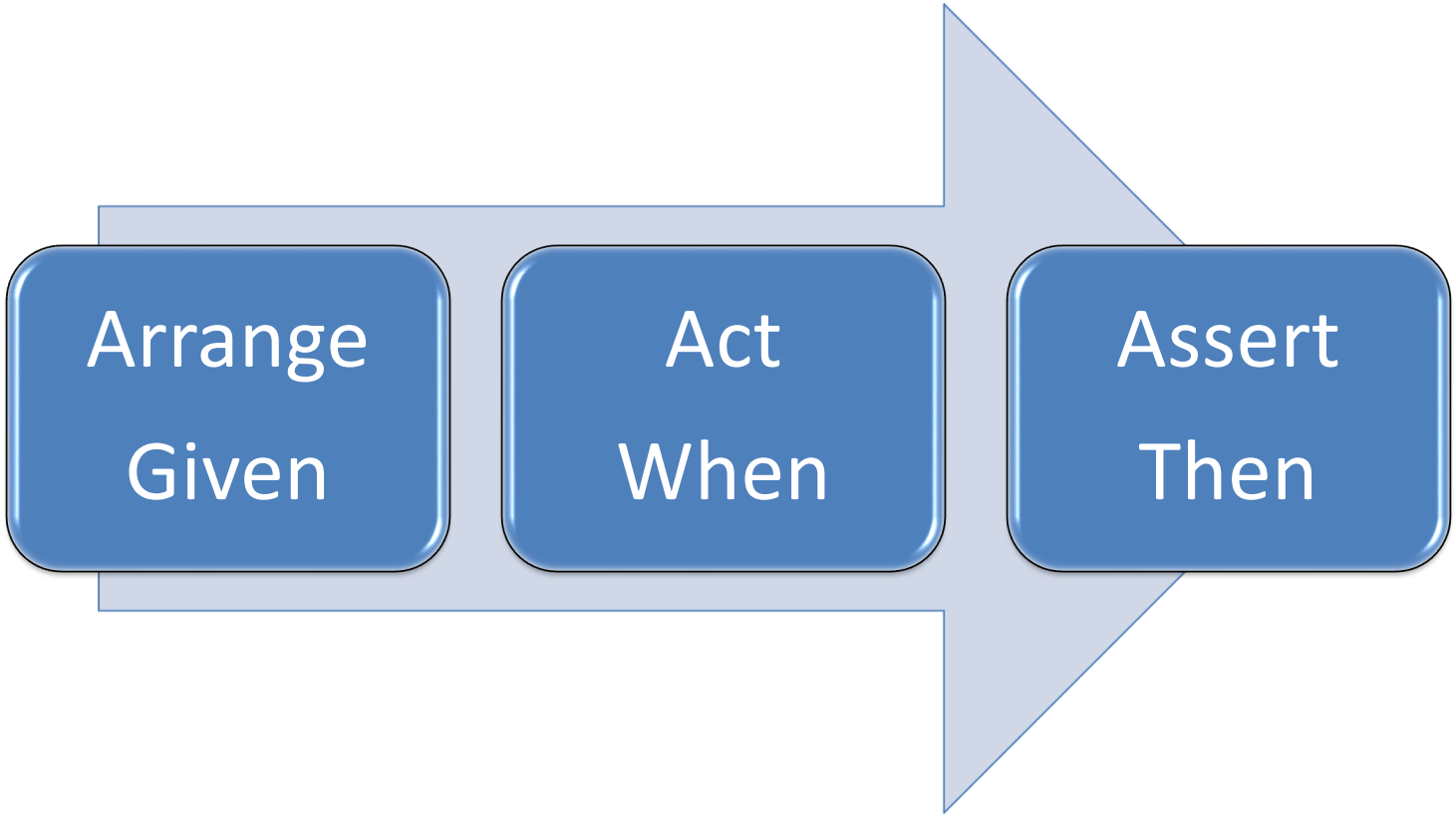
Test Doubles - Mocking

Unit Testing and Isolation

Structure of a test case



Given-When-Then test case style

- Similar to Arrange-Act-Assert
- It is a style of representing test cases
- Test cases broken down into three sections
  - ***Given*** – specified the initial state of the system before the desired behavior is exercised
  - ***When*** – specifies the behavior to exercise
  - ***Then*** – describes the changes the tester expects after specified behavior is invoked
- Leads to more resilient test cases
  - Based on the requirements vs implementation details

Structure of a test case

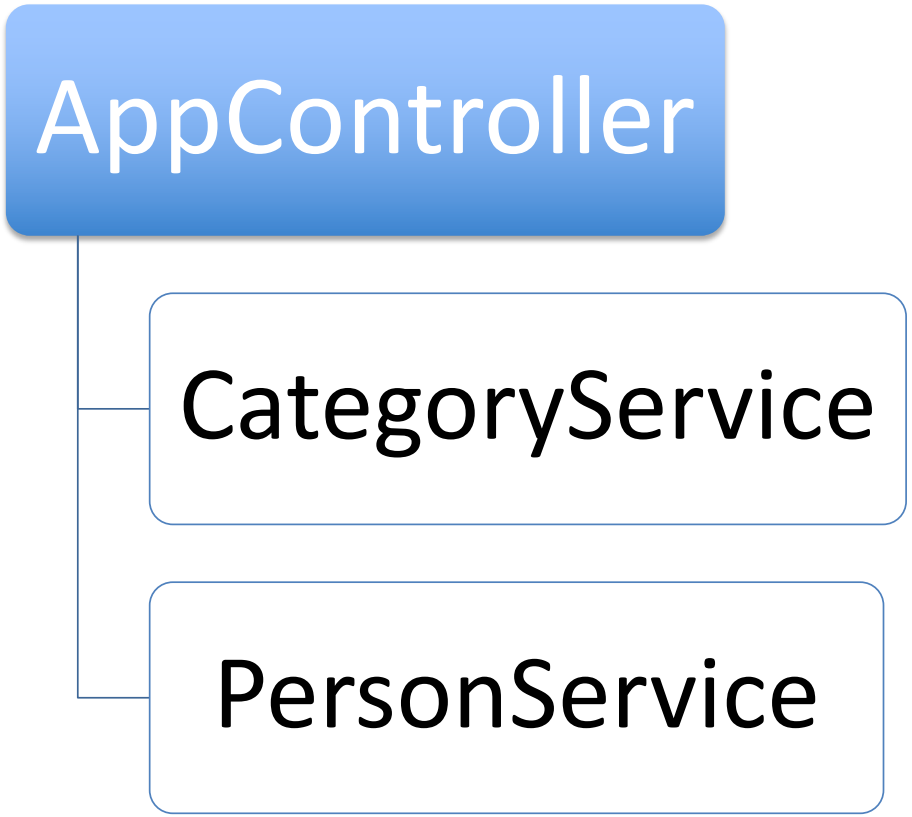
- Also leads to more readable tests

```
@Test
public void givenPersonWithEmptyNameWhenValidateThenIsInvalid() {
  // Given a person has an empty name
  Person person = new Person();
  person.setName("");

  // When I validate the person
  person.validate();

  // Then the person is invalid
  assertFalse(person.isValid());
}
```

