

Lab 2: Advanced Unit Testing

Software Testing 2022

2022/03/03

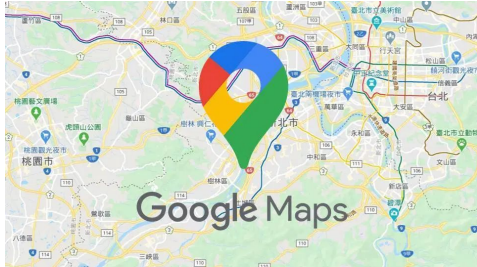
Remember Lab 1 ?

Class Vehicle



```
class VehicleTest {  
    @Test  
    void setSpeed() {...}  
  
    @Test  
    void setDir() {...}  
  
    @Test  
    void getSpeed() {...}  
  
    @Test  
    void getDir() {...}  
  
    @Test  
    void totalVehicle() {...}  
}
```

An Intelligent Vehicle, How To Test It?



Problem

Usually, the classification to be tested will have some external dependencies, may cause:

- Testing may be slow due to dependencies.
eg. Network, database, files, external objects, etc.
- The result of the misjudgment test is whether the SUT itself is wrong or the dependent object is wrong
- Wait for the development of dependent objects to be completed before testing the object under test
- Unable to test.
eg. the development environment is different from the formal environment

Solution: Test Double

Dummy:

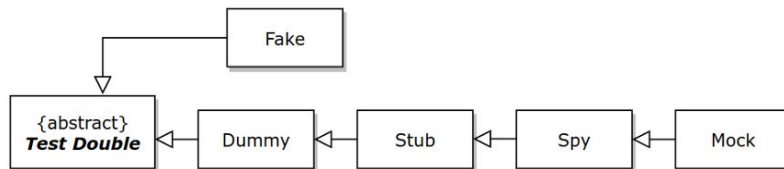
- It is used as a placeholder when an argument needs to be filled in.

Stub:

- It provides fake data to the SUT (System Under Test).

Spy:

- It records information about how the class is being used.



Mock:

- It defines an expectation of how it will be used. It will cause a failure if the expectation isn't met.

Fake:

- It is an actual implementation of the contract but is unsuitable for production.

Example

Real World



authorization code / token



```
{  
  "name": "Meow Yang",  
  "email": "aesophor0613@gmail.com",  
  "token": "Wu13RnH0w220sE"  
}
```

Fake



authorization code / token



Simple logic implements



```
{  
  "name": "Meow Yang",  
  "email": "aesophor0613@gmail.com",  
  "token": "Wu13RnH0w220sE"  
}
```


Stub



authorization code / token



Implements without logic



```
{  
  "name": "Meow Yang",  
  "email": "aesophor0613@gmail.com",  
  "token": "Wu13RnH0w220sE"  
}
```

Mock



Only care the interactive between target and Mock object

Spy



Can check the interactive between target and stub object

Fake

Fake

```
public interface GoogleApi {  
    String login(String code);  
}  
  
public class MyGoogleApi implements GoogleApi {  
    public String login(String code) {  
        //do something and return something  
    }  
}
```

Stub

In Case Require Network Connection

```
final String initialString = "From Server : Hi !";  
//Guess what server responses //Not good
```

```
final Socket socket = new Socket("127.0.0.1", 6666);
```

```
TcpClientParseCommunicate tcpClientParseCommunicate = new TcpClientParseCommunicate(socket);  
tcpClientParseCommunicate.communicate();  
tcpClientParseCommunicate.parseInput();  
StringBuffer sb = tcpClientParseCommunicate.getBuf();  
  
assertEquals(initialString, sb.toString());
```

Stub Test

```
class SocketStub extends Socket {  
    SocketStub(String host, int port) {  
        //Without connect with remote  
    }  
  
    public InputStream getInputStream() {  
        return targetStream;  
    }  
}
```

