Introduction to Mockito

- The basics: mock objects, stubbing, behavior verification, spying
- Tests with JUnit + Mockito
- Refactoring SUT code for dependency injection (mock object injection)

Ciências ULisboa

slides: Eduardo Marques, Vasco Vasconcelos, Francisco Martins, João Neto

- Mockito is a mocking framework for Java.
  - http://mockito.org
  - Other mocking frameworks: JMock, EasyMock, PowerMock, …
- Basic capabilities:
  - transparent use of mocks in place of concrete implementations
  - method stubbing
  - verification of interaction
  - spying on real objects
  - Online documentation at

    site.mockito.org/mockito/docs/current/org/mockito/Mockito.html

Mocking

- Mocking is the act of removing external dependencies from a unit test in order to create a controlled environment around it.
- Typically, we mock all the other classes that interact with the class we want to test.
- Common targets for mocking are:
  - Database connections,
  - Web services,
  - Classes that are slow,
  - Classes with side effects, and
  - Classes with non-deterministic behavior.

Mocking

- A **dummy** is just a placeholder object in order to run the test. It does not have behavior nor is called by the test.
- A **stub** is a fake class that comes with preprogrammed return values. It's injected into the class under test to give you absolute control over what's being tested as input. A typical stub is a database connection that allows you to mimic sets of scenarios without having a real database.
- A **mock** is a fake class that can be examined, after the test is finished, for its interactions with the class under test.
  - E.g., you can ask it whether a method was called or how many times it was called. Typical mocks are classes with side effects that need to be examined.
- A **spy** is based on a real object with original methods that do real things. Can be used like a stub to change return values of selected methods. Can also be used like a mock to describe interactions.

Mocking

- Dummies and stubs are used to prepare the environment for testing. They are not used for verification.
  - A dummy is employed to be passed as a value (e.g. as a parameter of a direct method call)
  - A stub passes some data to the SUT, substituting for one of its DOCs.
- The purpose of spies and mocks is to verify the correctness of the communication between the SUT and DOCs.

SUT – System Under Test
DOC – Depended On Component, any entity required by SUT to fulfill its duties

Mockito - stubbing method calls

```
import static org.mockito.Mockito.*;
...

List<String> mockedList = (List<String>)Mockito.mock(List.class);

// stub calls to size() and get()
when(mockedList.size()).thenReturn(3);
when(mockedList.get(0)).thenReturn("String at pos 0");
when(mockedList.get(1)).thenReturn("String at pos 1");
when(mockedList.get(2)).thenReturn("String at pos 2");
when(mockedList.get(3)).thenThrow(new IndexOutOfBoundsException());

// Interact with the mock object and observe
// the stubbed behavior being executed
for (int i=0; i <= mockedList.size(); i++) {
  System.out.println(mockedList.get(i));
}
```

```
String at pos 0
String at pos 1
String at pos 2
Exception in thread "main" java.lang.IndexOutOfBoundsException
    at vvs.basic_mockito.Stubbing.main(Stubbing.java:22)
```

Mockito - verifying interactions

```java
List<String> mockedList = (List<String>) Mockito.mock(LinkedList.class);

// Interact with the mock object
mockedList.set(0,"XPTO");
mockedList.set(1,"XPTO");
mockedList.clear();

// Verify interactions
verify(mockedList).set(0, "XPTO");
verify(mockedList).set(1, "XPTO");
verify(mockedList).clear();
verify(mockedList).size(); // WILL FAIL
```

```
Exception in thread "main" Wanted but not invoked:
linkedList.size();
-> at vvs.basic_mockito.VerifyInteractions.main(VerifyInteractions.java:25)

However, there were other interactions with this mock:
-> at vvs.basic_mockito.VerifyInteractions.main(VerifyInteractions.java:17)
-> at vvs.basic_mockito.VerifyInteractions.main(VerifyInteractions.java:18)
-> at vvs.basic_mockito.VerifyInteractions.main(VerifyInteractions.java:19)
   at vvs.basic_mockito.VerifyInteractions.main(VerifyInteractions.java:25)
```