Testing a class



Testing a class

```
@Test
public void appControllerTest() {
  // Arrange / Given …
  PersonService pService =  new PersonService(searchService, personRepo);
  CategoryService cService = new CategoryService(categoryRepo);
  AppController appController = new AppController(cService, pService);

  // Act / When …
  // ... run the tested action ...

  // Assert / Then …
  // ... perform the needed assertions ...
}
```

Testing a class

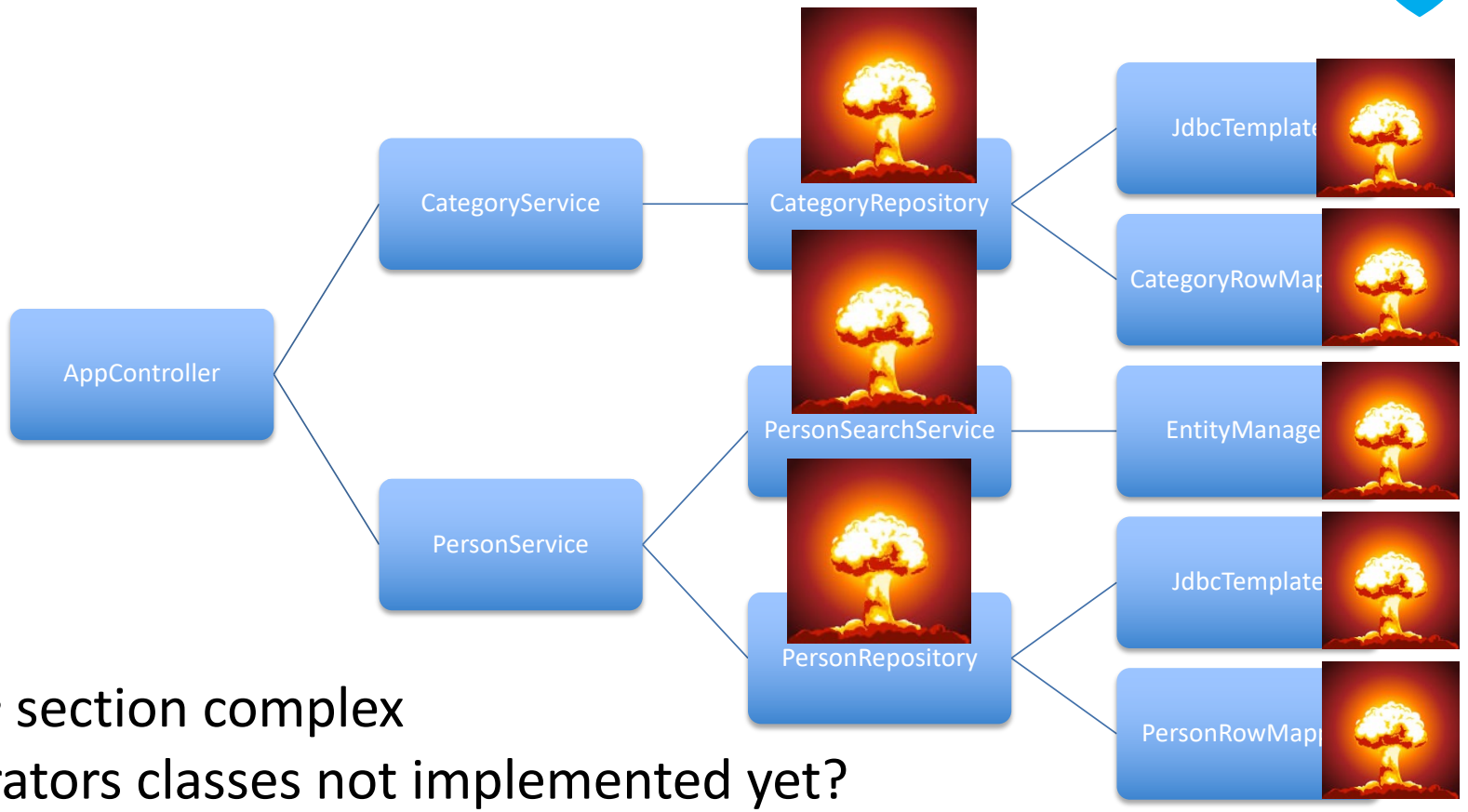
```
@Test
public void appControllerTest() {
  // Arrange / Given
  PersonService pService =  new PersonService(searchService, personRepo);
  CategoryService cService = new CategoryService(categoryRepo);
  AppController appController = new AppController(cService, pService);

  // Act / When …
  // ... run the tested action ...

  // Assert / Then …
  // ... perform the needed assertions ...
}
```

What the???