**SocketStub does not make network connection
Only return the written targetStream**

Stub Test - Cont.

```
final String initialString = "testTcpClientWithStub";  
final InputStream targetStream = new ByteArrayInputStream(initialString.getBytes());  
  
final Socket socket = new SocketStub(null, -1);  
  
TcpClientParseCommunicate tcpClientParseCommunicate = new TcpClientParseCommunicate(socket);  
tcpClientParseCommunicate.communicate();  
tcpClientParseCommunicate.parseInput();  
StringBuffer sb = tcpClientParseCommunicate.getBuf();  
  
assertEquals(initialString, sb.toString());
```

Mockito

It is a widely used testing framework, especially it can easily handle dependency injection scenarios, and it is relatively helpful to write Unit Test with it.

Can more easily handle and construct a variety of Test Double to conduct Unit Test.



<https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html>

Basic Structure

```
@ExtendWith(MockitoExtension.class)
public class ExampleTest {

    @Mock
    private List<Integer> list;

    @Test
    public void shouldDoSomething() {
        list.add(100);
    }
}
```

Constructor Injection

```
//instead:  
@Spy BeerDrinker drinker = new BeerDrinker();  
//you can write:  
@Spy BeerDrinker drinker;  
  
//same applies to @InjectMocks annotation:  
@InjectMocks LocalPub;
```

mockito will try to initialize the **@InjectMocks** variables in **@Spy**, either by constructing method, set method or variable injection.

Stub Test With Mockito

```
final String initialString = "testTcpClientWithStubMockito";
final InputStream targetStream = new ByteArrayInputStream(initialString.getBytes());

Socket clientStub = mock(Socket.class);
when(clientStub.getInputStream()).thenReturn(targetStream);

TcpClientParseCommunicate tcpClientParseCommunicate = new TcpClientParseCommunicate(socket);
tcpClientParseCommunicate.communicate();
tcpClientParseCommunicate.parseInput();
StringBuffer sb = tcpClientParseCommunicate.getBuf();

assertEquals(initialString, sb.toString());
```

Cheat Sheet

```
// Only one stub method
FooClass mockObject = mock(FooClass.class);
when(mockObject.method(value)).thenReturn(returnValue);

// Two stub method
FooClass mockObject = mock(FooClass.class);
when(mockObject.method1(value)).thenReturn(returnValue);
when(mockObject.method2(value1, value2)).thenReturn(returnValue2);

// Use matcher to match stub method
when(mockObject.method(anyInt(), anyBoolean())).thenReturn(value);
```



Mock

Mock Test With Mockito

```
Socket clientMock = mock(Socket.class);

TcpClientParseCommunicate tcpClientParseCommunicate
= new TcpClientParseCommunicate(clientMock);
tcpClientParseCommunicate.communicate();

verify(clientMock).getInputStream();
```


Mock Test With Mockito - Cont.

```
Socket clientMock = mock(Socket.class);

TcpClientParseCommunicate tcpClientParseCommunicate
= new TcpClientParseCommunicate(clientMock);

verify(clientMock, never()).getInputStream();
```

Cheat Sheet

- Frequency

- `verify(mockObject).method();`
- `verify(mockObject, times(666)).method();`
- `verify(mockObject, never()).method();`

- Argument Type

- `verify(mockObject).method("robert");`
- `verify(mockObject).method(anyString());`
- `verify(mockObject).method(2021, 3, 11);`
- `verify(mockObject).method(anyInt(), anyInt(), anyInt());`

Cheat Sheet - Cont.

- Capturing Arguments

```
ArgumentCaptor<Person> argument = ArgumentCaptor.forClass(Person.class);  
verify(mock).doSomething(argument.capture());  
assertEquals("John", argument.getValue().getName());
```

//capturing varargs:

```
ArgumentCaptor<Person> varArgs = ArgumentCaptor.forClass(Person.class);  
verify(mock).varArgMethod(varArgs.capture());  
List expected = asList(new Person("John"), new Person("Jane"));  
assertEquals(expected, varArgs.getAllValues());
```



Spy

Example

```
List list = new LinkedList();
List 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
System.out.println(spy.get(0));

//size() method was stubbed - 100 is printed
System.out.println(spy.size());

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

Example - Cont.