Mockito - spying on real objects

```
// "Real object"
LinkedList<String> real = new LinkedList<>();
// Spy
LinkedList<String> spy =  (LinkedList<String>) spy(real);

// Note: it will call real methods from LinkedList
// but over a copy of the original real object passed to spy method
spy.add("A"); // calls real method
spy.add("B"); // calls real method
System.out.println(spy.size());
System.out.println(spy.get(0) + " " + spy.get(1));

// We can then verify interactions
verify(spy).add("A");
verify(spy).add("B");
verify(spy).size();
verify(spy).get(0);
verify(spy).get(1);
verify(spy).get(2); // WILL FAIL
}
```

```
2
A B
Exception in thread "main" Wanted but not invoked:
linkedList.get(2);
-> at vvs.basic_mockito.Spying.main(Spying.java:33)
```

Mockito limitations

- Classes that cannot be mocked:
  - Final classes.
  - Anonymous classes.
- Enumerations cannot be mocked.
- Methods that cannot be stubbed:
  - `final` methods
  - `static` methods
  - `private` methods
  - `hashCode`() and `equals`()
- Code may be refactored for testability to handle some of these (e.g., remove final modifier from classes or methods).

Example: line counter

```
public class LineCounter {

  public static int count(DataInput in, boolean ignoreBlankLines)
                              throws IOException {
    int count = 0;
    String line;
    while ((line = in.readLine()) != null)
      if (line.length() > 0 || !ignoreBlankLines) {
        count ++;

    return count;
  }
}
```

- We will use a mock `DataInput` object to test `LineCounter.count()`.

A test for `LineCounter.count()`

```
@Test
public final void testCountBlanksConsidered() throws IOException {
  // Create mock object
  DataInput mock = mock(DataInput.class);

  // Stub behavior: make sure readLine() returns certain values in succession.
  when(mock.readLine())
  .thenReturn("line 1")
  .thenReturn("") // line 2 will be blank
  .thenReturn("line 3")
  .thenReturn(null);

  int lines = count(mock, false)

  assertEquals(3, lines, "line count not ignoring blanks failed");

  // You can also verify the interaction with the mock object:
  // below we verify readLine() has been called 4 times.
  // Change 4 to another value to see what happens.
  verify(mock, times(4)).readLine();
}
```

create mock object

stub readLine() calls

call method under test

verify return value

verify interactions with the mock object

see more test examples in the provided Eclipse project

Example: mocking an interface

```
public interface Car {
  boolean needsFuel();
  double  getEngineTemperature();
  void    driveTo(String destination);
}
```

- Model the behaviour of this interface into a test using mocking

Example: mocking an interface

```
class CarMockitoTest {

  // Mocking a car
  private Car myFerrari = mock(Car.class);

  @Test
  public void CarIsACarTest() {
    assertTrue(myFerrari instanceof Car);
  }

  @Test
  public void defaultBehaviourOfTest() {
    // by default, mock objects return zeros and falses
    //  from their attributes
    assertFalse(myFerrari.needsFuel(), “mock should return false");
    assertEquals(0.0, myFerrari.getEngineTemperature(), 1.0e-3,
          "new test double should return 0.0 as double");
  }
```

Example: mocking an interface

```
@Test
public void stubbingTest() {

  // we tell myFerrari what to do when asked about the fuel
  when(myFerrari.needsFuel()).thenReturn(true);

  assertTrue(myFerrari.needsFuel(),
    "after instructed test double should return what we want");
}

@Test
public void stubbingExceptionTest() {
    when(myFerrari.needsFuel()).thenThrow(new RuntimeException());

    assertThrows(RuntimeException.class, () -> {
      myFerrari.needsFuel();
    });
}
```