Testing a class - Collaborators



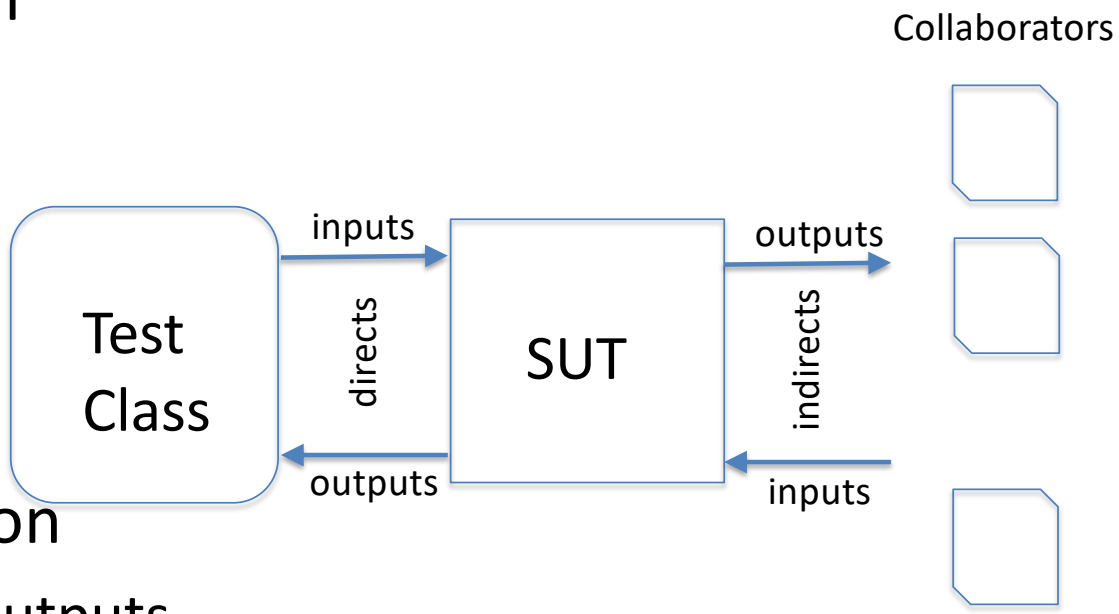
- Can make *Arrange* section complex
- And if the collaborators classes not implemented yet?
- And how to test in isolation?

Solution: Test Doubles

- Test double
  - A pretend object that substitutes a real object in the context of testing
- Frequently used technique to:
  - Make setup easier
  - Isolate code under test
  - Make test execute faster
  - Make execution deterministic
  - Simulate special conditions
  - Gain access to hidden information

Unit Testing

- Test in isolation



- State verification
  - Check direct outputs
- Behavior verification
  - Check indirect outputs

Test doubles come in different flavors

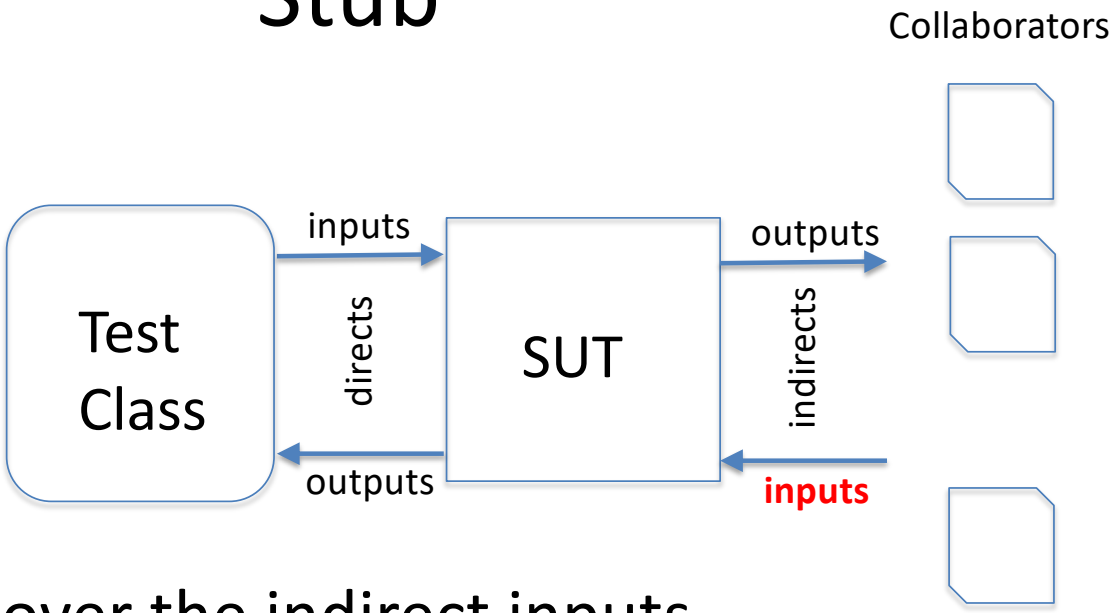
- ***Dummy*** objects
  - are passed around but never actually used
- ***Fake*** objects
  - actually have working implementations, but usually take some shortcut
- ***Stubs***
  - provide canned answers to calls made during the test.
- ***Spies***
  - are stubs that also record some information based on how they were called
- ***Mocks***
  - are pre-programmed with expectations which form a specification of the calls they are expected to receive.

Dummy

- Objects that do nothing
- And return as close to “nothing”

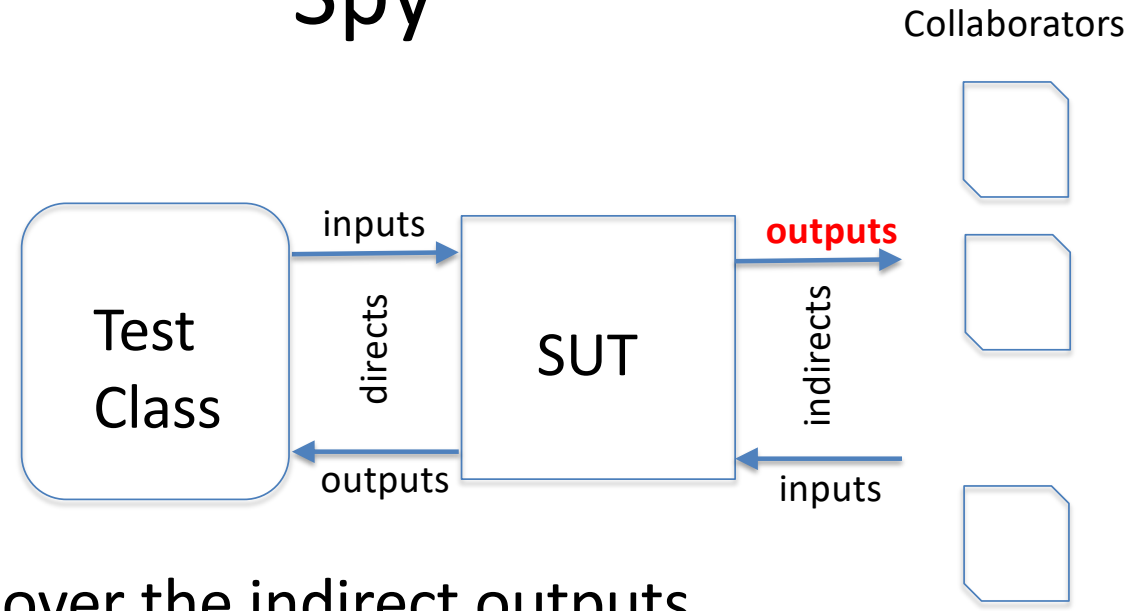
- When?
  - Fill methods arguments
  - Not needed in test

