

Sumar

Lista de Snippets.....	1
Obiective	2
1. Aspecte preliminare	2
2. Concepte folosite în testarea unitară.....	2
3. Obiecte Dummy.....	3
4. Obiecte Fake.....	3
5. Obiecte Stub	4
6. Obiecte Mock. Obiecte Spy	8
7. Framework-ul Mockito	8
01. Configurarea Mockito în proiectul de testare	8
02. Crearea <i>obiectelor mock</i>	8
03. Definirea apelurilor metodelor pentru obiectele mock	9
04. Verificarea apelurilor metodelor pentru obiectele mock	10
05. Crearea și utilizarea <i>obiectelor spy</i>	10
06. Mock vs Spy în Mockito.....	12
8. Stubs vs. Mocks	13
9. Resurse bibliografice	14

Lista de Snippets

Snippet 1. Clasa <code>QuizService</code>	2
Snippet 2. <i>Obiecte dummy</i> pentru testarea metodei <code>addQuiz (...)</code>	3
Snippet 3. In-memory repository	4
Snippet 4. Clasa de test pentru <code>QuizService</code> folosind repository in-memory (obiect fake)	4
Snippet 5. Clasa <code>RepositoryStub</code>	5
Snippet 6. Clasa <code>RepositoryStubException</code>	5
Snippet 7. Clasa de test <code>QuizServiceTestRepositoryStub</code>	6
Snippet 8. Clasa de test <code>QuizServiceTestRepositoryStubException</code>	7
Snippet 9. Configurarea Mockito în proiectul de testare	8
Snippet 10. Crearea obiectelor mock folosind metoda <code>mock ()</code>	8
Snippet 11. Crearea obiectelor mock folosind adnotarea <code>@Mock</code>	9
Snippet 12. Definirea apelurilor metodelor folosind metodele <code>when . . . then...</code> și <code>doNothing ()</code>	9
Snippet 13. Definirea apelurilor metodelor folosind metoda <code>doAnswer ()</code>	10
Snippet 14. Definirea apelurilor metodelor folosind metoda <code>doThrow ()</code>	10
Snippet 15. Verificarea apelului metodelor pentru obiectele mock folosind metoda <code>verify ()</code>	10
Snippet 16. Definirea si utilizarea obiectelor spy folosind metoda <code>spy ()</code>	11
Snippet 17. Definirea si utilizarea obiectelor spy folosind adnotarea <code>@Spy</code>	12

Obiective

- Definirea și exemplificarea conceptelor: **dummy**, **fake**, **stub**, **mock**, **spy**.
- Exemplificarea testării unitare în Java, folosind JUnit 5 și Mockito.

1. Aspecte preliminare

1. Pentru exemplificarea conceptelor se va considera aplicația pentru gestionarea quiz-urilor – **QuizProject**.
2. Proiectul Java are o arhitectură stratificată, iar clasele sunt organizate în pachete:
domain [Quiz, Difficulty],
validation [QuizValidator],
repository [IRepository, Exception,
 fake[RepositoryInMemory],
 stub[RepositoryStub, RepositoryStubException],
 file[FileRepository]],
service [QuizService],
ui [QuizViewer];
3. Presupunem că la nivelul clasei QuizService se dorește realizarea testării unitare. Particularitățile clasei QuizService sunt (vezi Snippet 1):
 - există un atribut de tip IRepository, adică avem o dependență care trebuie să fie gestionată la rularea testelor pentru obiecte de tip QuizService;
 - avem metodele addQuiz(...), allQuizzes(), size() și maxScoreQuizCounter() pentru care dorim să scriem teste.

```
13 public class QuizService {
14
15     private IRepository repository;
16
17     public QuizService(IRepository repository) { this.repository = repository; }
20
21     public QuizService() {}
24
25     public void setRepository(IRepository repository) { this.repository = repository; }
28
29     public void addQuiz(Quiz quiz) throws RepositoryException {...}
32
33     public List<Quiz> allQuizzes() { return repository.getAll(); }
36
37     public int size() { return repository.size(); }
40
41     public int maxScoreQuizCounter() {...}
61 }
```

