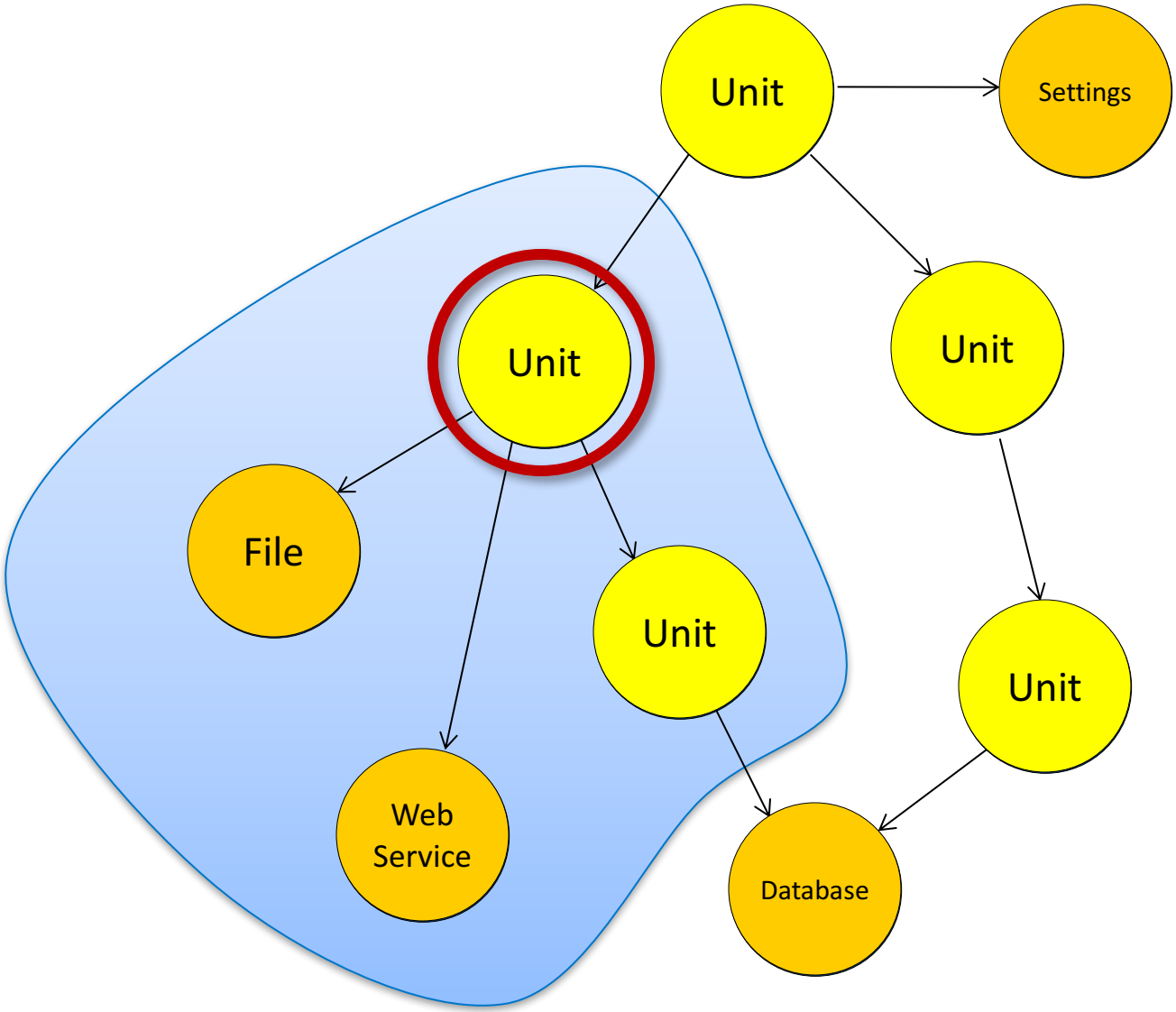
Mocking with

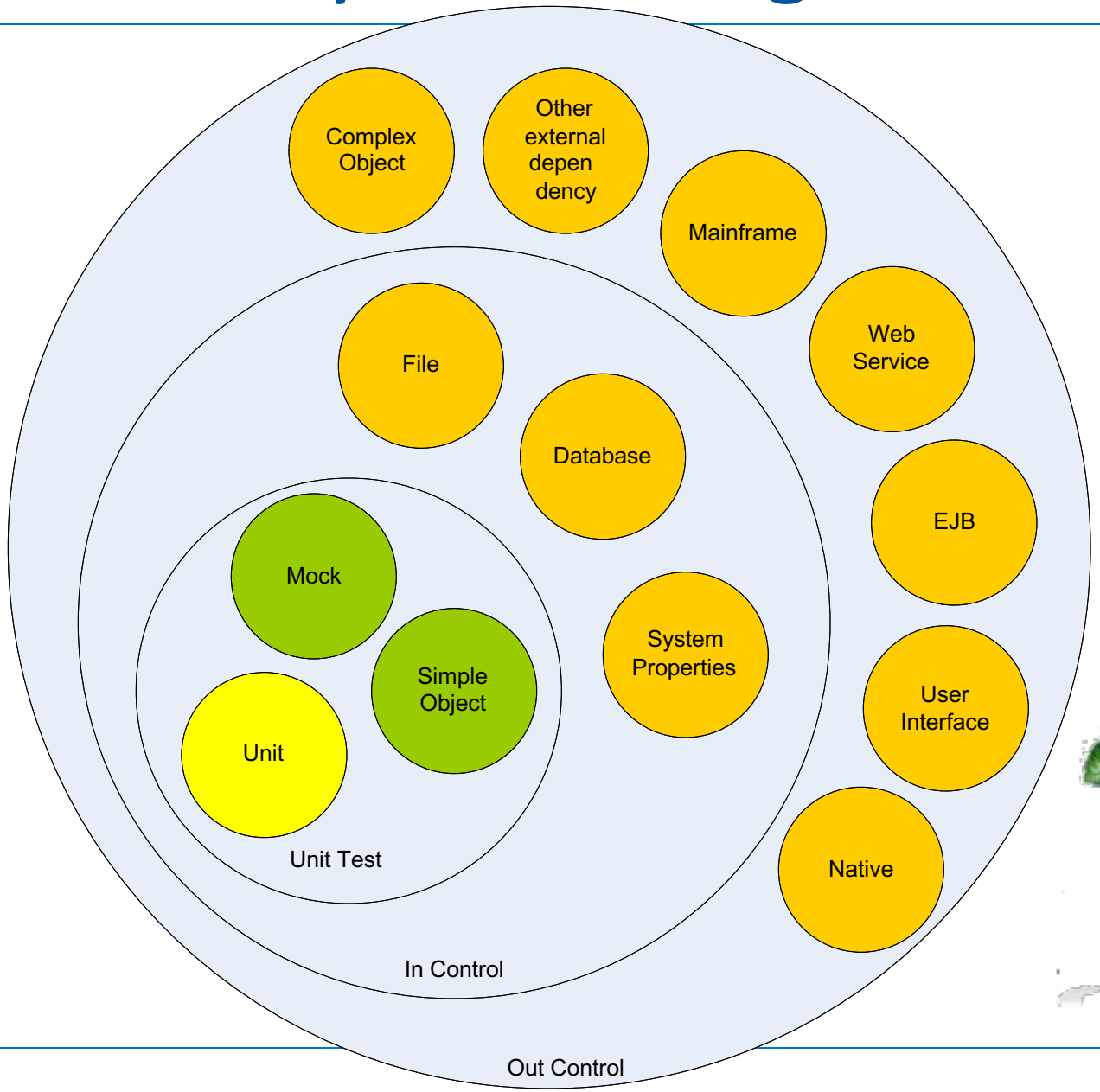
Unit Testing & Dependencies



Testing in Isolation

- A unit should be tested in isolation
  - Not depending on other units or the environment
- Dependencies are ‘mocked’
- A mock is a replacement for a real object