Stub



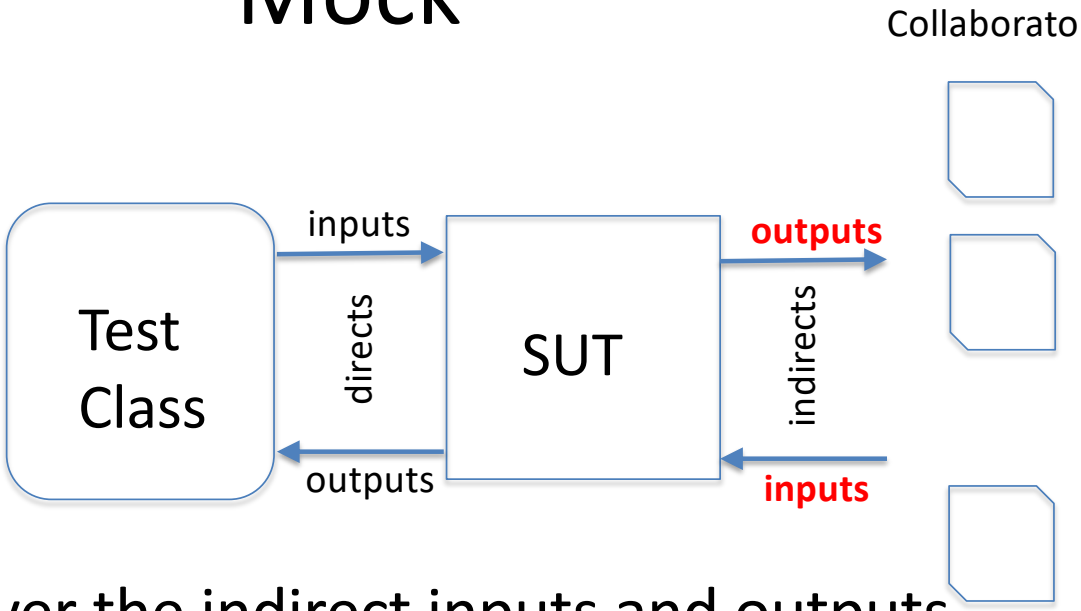
- Control over the indirect inputs
  - Objects do nothing; do not remember invocations
  - And return something useful
- Test case: Guide the test through a specific execution path

Spy



- Control over the indirect outputs
  - Spies remember calls, observe silently
    - May provide indirect outputs
  - Don't have any expectations
- Test case: need to check the interaction with the spy is equal to the expected

Mock



- Control over the indirect inputs and outputs
  - Remember calls
  - And return something useful
- Test case: Automatically check that interaction with mock is equal to expected
  - Can be strict or lenient

Manual Mocking - Example

```
public class TodoBusinessImpl {
   private TodoService todoService;

   TodoBusinessImpl(TodoService todoService) {
       this.todoService = todoService;
   }

   public List<String> retrieveTodos(String user, String topic) {
      List<String> filteredTodos = new ArrayList<String>();
      List<String> allTodos = todoService.retrieveTodos(user);
      for (String todo : allTodos) {
         if (todo.contains(topic))
            filteredTodos.add(todo);
      }
      return filteredTodos;
   }
}
```

```
public interface TodoService {
   List<String>  retrieveTodos(String user);
    …
}
```

How to test this code Independently from TodoService?

Manual Mocking – Test Case

```
public class TodoBusinessImplStubTest {
   @Test public void testUsingAStub() {
       TodoService todoService;
        todoService = new TodoServiceStub();

       TodoBusinessImpl todoBusinessImpl = new TodoBusinessImpl(todoService);
       List<String> todos = todoBusinessImpl.retrieveTodo("Ranga", "Spring");
       assertEquals(2, todos.size());
       // more asserts
   }
}
```

```
public class TodoServiceStub implements TodoService {
   public List<String> retrieveTodos(String user) {
      return List.of("Learn Spring MVC", "Learn Spring","Learn to Dance");
   }
}
```

Manual mocking

- Disadvantages
  - Time consuming
  - May pollute code base
  - More maintenance (e.g., if interface changes)
  - Much effort involved to add more sophisticated features (e.g., parameter checking, invocation checking, invocation order)

Mocking framework

- Apply a mock library to remove the grunt work
- There are a several *mock* frameworks:
  - jmock, mockito, jmockit, …
  - Supports the creation of several types of *Test Double*
- JMockit
  - Open source testing framework for Java
    - http://jmockit.github.io
    - Not updated since 2020
- **Mockito**
  - https://site.mockito.org

Mockito – Creating a test double

- How to create a test double?
  - Invoke **mock(Class cl**) to mock a class or interface
    - Returns an instance that pretends to be an instance of class (or interface) represented by **cl**
- *All* methods of the test double will be mocked for the duration of the test
  - Original implementation code won't be executed
  - Each call is redirected to Mockito
  - Behavior of mocked code is specified (implicitly or explicitly) in the test

Default behavior of a test double

- *Do nothing*

- And if methods are not void:
  - boolean
    - return **false**
  - numeric (primitive type/wrapper primitive type)
    - return **0**
  - object
    - *null*
    - Empty collection/map
      - For methods that return a Collection/Map