Snippet 1. Clasa QuizService

2. Concepte folosite în testarea unitară

La nivelul testării unitare se folosesc diferiți termeni pentru a indica obiecte, stări sau caracteristici care apar la proiectarea cazurilor de testare. Literatura de specialitate indică abordări diferite în descrierea terminologiei folosite. În acest tutorial se va folosi abordarea lui Gerard Meszaros și Martin Folwer. Aceste concepte sunt denumite generic *Test Doubles*, sugerând faptul că simulează funcționarea obiectelor utilizate în realitate (vezi Figure 1):

- **Dummy** – obiecte care sunt transmise ca parametri dar care nu sunt folosite de metodele apelate; e.g., diverși parametri precizați doar pentru a respecta semnatura metodei apelate;
- **Fake** – obiecte cu implementări funcționale/utilizabile, dar simpliste, care nu sunt adecvate pentru a fi incluse în livrabilul către client; e.g., o colecție de date *in-memory*;

- **Stubs** – obiecte sau metode ale unor obiecte care furnizează rezultate prestabilite atunci când sunt apelate în cadrul unui test; nu au altă utilitate în afara contextului testării unde au fost definite;
- **Spies** – obiecte stub care pot păstra/reține informații referitoare la modul în care au fost folosite; e.g., un serviciu pentru e-mail care reține numărul de mesaje transmise;
- **Mocks** – obiecte pentru care s-a stabilit un anumit comportament (behavior expectations) ce reprezintă o specificație a apelurilor pe care aceste obiecte se așteaptă să le primească.

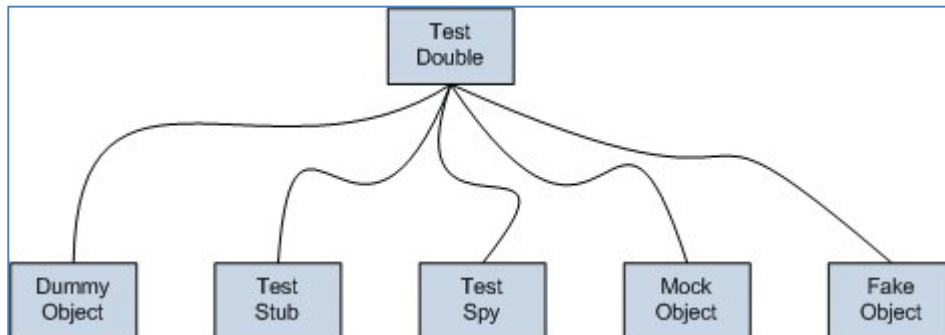


Figure 1. Clasificarea obiectelor Test Double

3. Obiecte Dummy

Obiectele dummy sunt obiecte care sunt transmise ca parametri unei metodei apelate, dar care nu sunt folosite de aceasta, e.g., diverși parametri precizați doar pentru a respecta semnatura metodei apelate.

Utilizarea *obiectelor dummy* pentru Lab02/Lab03/Lab04.

1. Presupunem că se dorește testarea metodei `addQuiz(...)`.
2. Un quiz are atributele: `id: String`, `noQuestions: int`, `difficulty: Difficulty {Easy, Medium, Hard}`, `correctAnswers: int`.
3. Pentru proiectarea cazurilor de testare bazate pe criteriul black-box se aplică **ECP** și **BVA**. Numărul de cazuri de testare care trebuie implementate este consistent, având în vedere:
 - numărul de atribute al entității `Quiz`;
 - numărul de condiții care pot fi impuse asupra atributelor.
4. Vom considera proiectarea cazurilor de testare doar pentru o parte dintre atributele entității `Quiz`, celelalte având o valoare care nu este relevantă pentru test; din acest motiv aceste atribute vor fi considerate *obiecte dummy*.
5. De exemplu, atributele `noQuestions` și `correctAnswers` iau valori conform tehnicilor **ECP** și **BVA** aplicate, iar valorile pentru atributele `id` și `difficulty` sunt considerate *obiecte dummy* (vezi Snippet 2).

```

assertEquals(5, service.allQuizzes().size());
service.addQuiz(new Quiz("q6", 5, "Medium", 3));
assertEquals(6, service.allQuizzes().size());

assertEquals(5, service.allQuizzes().size());
service.addQuiz(new Quiz("q6", 0, "Medium", 3));
Assert.fail("Repository Exception not thrown");
  
```

Snippet 2. Obiecte dummy pentru testarea metodei `addQuiz(...)`

4. Obiecte Fake

Obiectele fake sunt obiecte cu implementări funcționale/utilizabile, dar simpliste, nefiind adecvate pentru a fi incluse în produsul soft livrat la client; e.g., o colecție de date *in-memory*.