Example: mocking an interface

```java
@Test
public void verificationTest() {
  myFerrari.driveTo("Sweet home Alabama");
  myFerrari.needsFuel();

  // check if these methods were executed
  verify(myFerrari).driveTo("Sweet home Alabama");
  verify(myFerrari).needsFuel();
}

@Test
public void verificationFailureTest() {
  myFerrari.needsFuel();

  verify(myFerrari,never()).getEngineTemperature();
}
```

Example: mocking an existing class

```
class ListTest {

  // mocking a List (the SUT already exists in the Java API)
  private List<String> mockedList = mock(List.class);

  @Test
  void verifyTest() {
    //using mock object
    mockedList.add("one");
    mockedList.clear();

    //verification
    verify(mockedList).add("one");
    verify(mockedList).clear();
  }
```

Example: mocking an existing class

```
@Test
void stubTest() {
  // Once stubbed, the method will always return a stubbed value,
  // regardless of how many times it is called.
  when(mockedList.get(0)).thenReturn("first");
  when(mockedList.get(1)).thenThrow(new RuntimeException());

  assertEquals("first", mockedList.get(0));

  assertThrows(RuntimeException.class, () -> {
    mockedList.get(1);
  });

  // returns "null" because get(999) was not stubbed
  assertNull(mockedList.get(999));
}
```

Example: mocking an existing class

```
@Test
void argumentMatcherTest() {
  // stubbing using built-in anyInt() argument matcher
  // (allows flexible verification or stubbing)
  when(mockedList.get(anyInt())).thenReturn("element");

  assertEquals("element", mockedList.get(999)); // returns "element"

  // you can also verify using an argument matcher
  verify(mockedList).get(anyInt());

  mockedList.add("hello world");

  verify(mockedList).add(eq("hello world"));

  // argument matchers can also be written as Java 8 Lambdas
  verify(mockedList).add(argThat(str -> str.length() > 5));

}
```

```java
@Test
void invocationsTest() {
  mockedList.add("once");
  mockedList.add("twice");          mockedList.add("twice");
  mockedList.add("three times");    mockedList.add("three times");
  mockedList.add("three times");

  // following two verifications work exactly the same
  verify(mockedList).add("once");
  verify(mockedList, times(1)).add("once");

  // exact number of invocations verification
  verify(mockedList, times(2)).add("twice");
  verify(mockedList, times(3)).add("three times");

  // verification using never(). never() is an alias to times(0)
  verify(mockedList, never()).add("never happened");

  // verification using atLeast()/atMost()
  verify(mockedList, atLeastOnce()).add("three times");
  verify(mockedList, atLeast(2)   ).add("three times");
  verify(mockedList, atMost(5)    ).add("three times");
}
```

```java
@Test
void spyRealObjsTest() {
  List<String> list = new LinkedList<>();
  List<String> spy = spy(list);

  //optionally, you can stub out some methods:
  when(spy.size()).thenReturn(100);

  //using the spy calls *real* methods
  spy.add("one");
  spy.add("two");

  //prints "one" - the first element of a list
  assertEquals("one", spy.get(0));

  //size() method was stubbed - 100 is printed
  assertEquals(100, spy.size());

  //optionally, you can verify
  verify(spy).add("one");
  verify(spy).add("two");
}
```