Mockito - Mocking a type (List)

```
import org.testng.annotations.Test;
import static org.testng.Assert.*;
import static org.mockito.Mockito.*;
import java.util.List;

public class TestList {

  @Test public void testMock() {
    List<String> mockList = mock(List.class);

    mockList.add("AAAAAAAAAA");

    System.out.println("value of get: " + mockList.get(0));
    System.out.println("value of size: " + mockList.size());
    System.out.println("value of isEmpty: " + mockList.isEmpty());
  }
}
```

Result of running test case

```
Running TestList
value of get: null
value of size: 0
value of isEmpty: false
```

Mockito – Mixing real & mock objects

- You can mix real objects and mock objects in the same test case

```
@Test public void testMockAndReal() {
   ArrayList<String> mockList = mock(ArrayList.class);
   ArrayList<String> realList = new ArrayList<>();

   mockList.add("AAAAA");
   realList.add("AAAAA");

   System.out.println("mock: value of get: " + mockList.get(0));
   System.out.println("mock: value of size: " + mockList.size());

   System.out.println("real: value of get: " + realList.get(0));
   System.out.println("real: value of size: " + realList.size());
}
```

Result of running test case:

mock:value of get: null
mock: value of size: 0
real:value of get: AAAAA
real: value of size: 1

Example - Car class

```
public class  Car {
   public boolean needsFuel() { return true; }

   public double getEngineTemperature() {
     return -1.0;
   }
   public void driveTo(String dest) { }
}
```

Create a mock object

- Test doubles created by Mockito are just like normal objects
  - Can be used inside the test methods
  - Passed to code under test
  - Or simply go unused

```
import statements;
@Test
public class TestCreateMock {
  private Car myFerrari;

  public void testIfCarIsACar() {
    myFerrari = mock(Car.class);
    assertNotNull(myFerrari);
    assertTrue(myFerrari instanceof Car);
  }
}
```

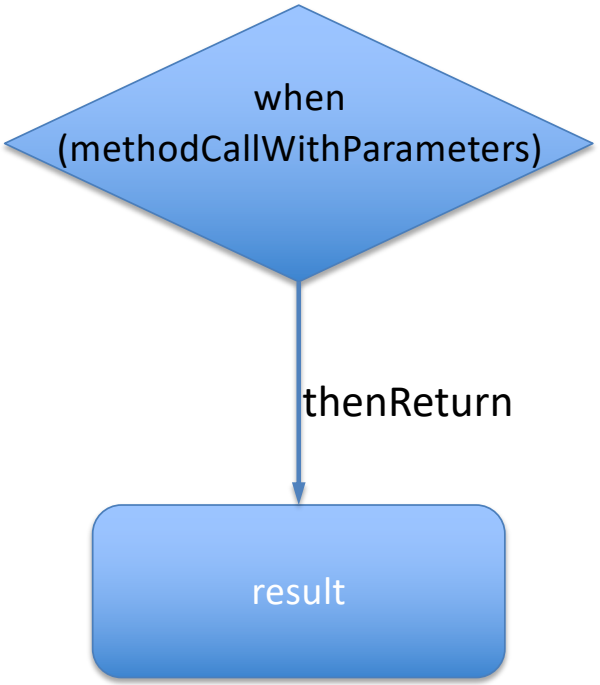
- *myFerrari* can become a dummy, stub, spy or mock

Programming a test double

- Advantage of having a test double is that we can program its behavior
- This should be as simple as possible
  - Mockito provides a default behavior for all methods of a test double
  - Simple interface for overriding default behavior
  - Test doubles are ***nice*** in Mockito

Programming a test double - 2

- Expected behavior of test double of a Car
  - *getEngineTemperature*() - should return **10.0**
  - and *needsFuel*() - should return **true**
- Cannot use default behavior of test double
- Behavior of mock is specified using



  - when(myFerrari.*getEngineTemperature*()).thenReturn(10.0);
  - when(myFerrari.*needsFuel*()).thenReturn(true);

Programming a test double - 3

Expected behavior of Car

- *getEngineTemperature*() - should return 10.0
- and *needsFuel*() - should return true

```
public class TestDesiredBehavior {

  private Car myFerrari;
  @Test public void testChangeDefaultBehavior() {
    myFerrari = mock(Car.class);
    assertFalse(myFerrari.needsFuel(), ”should return false");
    // programme expected behavior
    when(myFerrari.getEngineTemperature()).thenReturn(10.0);
    when(myFerrari.needsFuel()).thenReturn(true);
    // check expected behavior
    assertTrue(myFerrari.needsFuel(), ”should return true");
    assertEquals(myFerrari.getEngineTemperature(), 10.0,
                  "should return 10.0");
    assertTrue(myFerrari.needsFuel(), ”should return true");
  }
}
```

Mockito – Specify return value

- Two ways

1. **when***(invocationOnMock).**thenReturn**(value)*
   - **when***(invocationOnMock).***thenAnswer**(Answer<T>)
   - **when**(invocationOnMock).**thenCallRealMethod**()

2. **doReturn**(value).**when**(mock).invocation(args)
   - Similar to first when-thenReturn, however
     - Less readable
     - Not type safe
   - But in rare occasions is the right solution

Distinct behaviour for consecutive calls

- Chain several **thenReturn**(result) for the same invocation

```
when(myFerrari.needsFuel())
    .thenReturn(true)
    .thenReturn(false)
    .thenReturn(true);
```

- Or use overloaded **thenReturn**(T … args)

```
when(myFerrari.needsFuel())
    .thenReturn(true, false, true);
```

- **Warning**: Do not use multiple stubbing with the same arguments
  - Each stubbing overrides the previous one

```
when(myFerrari.needsFuel()).thenReturn(true);
when(myFerrari.needsFuel()).thenReturn(false);
```

equivalent

```
when(myFerrari.needsFuel())
    .thenReturn(false);
```

Example

```
public class TestDesiredBehavior {
   private Car myFerrari;

   @Test public void checkSeveralConsecutiveInvocations() {
       myFerrari = mock(Car.class);
       when(myFerrari.needsFuel()).thenReturn(true);
       when(myFerrari. getEngineTemperature())
         .thenReturn(10.0)
         .thenReturn(20.0);

       assertEquals(myFerrari.getEngineTemperature(), 10.0, “should return what we want");
       assertTrue(myFerrari.needsFuel(), ”should return what we want");
       assertEquals(myFerrari.getEngineTemperature(), 20.0, “should return what we want");
      assertEquals(myFerrari.getEngineTemperature(), 20.0, “should return what we want");
 }}
```