Utilizarea *obiectelor fake* pentru Lab02/Lab03/Lab04.

1. Presupunem că se dorește testarea metodei `addQuiz(...)`.

- Un quiz are atributele: id: String, noQuestions: int, difficulty: Difficulty {Easy, Medium, Hard}, correctAnswers: int.
- Considerăm o implementare *in-memory* pentru atributul de tip IRepository a clasei QuizService (vezi Snippet 3).

```

13 public class RepositoryInMemory implements IRepository {
14
15     private Vector<Quiz> quizzes;
16     private QuizValidator validator;
17
18     public RepositoryInMemory(QuizValidator validator) throws RepositoryException {
19         quizzes = new Vector<Quiz>();
20         this.validator = validator;
21         populate();
22     }
23
24     public void populate() throws RepositoryException
25     {
26         add(new Quiz( id: "q1", noQ: 10, difficulty: "Easy", correctAnswers: 7));
27         add(new Quiz( id: "q2", noQ: 15, difficulty: "Medium", correctAnswers: 3));
28         add(new Quiz( id: "q3", noQ: 9, difficulty: "Easy", correctAnswers: 9));
29         add(new Quiz( id: "q4", noQ: 5, difficulty: "Hard", correctAnswers: 2));
30         add(new Quiz( id: "q5", noQ: 20, difficulty: "Medium", correctAnswers: 9));
31     }
32
33     @Override
34     public void add(Quiz quiz) throws RepositoryException{...}

```

Snippet 3. In-memory repository

- Pentru testarea metodelor din clasa QuizService se poate folosi un *object fake*, i.e., repository *in-memory* (vezi Snippet 4).

```

19 public class QuizServiceTestWithRepoInMemory {
20
21     private QuizService service;
22
23     @Before
24     public void setUp() throws RepositoryException {
25         QuizValidator validator= new QuizValidator();
26         IRepository repo = new RepositoryInMemory(validator);
27         service = new QuizService(repo);
28     }
29
30     @Test
31     public void test01_add_valid_Quiz() throws RepositoryException{...}
32
33     @After
34     public void tearDown() { service = null; }

```

Snippet 4. Clasa de test pentru QuizService folosind repository in-memory (object fake)

5. Obiecte Stub

Obiectele stub sunt obiecte sau metode ale unor obiecte care furnizează rezultate prestabilite atunci când sunt apelate în cadrul unui test. Aceste obiecte nu au altă utilitate în afara contextului testării unde au fost definite, permițând un control absolut asupra datelor de intrare folosite la nivelul testării

unitare. De exemplu, pentru a simula conexiunea la o bază de date, obiectele stub create permit mimarea oricărui scenariu (conectare cu succes, conectare eșuată, etc.) fără a avea o bază de date concretă.

Utilizarea *obiectelor stub* pentru Lab04:

1. Presupunem că se dorește testarea metodelor din clasa QuizService.
2. Presupunem că avem definite clasele stub RepositoryStub și RepositoryStubException (vezi Snippet 5 și Snippet 6).

```
public class RepositoryStub implements IRepository {  
  
    private List<Quiz> quizzes;  
  
    public RepositoryStub() throws RepositoryException {  
        quizzes = Arrays.asList(new Quiz("q1", 10, "Easy", 7),  
                                new Quiz("q2", 15, "Medium", 3),  
                                new Quiz("q3", 9, "Easy", 9));  
    }  
  
    @Override  
    public void add(Quiz quiz) throws RepositoryException{  
        //default behaviour - does nothing  
    }  
  
    @Override  
    public List<Quiz> getAll(){  
        return new ArrayList<>(quizzes);  
    }  
  
    @Override  
    public int size() {  
        return quizzes.size();  
    }  
}
```