```
List list = new LinkedList();  
List spy = spy(list);  
  
//Impossible: real method is called so spy.get(0) throws IndexOutOfBoundsException  
//(the list is yet empty)  
    when(spy.get(0)).thenReturn("foo");  
  
//You have to use doReturn() for stubbing  
doReturn("foo").when(spy).get(0);
```

Cheat Sheet

You can use `doThrow()`, `doAnswer()`, `doNothing()`, `doReturn()` and `doCallRealMethod()` in place of the corresponding call with `when()`, for any method. It is necessary when you :

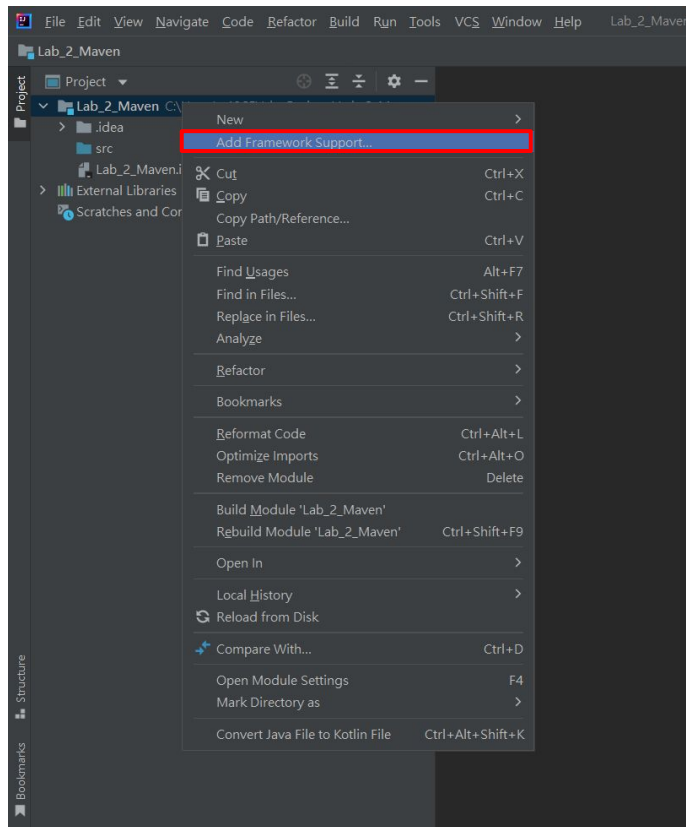
- stub void methods
- stub methods on spy objects
- stub the same method more than once, to change the behaviour of a mock in the middle of a test.

Lab

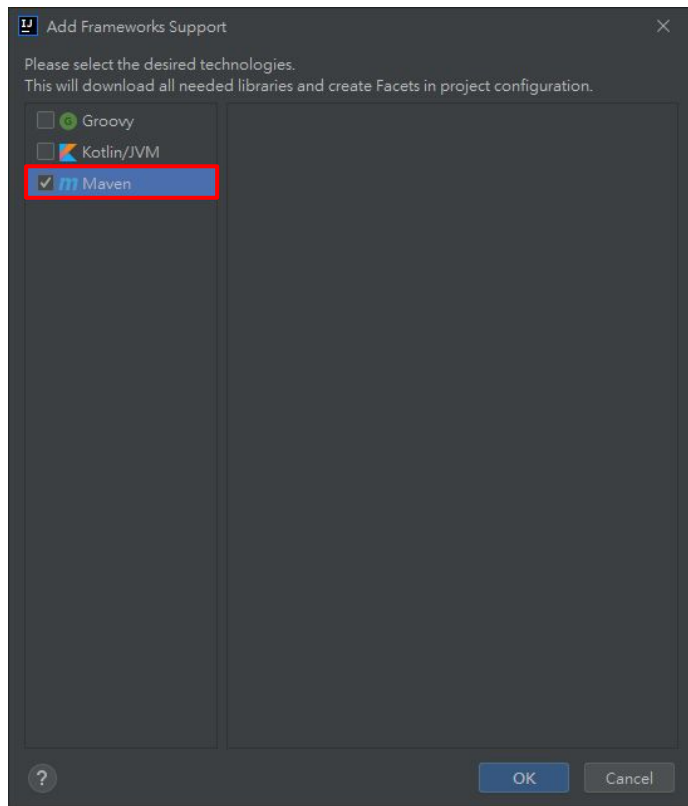