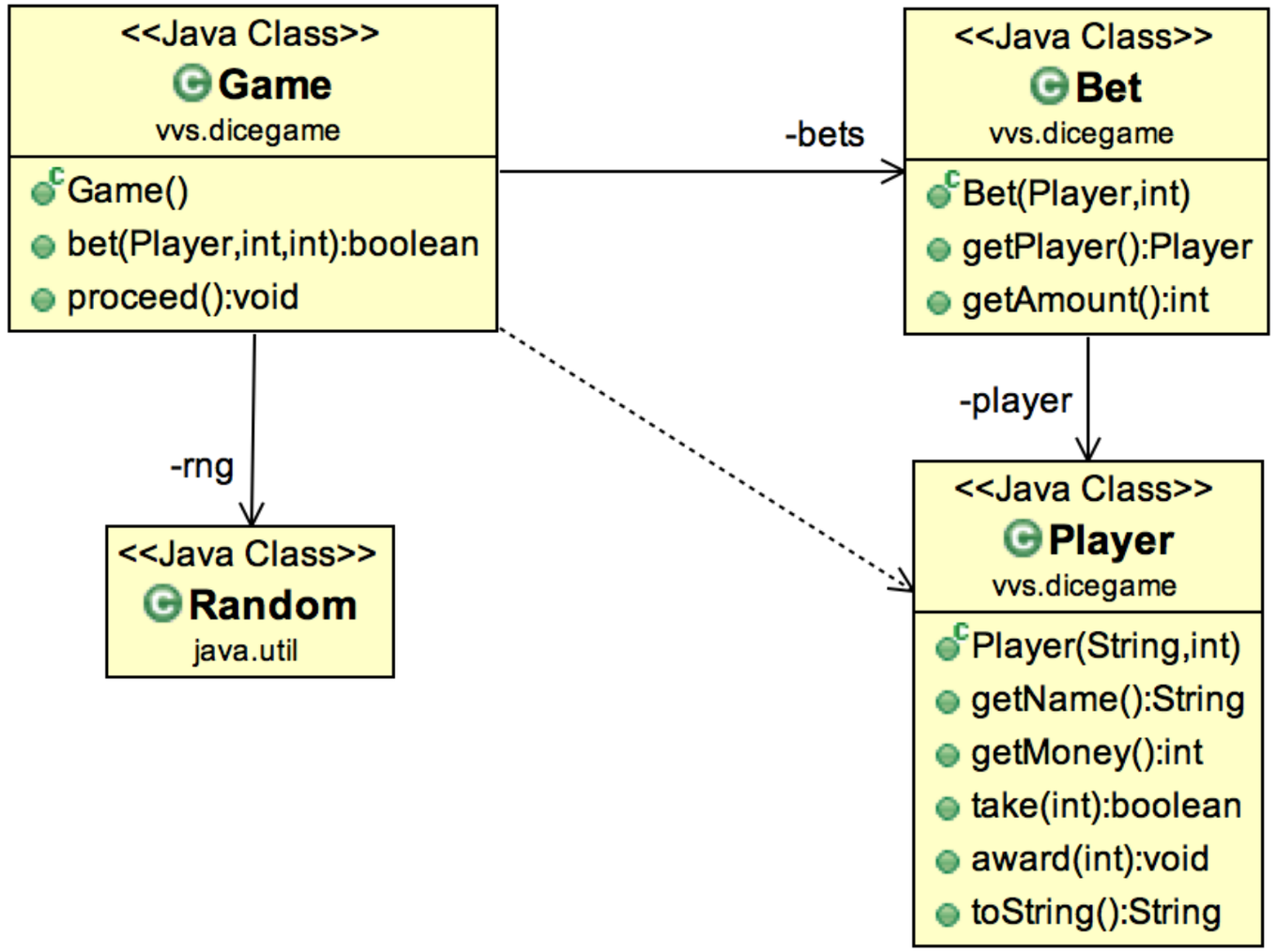
If we have a real object, we can use it on a spy, and stub only some of its methods (if needed)

Example: a simple dice game

Aim 1: verifying `Game` independently from the internal logic `Player` and `Random`.