Snippet 5. Clasa RepositoryStub

```
public class RepositoryStubException implements IRepository {  
  
    private List<Quiz> quizzes;  
  
    public RepositoryStubException() throws RepositoryException {  
        quizzes = Arrays.asList(new Quiz("q1", 10, "Easy", 7),  
                                new Quiz("q2", 15, "Medium", 3),  
                                new Quiz("q3", 9, "Easy", 9));  
    }  
  
    @Override  
    public void add(Quiz quiz) throws RepositoryException{  
        //default behaviour - throws exception  
        throw new RepositoryException("invalid quiz");  
    }  
  
    @Override  
    public List<Quiz> getAll(){  
        return new ArrayList<>(quizzes);  
    }  
  
    @Override  
    public int size() {  
        return quizzes.size();  
    }  
}
```

Snippet 6. Clasa RepositoryStubException

3. Clasele stub au comportament stabilit la implementare. Orice apel al metodelor va avea același comportament indiferent de valorile parametrilor de intrare, i.e., la fiecare apel se va returna aceeași valoare.
4. **Comportamentul obiectelor stub este static, prestabilit. Nu se poate modifica la utilizare.**
5. Testarea metodei `public void addQuiz(Quiz quiz) throws RepositoryException` din clasa `QuizService` presupune descrierea unor teste care pot să arunce sau nu excepție.
6. Clasa `QuizServiceTestRepositoryStub` (vezi Snippet 7) nu va putea să pună în evidență testarea cu și fără excepție a metodei `addQuiz(...)`. Testul pentru aruncarea unei excepții va eșua de fiecare data deoarece stub-ul definit nu aruncă excepție.

```
public class QuizServiceTestWithStub {

    private QuizService service;

    @Before
    public void setUp() throws RepositoryException {
        IRepository repo = new RepositoryStub();
        service = new QuizService(repo);
    }

    @Test
    public void test01_add_valid_Quiz() throws RepositoryException{
        //already exists in repo
        // service.addQuiz(new Quiz("q1", 10, "Easy", 7));
        // service.addQuiz(new Quiz("q2", 15, "Medium", 3));
        // service.addQuiz(new Quiz("q3", 9, "Easy", 9));

        assertEquals(3, service.allQuizzes().size());
        service.addQuiz(new Quiz("q6", 5, "Medium", 3));

        //behaviour of the stub for the addQuiz method - no quiz was added
        //assert examples
        assert true;
        assertEquals(3, service.allQuizzes().size());
        assert 3 == service.allQuizzes().size();
    }

    @Test(expected= RepositoryException.class)
    public void test02_add_invalid_Quiz() throws RepositoryException{
        System.out.println("test02_add_invalidQuiz() will always fail as the
stub does not throw the expected exception" );
        //already exists in repo
        // service.addQuiz(new Quiz("q1", 10, "Easy", 7));
        // service.addQuiz(new Quiz("q2", 15, "Medium", 3));
        // service.addQuiz(new Quiz("q3", 9, "Easy", 9));

        assertEquals(3, service.allQuizzes().size());
        service.addQuiz(new Quiz("q6", 0, "Medium", 3));
        //behaviour of the stub for the addQuiz method - no quiz was added
        //RepositoryStub cannot be used to test the behaviour for invalid
quizzes
        //this test will always fail, as no exceptions was thrown by the stub
!!!
        Assert.fail("Repository Exception not thrown");
    }
}
```

Snippet 7. Clasa de test `QuizServiceTestRepositoryStub`

7. Pentru a pune în evidență aruncarea excepției se va defini clasa de test `QuizServiceTestRepositoryStubException` (vezi Snippet 8) care folosește un