Lab 2

1. Download **StrangeGame.java** from Github.
 - a. <https://github.com/a4865g/NYCU-Software-Testing-2022>
2. Write tests for **StrangeGame** class which satisfy the following case:
 - a. If a **notorious** player **enter the game** on **0:00 - 11:59**, verify that prison doesn't do any imprisonment.
 - b. If a **notorious** player **enter the game** on **12:00 - 23:59**, assert the output correct.
 - c. Suppose **3** players go to the prison. Verify **prisonerLog** in **prison** will record prisoner's **playerid** with **spy** method. **Don't stub getLog function.**
 - d. Use **stub** method to test **getScore** function (PlayerID = your StudentID) to avoid connection to outer database.
 - e. Implement **paypalService** interface as a **fake** object to test donate function.
3. Name your test function **test_a to test_e** which belong to each case.
4. Upload **StrangeGameTest.java** to E3

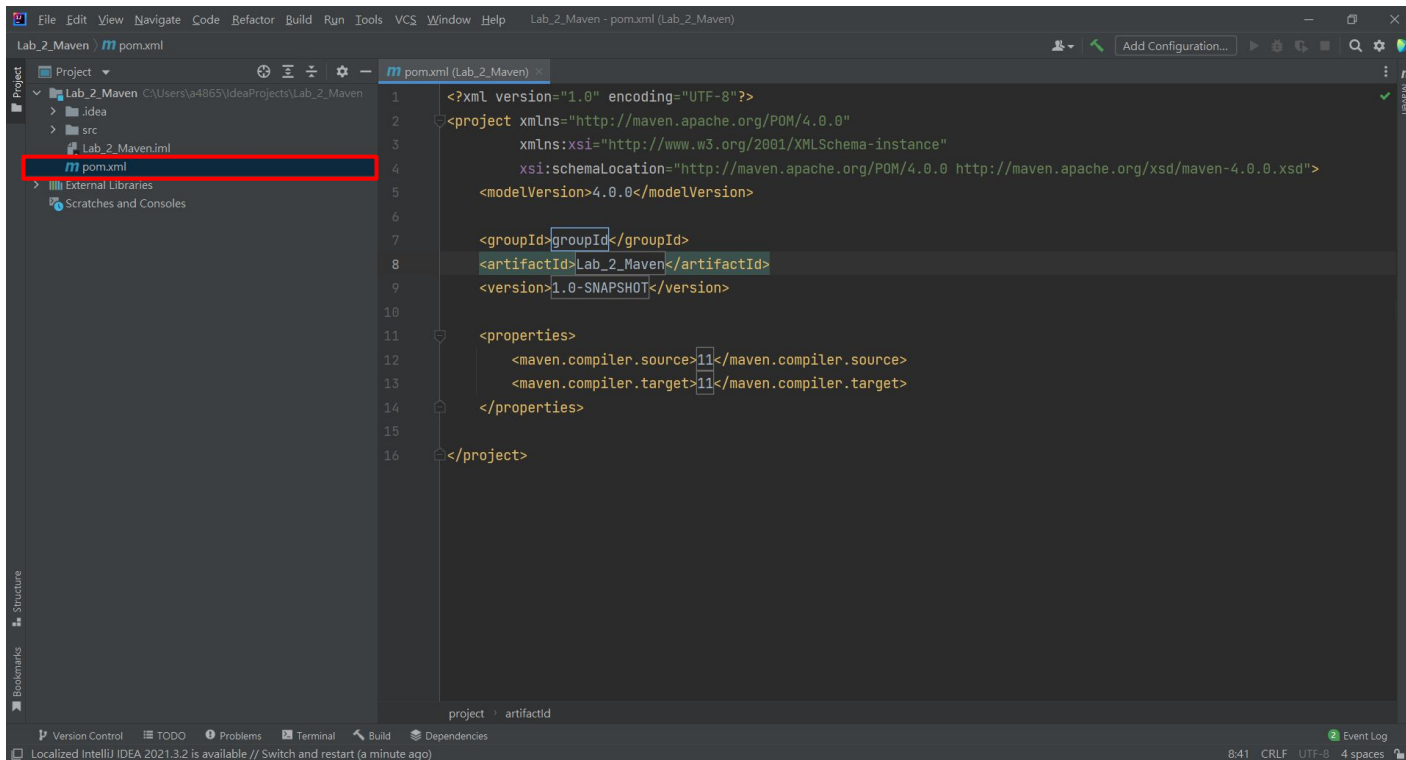
Import mockito - Method 1 (Maven)



Import mockito - Method 1

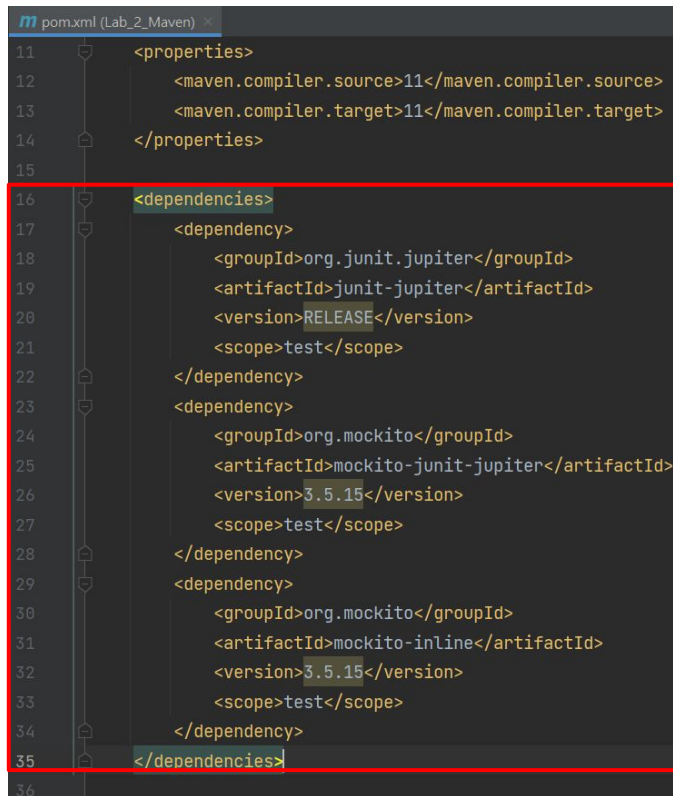


Import mockito - Method 1



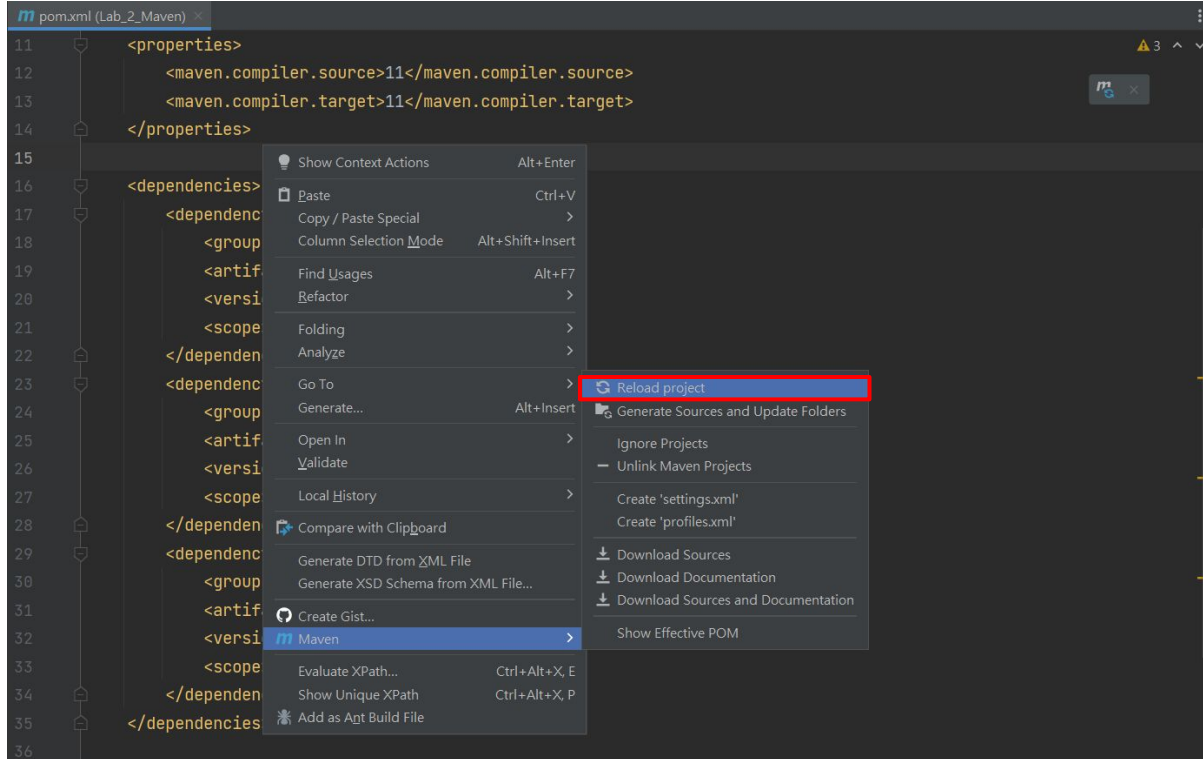
Import mockito - Method 1

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>RELEASE</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>3.5.15</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-inline</artifactId>
    <version>3.5.15</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```



