inOrder.verify(waterSource1).doSelfCheck();
  inOrder.verify(waterSource2).getWaterPressure();
  inOrder.verify(waterSource1).getWaterTemperature();
}
```

Warning: The need to use a custom answer may indicate that tested code is too complicated and should be re-factored.

VERIFYING WITH ARGUMENT MATCHING

During a verification of an interaction, Mockito uses equals () methods on the passed arguments. This is usually enough. It is also possible to use the standard matchers, described earlier about stubbing, as well as custom matchers. However, in some situations it may be helpful to keep the actual argument value to make custom assertions on it. Mockito offers an ArgumentCaptor class, enabling us to retrieve the argument passed to a mock.

```
//when
flowerSearcherMock.findMatching(searchCriteria);

//then
ArgumentCaptor<SearchCriteria> captor = ArgumentCaptor.forClass(SearchCriteria.class);
verify(flowerSearcherMock).findMatching(captor.capture());
SearchCriteria usedSearchCriteria = captor.getValue();
assertEquals(usedSearchCriteria.getColor(),"yellow");
assertEquals(usedSearchCriteria.getNumberOfBuds(), 3);
```

ArgumentCaptor can be also created using @Captor annotation (see appropriate section with annotations).

Warning: It is recommended to use ArgumentCaptor with verification, but not with stubbing. Creating and using a captor in two different test blocks can decrease test readability. In addition to a situation when a stubbed method is not called, no argument is captured, which can be confusing.

Hot Tip It is possible to retrieve arguments of all calls of a given method using captor . getAllValues ( ).

Warning: When an SUT internally uses the same object reference for multiple calls on a mock, every time changing its internal state (e.g., adding elements to the same list) captor . getAllValues () will return the same object in a state for the last call.

VERIFYING WITH TIMEOUT

Mockito lets you verify interactions within a specified time frame. It causes a verify() method to wait for a specified period of time for a requested interaction rather than fail immediately if that had not already happened. It can be useful while testing multi-threaded systems.

```
@Test
public void shouldFailForLateCall() {
                WaterSource waterSourceMock = mock(WaterSource.class);
                Thread t = waitAndCallSelfCheck(40, waterSourceMock);

                t.start();

                verify(waterSourceMock, never()).doSelfCheck();
                try {
                                verify(waterSourceMock, timeout(20)).doSelfCheck();
                                fail("verification should fail");
                } catch (MockitoAssertionError e) {
                                //expected
                }
}
```

Warning: Currently , verifying with timeout doesn't work with inOrder verification.

Warning: All the multi-thread tests can become non-deterministic (e.g., under heavy load).

SPYING ON REAL OBJECTS

With Mockito, you can use real objects instead of mocks by replacing only some of their methods with the stubbed ones. Usually there is no reason to spy on real objects, and it can be a sign of a code smell, but in some situations (like working with legacy code and IoC containers) it allows us to test things impossible to test with pure mocks.

```
@Test
public void shouldStubMethodAndCallRealNotStubbedMethod() {
                Flower realFlower = new Flower();
                realFlower.setNumberOfLeafs(ORIGINAL_NUMBER_OF_LEAFS);
                Flower flowerSpy = spy(realFlower);
                willDoNothing().given(flowerSpy).setNumberOfLeafs(anyInt());

                flowerSpy.setNumberOfLeafs(NEW_NUMBER_OF_LEAFS); //stubbed - should do nothing

                verify(flowerSpy).setNumberOfLeafs(NEW_NUMBER_OF_LEAFS);
                assertEquals(flowerSpy.getNumberOfLeafs(), ORIGINAL_NUMBER_OF_LEAFS); //value was not
changed
}
```

Hot Tip When working with spies it is required to use the willXXX..given/doXXX..when methods family instead of given .. willXXX/when.. thenXXX. This prevents unnecessary calls to a real method during stubbing.

Warning: While spying, Mockito creates a copy of a real object, and therefore all interactions should be passed using the created spy.

ANNOTATIONS

Mockito offers three annotations–@Mock, @Spy, @Captor– to simplify the process of creating relevant objects using static methods. @InjectMocks annotation simplifies mock and spy injection. It can inject objects using constructor injection, setter injection or field injection.

```
//with constructor: PlantWaterer(WaterSource waterSource,
//   WateringScheduler wateringScheduler) {...}

public class MockInjectingTest {
  @Mock
  private WaterSource waterSourceMock;

  @Spy
  private WateringScheduler wateringSchedulerSpy;

  @InjectMocks
  private PlantWaterer plantWaterer;