obiect stub al clasei `RepositoryStubException` care aruncă excepție la fiecare apel al metodei `addQuiz(...)`.

```
public class QuizServiceTestWithStubException {

    private QuizService service;

    @Before
    public void setUp() throws RepositoryException {
        IRepository repo = new RepositoryStubException();
        service = new QuizService(repo);
    }

    @Test
    public void test01_add_valid_Quiz() {
        //already exists in repo
        // service.addQuiz(new Quiz("q1", 10, "Easy", 7));
        // service.addQuiz(new Quiz("q2", 15, "Medium", 3));
        // service.addQuiz(new Quiz("q3", 9, "Easy", 9));

        assertEquals(3, service.allQuizzes().size());
        try {
            service.addQuiz(new Quiz("q6", 5, "Medium", 3));
            Assert.fail("No exception was thrown");
        } catch (RepositoryException e) {
            //behaviour of the stub for the addQuiz method - an exception is
            thrown

            //assert examples
            assert true;
            assertEquals(3, service.allQuizzes().size());
            assert 3 == service.allQuizzes().size();
        }
    }

    @Test(expected= RepositoryException.class)
    public void test02_add_invalid_Quiz() throws RepositoryException{
        //already exists in repo
        // service.addQuiz(new Quiz("q1", 10, "Easy", 7));
        // service.addQuiz(new Quiz("q2", 15, "Medium", 3));
        // service.addQuiz(new Quiz("q3", 9, "Easy", 9));

        assertEquals(3, service.allQuizzes().size());
        service.addQuiz(new Quiz("q6", 0, "Medium", 3));
        //behaviour of the stub for the addQuiz method - an exception was
        thrown
        Assert.fail("Repository Exception not thrown");
    }

    @Test
    public void test03_MaxScoreQuizCounter_validOutput() {
        //already exists in repo
        // service.addQuiz(new Quiz("q1", 10, "Easy", 7));
        // service.addQuiz(new Quiz("q2", 15, "Medium", 3));
        // service.addQuiz(new Quiz("q3", 9, "Easy", 9));

        //assert examples
        assertEquals(1, service.maxScoreQuizCounter());
        assert 1 == service.maxScoreQuizCounter();
    }
}
```

Snippet 8. Clasa de test `QuizServiceTestRepositoryStubException`

6. Obiecte Mock. Obiecte Spy

Obiectele mock sunt obiecte pentru care s-a stabilit un anumit comportament (behavior expectations) și reprezintă o specificație pentru apelurile pe care aceste obiecte se așteaptă să le primească. Obiectele mock contorizează apelurile primite. În cadrul testelor se poate verifica dacă toate acțiunile așteptate au avut loc.

Obiectele spy sunt *obiecte stub* care pot păstra/reține informații referitoare la modul în care au fost folosite; e.g., un serviciu pentru e-mail care reține numărul de mesaje transmise.

Utilizarea unui framework pentru mocking (necesară pentru rezolvarea unor task-uri ale Lab04) este exemplificată în **Secțiunea 7 - Mockito**. **Pentru rezolvarea cerințelor din Lab04 se poate folosi orice alt framework pentru mocking.**

7. Framework-ul Mockito

Mockito este un framework pentru gestionarea obiectelor mock. Mockito permite crearea și configurarea obiectelor mock, facilitând testarea claselor cu dependențe. Mockito este folosit la nivelul testării unitare împreună cu alte framework-uri de testare, e.g., JUnit, TestNG.

Utilizarea **obiectelor mock** și **metodelor stub** pentru Lab04 folosind Mockito și JUnit 4 este prezentată pe scurt, mai jos.

01. Configurarea Mockito în proiectul de testare

- Se adaugă în fișierul *pom.xml* dependența pentru *mockito-core*. Dacă este utilizat framework-ul JUnit 5 este necesară adăugarea și celei de a doua dependențe, către *mockito-junit-jupiter* (vezi Snippet 9).

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <version>2.23.0</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-junit-jupiter</artifactId>
  <version>2.23.0</version>
  <scope>test</scope>
</dependency>
```

Snippet 9. Configurarea Mockito în proiectul de testare

02. Crearea obiectelor mock

- Varianta 1:** folosind metoda statică *mock(...)*
 - Obiectele mock se pot crea prin apelul metodei statice *mock(...)*, care primește ca parametru tipul obiectului mock care va fi creat (vezi Snippet 10).
 - Mockito permite ca parametru o interfață sau o clasă.

```
29 public class QuizServiceTestWithMockConstructor {
30
31     private IRepository repo;
32
33     private QuizService service;
34
35     @Before
36     public void setUp() {
37         repo = mock(IRepository.class);
38         service = new QuizService(repo);
39     }
```

Snippet 10. Crearea obiectelor mock folosind metoda *mock()*

- **Varianta 2:** folosind adnotările `@Mock` și `@InjectMocks`
 - Obiectele mock se pot crea prin folosirea adnotării `@Mock`;
 - Injectarea dependențelor mock create se realizează folosind adnotarea `@InjectMocks` (vezi Snippet 11);
 - Injectarea dependențelor mock se realizează aplicând, în ordine, una dintre variantele: un constructor definit, o metodă setter definită, atributul/câmpul propriu zis;
 - Utilizarea obiectelor mock create prin folosirea adnotărilor necesită apelul metodei statice `MockitoAnnotations.initialize(this)`. În caz contrar se aruncă excepția `NullPointerException`. Metoda `initialize(this)` se apelează înainte de execuția testului propriu-zis, de obicei în metoda adnotată cu `@Before` (vezi Snippet 11).