Matching of argument values

- What happens when mocked methods have arguments?
- Mockito verifies argument values using equals() method
  - when(mockList.get(0)).thenReturn(“abc”);
  - Also applies for *verify*
- Sometimes, it is desirable to have more flexibility
  - **any()** - Matches **anything**, including nulls
  - **any(Class<T> type**) - Matches any object of given type, excluding nulls
  - **anyString()** – Matches any **non-null** String
  - **anyInt()** – Matches any int or **non-null** Integer
  - **anyCollection()** – Matches any **non-null** Collection
  - isNull(), notNull()
  - eq(t1), same(t1), or(t1, t2), and(t1, t2), not(t)
    - when(mockList.get(or(1, 0))).thenReturn(“abc”);

Example

```
List<Integer> mockList1 = mock(List.class);
List<Integer> mockList2 = mock(List.class);

when(mockList1.add(any(Integer.class))).thenReturn(true);
when(mockList2.add(1)).thenReturn(true);

assertTrue(mockList1.add(1));
assertTrue(mockList2.add(1));
assertTrue(mockList1.add(2));
assertFalse(mockList2.add(2));
```

Throwing an exception

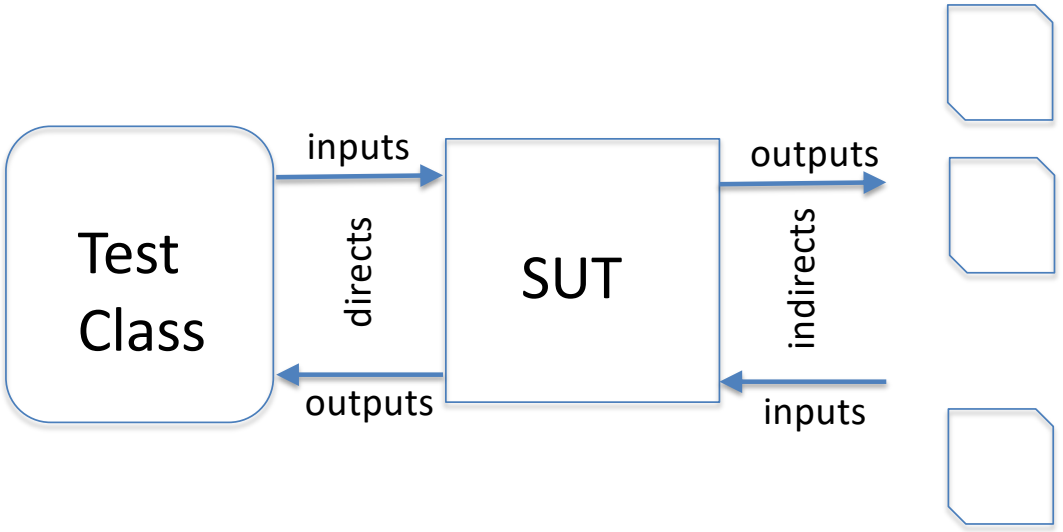
- When method invocation on test double should throw an exception use
  - *when(mock.method(args)).thenThrow(some exception);*
  - It is possible to chain **thenThrow** with **thenResult**
  - **myFerrari.needsFuel()** returns true, false, and throws an *IllegalStateException* exception

```
when(myFerrari.needsFuel())
    .thenReturn(true, false)
    .thenThrow(new IllegalStateException());
```

- For **void** methods use another approach:
  - **doThrow**(some exception).**when**(mock).someMethod();

What do we know?

- We know how to supply the indirect inputs
  - *when(…).thenReturn(…)*
  - *doReturn(…).when(…)*
- What is missing is how to **check the indirect outputs**



Verifying invocations on test doubles

- Mockito keeps track of all methods invoked on test doubles
- Can check that certain methods have been called on some test doubles during test execution
- Two overloaded **verify** methods
  - One accepts a test double and a VerifyMode
    - **verify(mock, VerificationMode).somemethod(args);**
    - times(int), atLeast(int), atMost(int), never(), only(), …
  - Other *accepts* test double and assumes VerifyMode equal to *times(1)*
    - **verify(mock).somemethod(args);**
  - If verification fails, then test case also fails

Verifications – Example

- Needs fuel and drive car to *Lisbon* and then to *Porto*

```
@Test
public void testVerTravelLisbonToPorto() {
  myFerrari = mock(Car.class);
  myFerrari.needsFuel();
  myFerrari.driveTo("Lisbon");
  myFerrari.driveTo("Porto");
  // verify invocations on test double
  verify(myFerrari).needsFuel();
  verify(myFerrari).driveTo("Lisbon");
  verify(myFerrari).driveTo("Porto");
}
```

- Invocation order is **not** relevant

Verification not met

```
@Test
 public void testFailedInvocationExpectation() {
      myFerrari = mock(Car.class);
      myFerrari.driveTo(“Home");

      verify(myFerrari). driveTo(“Alameda”);
}
```

- Error message:

  org.mockito.exceptions.verification.ArgumentsAreDifferent:
  Argument(s) are different! Wanted:
  car.driveTo("Alameda");
  -> at
  tvs.TestErrors.testVerificationFailureArguments(TestErrors.java:25)
  Actual invocations have different arguments:
  car.driveTo("Home");

- Would also fail if *driveTo(“Home”)* not invoked

Verifications – Example 2

- Drive car to *Porto* and then to *Lisbon*

```
@Test
public void testVerTravelPortoToLisbon() {
   myFerrari = mock(Car.class);
   myFerrari.driveTo("Porto");
   myFerrari.driveTo("Lisbon");
   // verify invocations on test double
   verify(myFerrari).driveTo("Lisbon");
   verify(myFerrari).driveTo("Porto");
}
```

- What happens in this case?
- Test case **passes!**
  - Invocation order is **not** relevant