```
m pom.xml (Lab_2_Maven) x
11 <properties>
12   <maven.compiler.source>11</maven.compiler.source>
13   <maven.compiler.target>11</maven.compiler.target>
14 </properties>
15
16 <dependencies>
17   <dependency>
18     <groupId>org.junit.jupiter</groupId>
19     <artifactId>junit-jupiter</artifactId>
20     <version>RELEASE</version>
21     <scope>test</scope>
22   </dependency>
23   <dependency>
24     <groupId>org.mockito</groupId>
25     <artifactId>mockito-junit-jupiter</artifactId>
26     <version>3.5.15</version>
27     <scope>test</scope>
28   </dependency>
29   <dependency>
30     <groupId>org.mockito</groupId>
31     <artifactId>mockito-inline</artifactId>
32     <version>3.5.15</version>
33     <scope>test</scope>
34   </dependency>
35 </dependencies>
36
```

Import mockito - Method 1



Import mockito - Method 2 (JAR)

Steps for adding external jars in IntelliJ IDEA:

1. Click **File** from the toolbar
2. Project Structure (**CTRL** + **SHIFT** + **ALT** + **S** on Windows/Linux, **⌘** + **;** on Mac OS X)
3. Select Modules at the left panel
4. Dependencies tab
5. '+' → JARs or directories

```
import org.junit.jupiter.api.extension.ExtendWith;  
import org.mockito.*;  
import org.mockito.junit.jupiter.MockitoExtension;  
  
import static org.mockito.Mockito.*;
```



master ▾

[NYCU-Software-Testing-2022 / Lab_2 / jar](#)



a4865g Add Lab 2

..



byte-buddy-1.10.22.jar



byte-buddy-agent-1.10.22.jar



mockito-core-3.8.0.jar



mockito-junit-jupiter-3.8.0.jar



objenesis-3.1.jar