```

25 public class QuizServiceTestWithMockAnnotations {
26
27     @Mock
28     private IRepository repo;
29
30     @InjectMocks
31     private QuizService service;
32
33     @Before
34     public void setUp() {
35         MockitoAnnotations.initMocks( testClass: this );
36     }

```

Snippet 11. Crearea obiectelor mock folosind adnotarea `@Mock`

03. Definirea apelurilor metodelor pentru obiectele mock

Prin definirea apelurilor metodelor pentru obiectele mock se stabilește comportamentul pentru obiectele mock definite.

- **Varianta 1:** pentru metodele care au tip returnat diferit de `void` (vezi Snippet 12);
 - Se precizează obiectul mock, numele metodei, parametrii folosiți la apel și valoarea returnată: `Mockito.when(mock.method(args)).thenReturn(value)`;
- **Varianta 2:** pentru metodele care returnează `void` (vezi Snippet 12, Snippet 13, Snippet 14);
 - metoda `doNothing()`;
 - se precizează obiectul mock, numele metodei, parametrii folosiți la apel: `Mockito.doNothing().when(mock).method(args)`;
 - `doNothing()` este comportamentul implicit pentru metodele care returnează `void`;
 - metoda `doAnswer()`;
 - se descrie un anumit comportament pentru obiectul mock;
 - metoda `doThrow()`.

```

@Test
public void test05_add_valid_Quiz2() throws RepositoryException {
    Quiz q= new Quiz("q", 5, "Medium", 3);
    Quiz q1= new Quiz("q1", 10, "Easy", 9);
    Mockito.when(repo.getAll()).thenReturn(Arrays.asList(q1));
    Mockito.doNothing().when(repo).add(q);

    service.addQuiz(q);
    ...
}

```

Snippet 12. Definirea apelurilor metodelor folosind metodele `when...then...` și `doNothing()`

```

Quiz q= new Quiz("q", 5, "Medium", 3);
Mockito.doAnswer((Answer<Void>) invocation -> {
    Object[] arguments = invocation.getArguments();
    if (arguments != null && arguments.length == 1 && arguments[0] != null) {
        Quiz quiz = (Quiz) arguments[0];
        System.out.println(quiz);
    }
    return null;
}).when(repo).add(q); //same behaviour as test_add_valid_Quiz2

service.addQuiz(q);

```

Snippet 13. Definirea apelurilor metodelor folosind metoda doAnswer()

```

@Test(expected= RepositoryException.class)
public void test06_add_invalid_Quiz1() throws RepositoryException{
    Quiz q1= new Quiz("q1", -3, "Hard", 2);

    Mockito.doThrow(RepositoryException.class).when(repo).add(q1);
    service.addQuiz(q1);
}

```

Snippet 14. Definirea apelurilor metodelor folosind metoda doThrow()

04.Verificarea apelurilor metodelor pentru obiectele mock

- Se realizează atât pentru metodele pentru care timpul returnat este void sau nu;
 - metoda verify() cu diferite modalități de verificare a numărului de apeluri: times(int), atLeast(int), atMost(int), calls(int), never(), only(int), atLeastOnce() (vezi Snippet 15);

```

@Test
public void test05_add_valid_Quiz2() throws RepositoryException {
    Quiz q= new Quiz("q", 5, "Medium", 3);
    Quiz q1= new Quiz("q1", 10, "Easy", 9);
    Mockito.when(repo.getAll()).thenReturn(Arrays.asList(q1));
    Mockito.doNothing().when(repo).add(q);

    service.addQuiz(q);

    Mockito.verify(repo, times(1)).add(q);
    Mockito.verify(repo, never()).getAll();

    //assert examples
    assert true;
    assertEquals(1, service.allQuizzes().size());
    assert 1 == service.allQuizzes().size();