Verification in order

- Sometimes, the invocation order is important
- Must use *org.Mockito.InOrder* class
  - Initialize InOrder with test doubles relevant for order
    - InOrder inOrder = inOrder(myMock, …);
  - The verify method must be invoked on InOrder object
- Verification in order is flexible
  - You do not have to check all interactions

Verification in order – Example

```
@Test public void testVerTravelPortoToLisbon2() {
  myFerrari = mock(Car.class);
  myFerrari.driveTo(“Porto”);
  myFerrari.driveTo(“Lisbon”);
  InOrder inOrder = inOrder(myFerrari);

  // verify invocations on test double

  inOrder.verify(myFerrari).driveTo(“Lisbon”);
  inOrder.verify(myFerrari).driveTo(“Porto”);
}
```

```
Verification in order failure
Wanted but not invoked:
car.driveTo("Porto");
-> at tvs.TestErrors.testVerTravelPortoToLisbon2(TestErrors.java:38)
Wanted anywhere AFTER following interaction:
car.driveTo("Lisbon");
-> at tvs.TestErrors.testVerTravelPortoToLisbon2(TestErrors.java:32)
```

Verifying invocation did not occur

- Two ways
- First approach: Use *verify()* with *never()*
  - verify(myFerrari, never()).needsFuel();
- Second approach: Use *verifyNoMoreInteractions()*
  - Checks that are not other invocations besides those previously checked with *verify()*

```
mockedList.add("one");
mockedList.add("two");

verify(mockedList).add("one");
verifyNoMoreInteractions(mockedList);
```

verification fails

  - Leads to **over specified** and **less maintainable** tests

Example

```
public class Messenger {
  private TemplateEngine engine;
  private MailServer mailServer;

  public Messenger(MailServer mailServer,
                   TemplateEngine templateEngine) {
    this.mailServer = mailServer;
    this.engine = templateEngine;
  }

  public void sendMessage(Client cl, Template temp) {
    String content = engine.prepareMessage(temp, cl);
    mailServer.send(cl.getEmail(), content);
  }
}
```

- Testing *sendMessage* with state behavior is hard since it returns no value and global state of system is not modified

Example – Test in Isolation

- Goals of this example:
  - Test Messenger in **isolation**
  - Check that it does what is supposed
    - The send() message on mail server is invoked with the right arguments
  - Use the several types of test doubles

Test Double – Dummy Object

```
public void sendMessage(Client cl, Template temp)  {
  String content = engine.prepareMessage(temp, cl);
  mailServer.send(client.getEmail(), content);
}
```

- Dummy Object
  - Template **temp**
- How to create a dummy object with Mockito?
- Just invoke the **mock()** method
  - Template template = mock(Template.class);

Test Double – Stub

```
public void sendMessage(Client cl, Template temp)  {
  String content = engine.prepareMessage(temp, cl);
  mailServer.send(client.getEmail(), content);
}
```

- Stub
  - TemplateEngine **engine** and Client **cl**
- Also created using the **mock()** method
- And then program their behavior with **when()**
  - **when(**engine.prepareMessage(template,cl)**)**
    .thenReturn(MSG_CONTENT**);**
  - **when(**cl.getEmail()**).thenReturn(**CL_EMAIL**);**

Test Double – Spy

```
public void sendMessage(Client cl, Template temp)  {
  String content = engine.prepareMessage(temp, cl);
  mailServer.send(client.getEmail(), content);
}
```

- Test Spy
  - MailServer **mailServer**
- Create spy with **mock()**
- Program behavior if needed with **when()**
- Verify interactions with **verify()**
  - **verify(**mailServer).send(CL_EMAIL, MSG_CONTENT);

TestMessenger – Spy Version

```
public class TestMessenger {
  private static final String CL_EMAIL = "some@email.com";
  private static final String MSG_CONTENT = "Dear John! You are fired.";
  @Test public void shouldSendEmailSpyVersion() {
    //Arrange
    Template template = mock(Template.class);
    Client client = mock(Client.class);
    MailServer mailServer = mock(MailServer.class);
    TemplateEngine templateEngine = mock(TemplateEngine.class);

    Messenger sut = new Messenger(mailServer, engine);

    when(client.getEmail()).thenReturn(CL_EMAIL);
    when(templateEngine.prepareMessage(template, client))
            .thenReturn(MSG_CONTENT);
    // Act
    sut.sendMessage(client, template);

    // Assert
    verify(mailServer).send(CL_EMAIL, MSG_CONTENT);

  }}
```

Test double: Mock Version

- How to create a mock in Mockito?
- **You can't!**
- **But**
  - In Mockito, you can specify the indirect inputs and outputs for a spy
  - So, what you can do with a mock you can also do with a spy
- And spy test double **follow** the AAA pattern

Extra - How to Handle Exceptions

- Use default approach
  - Use the expectedExceptions attribute of @Test
- Disadvantages
  - Cannot inspect all properties of caught exception
  - Exception could be thrown in the wrong place
  - Cannot examine the properties of OUT after exception
  - Cannot use the pattern Arrange/Act/Assertion

Example

```
public class RequestHandler {
  private final RequestProcessor processor;
  public void handle(Request request)
                    throws InvalidRequestException {
    if (invalidRequest(request)) {
      throw new InvalidRequestException();
    }
    processor.process(request);
  }
  // ...
}
```

- How to check that if request is invalid then exception id thrown and *process()* is not invoked?

Expected exception and interactions testing - 1

```
@Test(expectedExceptions=InvalidRequestException.class)
public void givenInvalidRequestWhenHandleThenThrowException()
                throws InvalidRequestException {
  RequestProcessor processor = mock(RequestProcessor.class);
  Request request = createInvalidRequest();
  RequestHandler sut = new RequestHandler(processor);

  sut.handle(request);
}
```

- Problem?

- Does not verify interactions with processor!

Expected exception and interactions testing - 2

```
@Test
public void givenInvalidRequestWhenHandleThenThrowException() {
  RequestProcessor processor = mock(RequestProcessor.class);
  Request request = createInvalidRequest();
  RequestHandler sut = new RequestHandler(processor);

  try {
    sut.handle(request);
    fail("Does not throw exception when request is invalid");
  } catch (InvalidRequestException ire) {
    verify(processor, never()).process(any());
  }
}
```

- Problem?
- Does not follow AAA pattern!