  @BeforeMethod
  public void init() {
    MockitoAnnotations.initMocks(this);
  }

  @Test
  public void shouldInjectMocks() {
    assertNotNull(plantWaterer.getWaterSource());
    assertNotNull(plantWaterer.getWateringScheduler());
  }
}
```

Annotation	Responsibility
@Mock	Creates a mock of a given type
@Spy	Creates a spy of a given object
@Captor	Creates an argument captor of a given type
@InjectMocks	Creates an object of a given type and injects mocks and spies existing in a test

Hot Tip To get annotations to function, you need to either call MockitoAnnotations.initMocks(testClass) (usually in a @Before method) or use MockitoJUnit4Runner as a JUnit runner.

Hot Tip To make a field injection with @InjectMock, Mock-ito internally uses reflection. It can be especially useful when, in the production code, dependencies are injected directly to the private fields (e.g., by an IoC framework).

CHANGING THE MOCK DEFAULT RETURN VALUE

Mockito enables us to redefine a default value returned from **non-stubbed** methods

Default Answer	Description
RETURNS_DEFAULTS	Returns a default "empty" value (e.g., null, 0, false, empty collection) - used by default
RETURNS_SMART_NULLS	Creates a spy of a given object
RETURNS_MOCKS	Returns a default "empty" value, but a mock instead of null
RETURNS_DEEP_STUBS	Allows for a simple deep stubbing (e.g., Given(ourMock.getObject().getValue()). willReturn(s))
CALLS_REAL_METHODS	Call a real method of spied object

Warning: The last three default answers should not be needed when working with well-crafted, testable code. The behavior can be configured per mock during its creation or globally for all tests using GlobalConfiguration mechanism (it helps to use RETURNS_SMART_NULLS by default).

```
PlantWaterer plantWatererMock =
      mock(PlantWaterer.class, Mockito.RETURNS_DEEP_STUBS);
given(plantWatererMock.getWaterSource().getWaterPressure()).willReturn(5);

@Mock(answer = Answers.RETURNS_SMART_NULLS)
private PlantWaterer plantWatererMock;
```

Sample verbose exception received using SmartNull:

```
org.mockito.exceptions.verification.SmartNullPointerException:
You have a NullPointerException here:
-> at PlantWaterer.generateNPE(PlantWaterer.java:24)
because this method call was *not* stubbed correctly:
-> at PlantWaterer.generateNPE(PlantWaterer.java:24)
wateringScheduler.returnNull();

                at PlantWaterer.generateNPE(PlantWaterer.java:24)
                at DefaultValuesTest.shouldReturnNicerErrorMessageOnNPE(DefaultValuesTest.java:64)
```

Hot Tip

Check "Beyond the Mockito Refcard" (see link below) to get a complete tutorial on how to easily configure SmartNulls for the whole project. Also, other useful information is there that's not in this Refcard.

RESETTING MOCK

In some rare cases (like using a mock as a bean in an IoC container) you may need to reset a mock using the Mockito. reset (T ... mocks) static method. This causes the mock to forget all previous behavior and interactions.

Warning: In most cases, using the reset method in a test is a code smell and should be avoided. Splitting a large test into smaller ones with mocks

LIMITATIONS

Mockito has a few limitations worth remembering. They are generally technical restrictions, but Mockito authors believe using hacks to work around them would encourage people to write poorly testable code. Mockito cannot:

- mock final classes
- mock enums
- mock final methods
- mock static methods
- mock private methods
- mock hashCode() and equals()

Nevertheless, when working with a badly written legacy code, remember that some of the mentioned limitations can be mitigated using the PowerMock or JMockit libraries.

Hot Tip

PowerMock or JMockit can be used to work with code that cannot be mocked with pure Mockito.

FURTHER READING

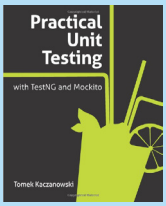
- http://blog.solidsoft.info/mockito-docs/ - official Mockito documentation (redirect to the current version)
- http://blog.solidsoft.info/beyond-the-mockito-refcard/ - a series of articles extending information contained In this Refcard

AUTHOR BIO

Marcin Zajaczkowski

Marcin Zajaczkowski is an experienced architect who specializes in creating high quality software. Aligning himself closely with the Agile methodologies and the Software Craftsmanship movement, Marcin believes in the value of good, testable and maintainable code. Marcin aims to forge excellent software that makes the client delighted and the team proud of how the code itself looks. In his teaching as a conference speaker, college lecturer, IT coach and trainer, Marcin shows how to guide software development effectively using tests (with TDD, pair programming, Clean Code, design patterns, etc.) and maintain a quality-oriented development environment (with CI, Sonar, automatic deployment, etc.). He is also a FOSS Projects author and contributor, a Linux enthusiast, and blogs at http://blog.solidsoft.info/ Marcin co-operates with Pragmatists (http://pragmatists.pl/) - a freaking good agile outsourcing Java shop based in Poland.

RECOMMENDED BOOK



Practical Unit Testing with Test NG and Mockito

This book explains in detail how to implement unit tests using two very popular open source Java technologies: TestNG and Mockito. It presents a range of techniques necessary to write high quality unit tests - e.g. mocks, parametrized tests and matchers. It also discusses trade-offs related to the choices we have to make when dealing with some real-life code issues. http://practicalunittesting.com/

DZone, Inc.
150 Preston Executive Dr.
Suite 200
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-48-6
ISBN-10: 1-936502-48-8
50795
9 781936 502486
$7.95

Version 1.0