    Mockito.verify(repo, times(2)).getAll();
}

```

Snippet 15. Verificarea apelului metodelor pentru obiectele mock folosind metoda verify()

05. Crearea și utilizarea obiectelor spy

- **Varianta 1:** folosind metoda spy();
 - permite definirea de obiecte mock cu comportament parțial real și parțial mock (vezi Snippet 16);
 - se consideră un obiect concret din care, prin apelul metodei spy() se creează un obiect mock pentru care se poate modifica comportamentul specific unei metode prin definirea unui stub (prin apelul metodelor when(...).thenReturn(...) sau doReturn(...).when(...).<apel_metoda>);

```

public class QuizValidatorPartialMock {

    private QuizValidator validator;

    @Before
    public void setUp() {
        validator = new QuizValidator();
    }

    @Test
    public void test01_partialMock() {
        QuizValidator spiedValidator = spy(validator);

        Quiz q1 = new Quiz("q1", 10, "Easy", 9);

        //the real validate method is called
        assert (spiedValidator.validate(q1).equals(new ArrayList<String>()));

        //invalid Quiz "Hardest" instead of "Hard"
        Quiz q2 = new Quiz("q2", 7, "Hardest", 3);
        //the behaviour of the validate method for object q2 is stubbed to
        consider it a valid Quiz
        Mockito.when(spiedValidator.validate(q2)).thenReturn(new
        ArrayList<String>());

        //the stub validate method is called, with param q2
        assert spiedValidator.validate(q2).equals(new ArrayList<String>());

        //call again the real validate method, with param q1
        assert (spiedValidator.validate(q1).equals(new ArrayList<String>()));

        Mockito.verify(spiedValidator, times(2)).validate(q1);

        //the real validator is used to validate q2
        assert validator.validate(q2).equals(Arrays.asList("Difficulty field:
        [invalid difficulty not in {easy, medium, hard}]\n"));
    }
}

```

Snippet 16. Definirea și utilizarea obiectelor spy folosind metoda spy()

- **Varianta 2:** folosind adnotarea @Spy;
 - permite definirea de obiecte mock cu comportament parțial real și parțial mock folosind adnotarea @Spy și apelând metoda MockitoAnnotations.initMocks(this) (vezi Snippet 17);
 - comportamentul este similar cu cel al obiectului spy creat folosind metoda spy() (Varianta 1);

```

public class QuizValidatorSpyAnnotation {

    @Spy
    private QuizValidator validator;

    @Before
    public void setUp() {
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void test01_spy() {
        //invalid Quiz "Hardest" instead of "Hard"
        Quiz q1 = new Quiz("q1", 7, "Hardest", 3);

        //the behaviour of validate for object q2 is stubbed to consider it a
        valid Quiz
    }
}

```

```

Mockito.when(validator.validate(q1)).thenReturn(new
ArrayList<String>());

assert validator.validate(q1).equals(new ArrayList<String>());

//invalid Quiz "Hardest" instead of "Hard"
Quiz q2= new Quiz("q1", 7, "Hardest", 3);

//assert examples
assert validator.validate(q2).equals(new ArrayList<String>()) ==
false;
assert validator.validate(q2).equals(Arrays.asList("Difficulty field:
[invalid difficulty not in {easy, medium, hard}]\n"));
}