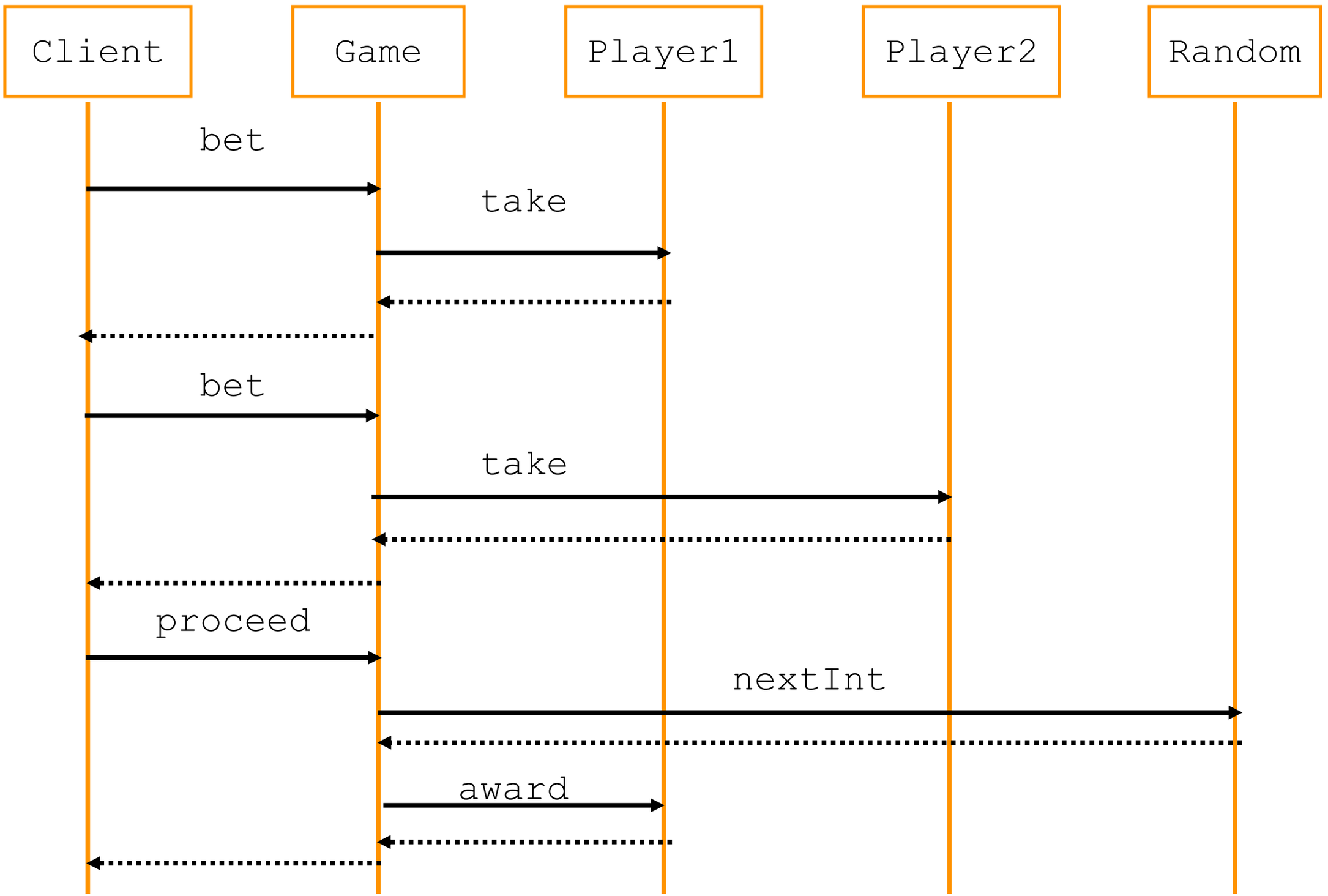
We will use mock objects for `Player` and `Random`.

Note: `Bet` is a "value object" (simply holds values; without any other internal logic).




```
Player state: Alberto/100 ; Manuel/100
Bet of 30 euros placed by Alberto on 5!
Bet of 50 euros placed by Manuel on 6!
Total bets: 80 euros
Dice has rolled: 6 !!
The winner is Manuel!
Player state: Alberto/70 ; Manuel/130
```

Sample interaction for 2 players



We need some refactoring first

```
private final Random rng;
private final Bet[] bets;
public Game() {
  // Not a test friendly constructor
  // as it instantiates the object directly.
  // The use of Random also raises problems
  // to test repeatability / reproducibility
  // (even with a fixed seed):
   rng = new Random();
   bets = new Bet[6];
}
// Let's add a test-friendly constructor.
// It allows for dependency injection.
Game(Random r) {
   rng = r;
   bets = new Bet[6];
}
```