Expected exception and interactions testing - 3

```
@Test public void
givenInvalidRequestWhenHandleThenThrowException() {
  //Arrange
  RequestProcessor processor = mock(RequestProcessor.class);
  Request request = createInvalidRequest();
  RequestHandler sut = new RequestHandler(processor);

  // Act
  assertThrows(InvalidRequestException.class,
               () -> sut.handle(request));
  // Assert
  verify(processor, never()).process(any());
}
```

Classical and mocking testing

- Classical style
  - Use real objects whenever possible
  - and a test double if it is awkward to use a real object
  - Uses **state verification**
    - May imply breaking encapsulation of CUT
- Mockist style
  - Use a mock for any object with an interesting behavior
  - Uses **behavior verification**
  - High isolation
  - High dependency on implementation details

Extra

- Reduce boilerplate code

- More information:
  - https://site.mockito.org

Reduce boilerplate code

- How to avoid the several ***myMock = mock(SomeClass.class)*** that appear in the several test method?

- First approach
  - Define a field for each test double in the test class
  - Create test doubles in @BeforeTest method
  - But there is still some repetitive code
    - Creation of the SUT and test doubles
    - Inject test doubles into SUT

First approach

```
import org.mockito.Mock;

public class AnnotationsTest {
  private Collaborator collaborator;
  private SUT sut;

  @BeforeMethod private void setup() {
    sut = new SUT();
    collaborator = Mockito.mock(Collaborator.class);
    sut.setCollaborator(collaborator);
  }

  // test methods
}
```

2nd approach – @Mock Mockito Annotation

- All @Mock fields automatically initialized with a test double

```
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

public class AnnotationsTest {
  @Mock private Collaborator collaborator;
  private SUT sut;

  @BeforeTest private void setup() {
    sut = new SUT();
    sut.setCollaborator(collaborator);
  }
  // test methods
}
```

- Invoke *openMocks()* to initialize all test doubles declared with annotations
- Still need to inject test double in sut

2nd approach – Using openMocks explicitly

```
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

public class AnnotationsTest {
  @Mock private Collaborator collaborator;
  private SUT sut;
  private AutoCloseable closeable;

  @BeforeMethod private void openMocks() {
     closeable = MockitoAnnotations.openMocks(this);
  }

  @AfterMethod private void releaseMocks() throws Exception {
    closeable.close();
  }

  @BeforeMethod private void setup() {
    sut = new SUT();
    sut.setCollaborator(collaborator);
  }
  // test methods
}
```

2nd approach – Using openMocks implicitly

```
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import org.mockito.testng.MockitoTestNGListener;
import org.testng.annotations.Listeners;

@Listeners(MockitoTestNGListener.class)
public class AnnotationsTest {
  @Mock private Collaborator collaborator;
  private SUT sut;

  @BeforeMethod public void setup() {
    sut = new SUT();
    sut.setCollaborator(collaborator);
  }
  // test methods
}
```

3rd approach – @InjectMocks Annotation

- Automatizes the injection of test doubles in out
  1. Annotate the fields of test class with @InjectMock
  2. And annotate test doubles to inject with @Mock

- Mockito injects mocks only by
  1. constructor injection,
  2. property injection
  3. setter injection

3rd approach – @InjectMocks Annotation

- Automatizes the injection of test doubles in cut
- Annotate the fields of test class with @InjectMock
- And annotate test doubles to inject with @Mock

```
@Listeners(MockitoTestNGListener.class)
@Test public class ArticleManagerTest {
    @Mock private MyCalculator calculator;   // the test doubles  to
    @Mock private MyDatabase dbMock;      // inject

    @InjectMocks private MyManager manager;

    @Test public void shouldDoSomething() {
       manager.start();
       verify(dbMock).addListener(any());
    }
 }
```

3rd approach – @InjectMocks - 2

- Mockito injects mocks only by

1. Constructor injection
   - biggest constructor is chosen, then arguments are resolved with test doubles declared in the test only

```
public class MyManager {
  MyManager(MyCalculator calc, MyDatabase db)
  { /* parameterized constructor  */ }
}
```

2. Property injection
   - For each test double, check if there is a set method that can be used to inject the test double

```
public class MyManager {
    MyManager() {  } // no-arg constructor
     // setter
     void setDatabase(MyDatabase db) { }
     void setCalculator(MyCalculator calc) { }
}
```

3. Field injection
   1. For each test double, check if there is a field with the same type and set the field with reflection

```
public class MyManager {
      private MyDatabase database;
      private MyCalculator calculator;
   }
```

Spying on real objects

- Sometimes, you need to record the interactions and execute the real code on a collaborator
- Use spy() instead of mock()
- Creates a test double that
  - Is associated with a copy of a real object
  - records the interactions
  - For each interaction, it delegates the call to the real object
- This is not the same as the spy test double
- For spies, use *doReturn|Answer|Throw()* instead of *when()*

When **doReturn**(value) is mandatory

- When spying real objects and calling real methods on a spy brings side effects

```
List list = new ArrayList();
List spy = spy(list);
// specify behavior: get(0) returns “foo”
when(spy.get(0)).thenReturn("foo");

//Correct solution:
doReturn("foo").when(spy).get(0);
```

throws IndexOutOfBoundsException
(the real list is yet empty)

- Overriding a previous exception-stubbing

```
when(mock.foo()).thenThrow(new RuntimeException());
// … want to change behavior to return a value instead
when(mock.foo()).thenReturn("bar");

doReturn("bar").when(mock).foo();
```

throws RuntimeException

doNothing()

- Use doNothing() for setting void methods to do nothing
  - Stubbing consecutive calls on a void method

```
doNothing().doThrow(new RuntimeException()).when(mock).someVoidMethod();

mock.someVoidMethod();  // does nothing
mock.someVoidMethod();  // throws exception
```

  - To spy real objects and override (real) behaviour of void methods for doing nothing

```
List spy = spy(list);
doNothing().when(spy).clear();

spy.add("one");
spy.clear(); //clear() does nothing, so the list still contains "one"
```