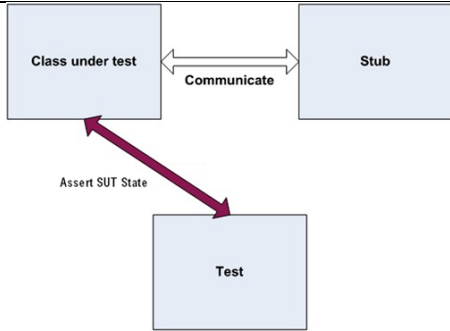
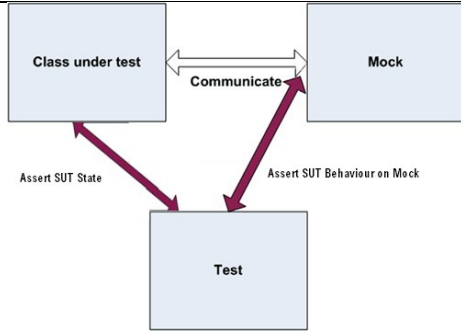
```

Snippet 17. Definirea si utilizarea obiectelor spy folosind adnotarea @Spy

06.Mock vs Spy în Mockito

Aspect analizat	Mocks	Spies
<i>Creare/instanțiere</i>	- se utilizează clasa corespunzătoare tipului obiectului <i>mock</i> ;	- se utilizează un obiect concret pentru care se creează o copie;
<i>Caracteristici</i>	- obiectul <i>mock</i> reprezintă un șablon al clasei respective nu este o instanță propri-zisă;	- obiectul <i>spy</i> reprezintă un obiect similar cu instanța din care a fost creat și poate fi folosit în manieră similară;
<i>Apelul metodelor clasei</i>	- prin metode specifice (when(...).thenReturn(...)) se poate seta un comportament pentru gestionarea interacțiunilor ulterioare; - nu există alte implicații asupra altor obiecte;	- se apelează implementările reale ale metodelor și au ca efect modificarea stării obiectului <i>spy</i> ;
<i>Obiectiv</i>	- urmărirea interacțiunilor cu obiectul <i>mock</i> creat;	- urmărirea interacțiunilor cu obiectul <i>spy</i> creat;

8. Stubs vs. Mocks

Aspect analizat	Stubs	Mocks
Șablon de utilizare	<ol style="list-style-type: none"> Setup – pregătirea obiectului testat și a stub-urilor folosite Exercise – rularea/execuția funcționalității Verificare – verificarea stării obiectului testat prin metode <code>assert</code> Teardown – eliberarea resurselor folosite 	<ol style="list-style-type: none"> Setup – pregătirea obiectului testat (software under test, SUT) Set expectations – stabilirea comportamentului pentru obiectelor mock utilizate de obiectul testat Exercise – rularea/execuția funcționalității Verify expectations – verificarea apelurilor adecvate pentru obiectul mock Verificare – verificarea stării obiectului testat prin metode <code>assert</code> Teardown – eliberarea resurselor folosite
Tip de testare	Evaluează starea obiectelor (<i>state-based testing</i>)	Evaluează comportamentul obiectelor (<i>interaction-based testing sau behavioural-based testing</i>)
Obiectiv	What is the result?	What is the result? How the result has been achieved?
Descriere	Un stub furnizează datele de test pentru SUT. Testul verifică prin metode <code>assert</code> asupra SUT dacă se obțin rezultatele așteptate, utilizând datele furnizate de stub.	Un mock așteaptă să fie apelat de către SUT. Există posibilitatea un obiect mock să pună la dispoziție mai multe metode care să fie utilizate de SUT. Verificarea asigură că obiectul mock a fost utilizat în manieră corectă. Astfel, testul este considerat <i>passed</i> dacă SUT interacționează corect cu obiectul mock.
	Ambele au scopul de a elimina la testare dependențele cu SUT ==> testarea devine mai simplă, focusul testării rămâne asupra SUT.	
Rezultatul testării	Utilizarea stub-urilor nu duce la eșuarea unui test. Metoda <code>assert</code> are ca focus obiectul testat (SUT).	Testul verifică prin intermediul obiectului mock dacă testul a eșuat.
		

Concluzie: La nivelul testării unitare orice obiect dependență creat este un *obiect fake*. Dacă are loc verificarea apelurilor asupra acestuia, atunci avem un *obiect mock*; altfel avem doar un *obiect stub*.

9. Resurse bibliografice

1. Martin Fowler, Mocks aren't stubs, <https://martinfowler.com/articles/mocksArentStubs.html>
2. Mockito vs. others mocking frameworks, <https://www.baeldung.com/mockito-vs-easymock-vs-jmockit>
3. Test Doubles, <https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da>
4. The art of unit testing, <https://www.manning.com/books/the-art-of-unit-testing-second-edition>
5. Unit tests with Mockito, <http://www.vogella.com/tutorials/Mockito/article.html>
6. Mockito tutorial, <https://javacodehouse.com/blog/mockito-tutorial/>
7. Mockito tutorial, <https://www.journaldev.com/21816/mockito-tutorial#mockito-maven-dependencies>
8. Injectarea dependențelor în Mockito, <https://www.journaldev.com/21887/mockito-injectmocks-mocks-dependency-injection>
9. Mockito Spies, <https://www.baeldung.com/mockito-spy>