Test setup

```java
public class GameTest {
  // Mock objects used by all tests
  @Mock Random rng;
  @Mock Player player1;
  @Mock Player player2;
  // Mockito object to test call order
  InOrder callOrder;
  // Game object
  Game game;

  // Setup method executed before each test
  @BeforeEach
  public void setup() {
    // Instantiate all fields annotated with @Mock
    MockitoAnnotations.initMocks(this);
    // Shared stubbing for all tests
    when(player1.getName()).thenReturn(NAME_P1);
    when(player1.take(BET_VALUE_P1)).thenReturn(true);

    when(player2.getName()).thenReturn(NAME_P2);
    when(player2.take(BET_VALUE_P2).thenReturn(true);

    when(rng.nextInt(6)).thenReturn(WINNING_BET);
    callOrder = inOrder(rng, player1, player2);
    game = new Game(rng);
  }
  ...
```

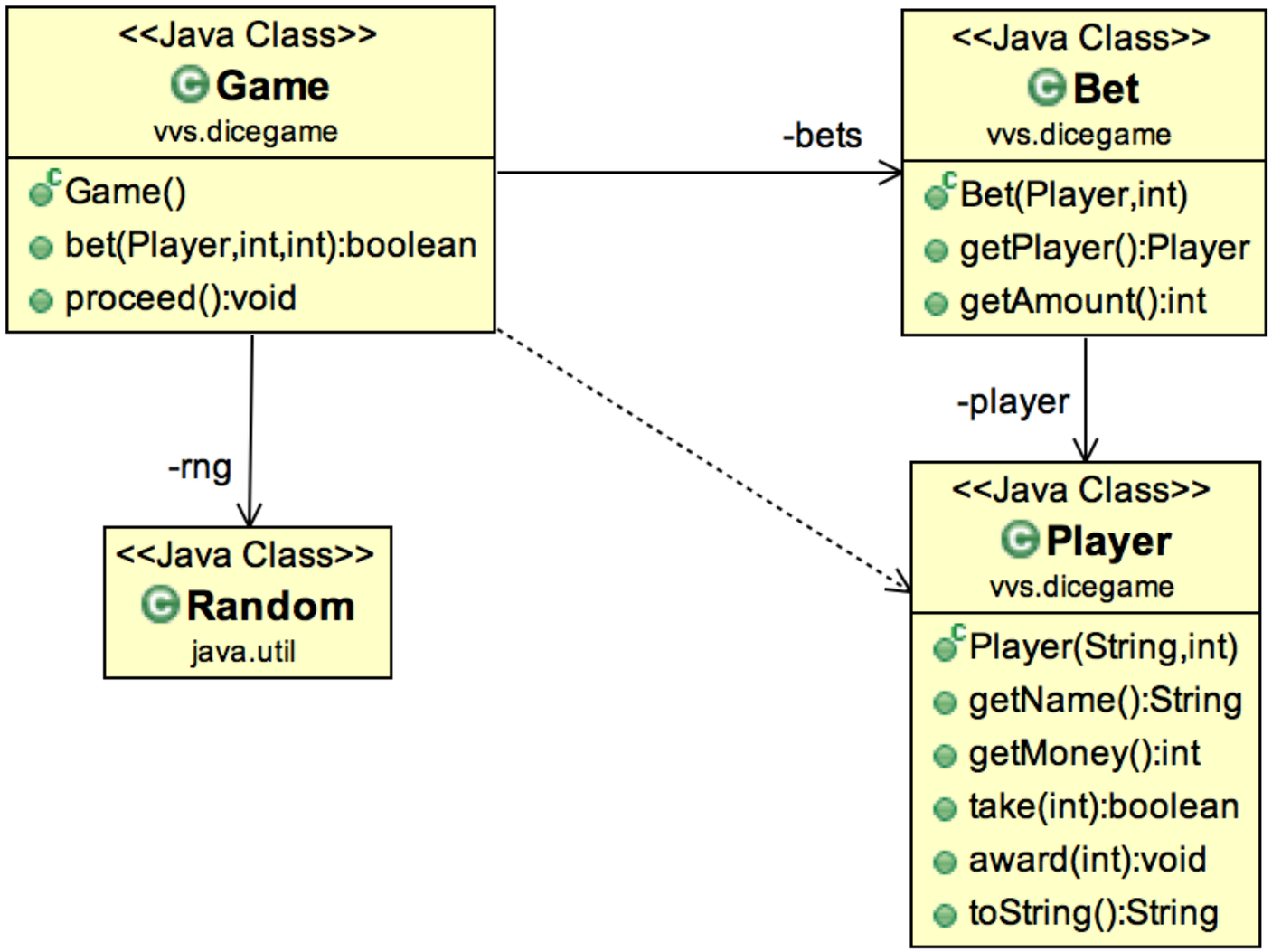
```java
@Test
public final void testGamePlayer1Wins() {
 testGameWithBets(player1, WINNING_BET, LOSING_BET_2);
}
@Test
public final void testGamePlayer2Wins() {
 testGameWithBets(player2, LOSING_BET_1, WINNING_BET);
}
@Test
public final void testGameNoWinners() {
  testGameWithBets(null, LOSING_BET_1, LOSING_BET_2);
}
private void testGameWithBets(Player winner, int p1Bet, int p2Bet) {
  game.bet(player1, p1Bet, BET_VALUE_P1);
  game.bet(player2, p2Bet, BET_VALUE_P2);
  game.proceed();

  callOrder.verify(player1).take(BET_VALUE_P1);
  callOrder.verify(player2).take(BET_VALUE_P2);
  callOrder.verify(rng).nextInt(6);

  if (winner != null)
    callOrder.verify(winner).award(BET_VALUE_P1+BET_VALUE_P2);

  callOrder.verify(rng,never()).nextInt();
  callOrder.verify(player1,never()).take(anyInt());
  callOrder.verify(player2,never()).take(anyInt());
  callOrder.verify(player1,never()).award(anyInt());
  callOrder.verify(player2,never()).award(anyInt());
}
```

Example 2: using spies



Aim 2: verifying `Game` interactions with real `Player` objects.
We will use a mock object for `Random` and spies for `Player`.

```
Player state: Alberto/100 ; Manuel/100
Bet of 30 euros placed by Alberto on 5!
Bet of 50 euros placed by Manuel on 6!
Total bets: 80 euros
Dice has rolled: 6 !!
The winner is Manuel!
Player state: Alberto/70 ; Manuel/130
```

Test setup

```
public class GameTest {
    // Mock objects used by all tests
    @Mock Random  rng;
    @Spy Player  player1 = new Player(NAME_P1, INITIAL_MONEY);
    @Spy Player  player2 = new Player(NAME_P2, INITIAL_MONEY);
    // Mockito object to test call order
    InOrder callOrder;
    // Game object
    Game    game;

    // Setup method executed before each test
    @BeforeEach
    public void setup() {
      // Instantiate all fields annotated with @Mock
      MockitoAnnotations.initMocks(this);
      // Shared stubbing for all tests
      when(rng.nextInt(6)).thenReturn(WINNING_BET - 1);
      callOrder = inOrder(rng, player1, player2);
      game = new Game(rng); // inject dependency
    }
    ...
```

Verifying the state of spy objects

```
// Parameterized test utility method
private void testGameWithBets (Player winner, int p1Bet, int p2Bet,
                               int expMoney1, int expMoney2) {
  game.bet(player1, p1Bet, BET_VALUE_P1);
  game.bet(player2, p2Bet, BET_VALUE_P2);
  game.proceed();

  // Verify spy objects state
  assertEquals("player 1 - money", expMoney1, player1.getMoney());
  assertEquals("player 2 - money", expMoney2, player2.getMoney());

  // Verify interactions as in the first example
  …
}
```