- A mock emulates behavior of the real object
- With mocks, a developer has control over the environment

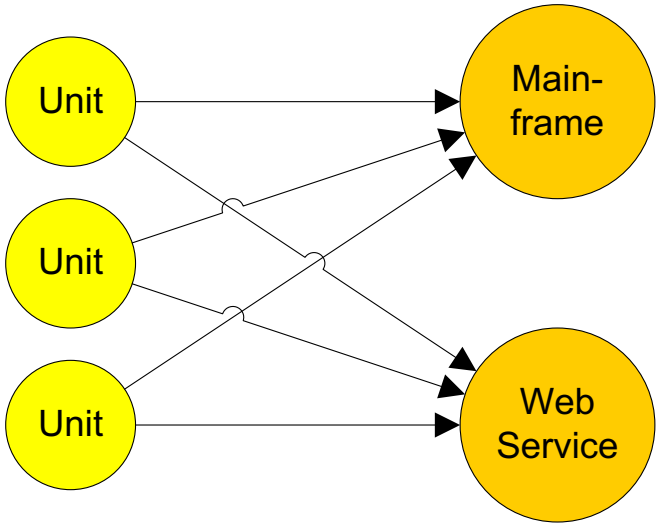
Dependency onion ring



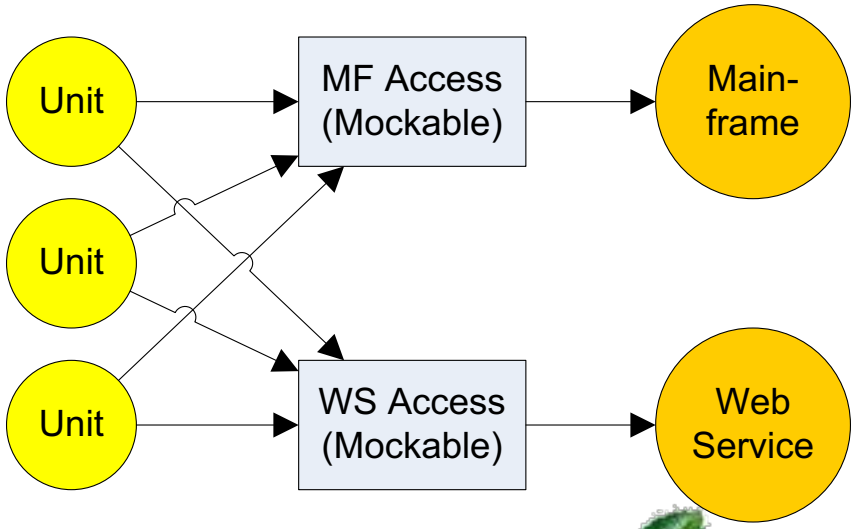
Design for testability

Loosely coupling improves testability

- Mocking becomes easier

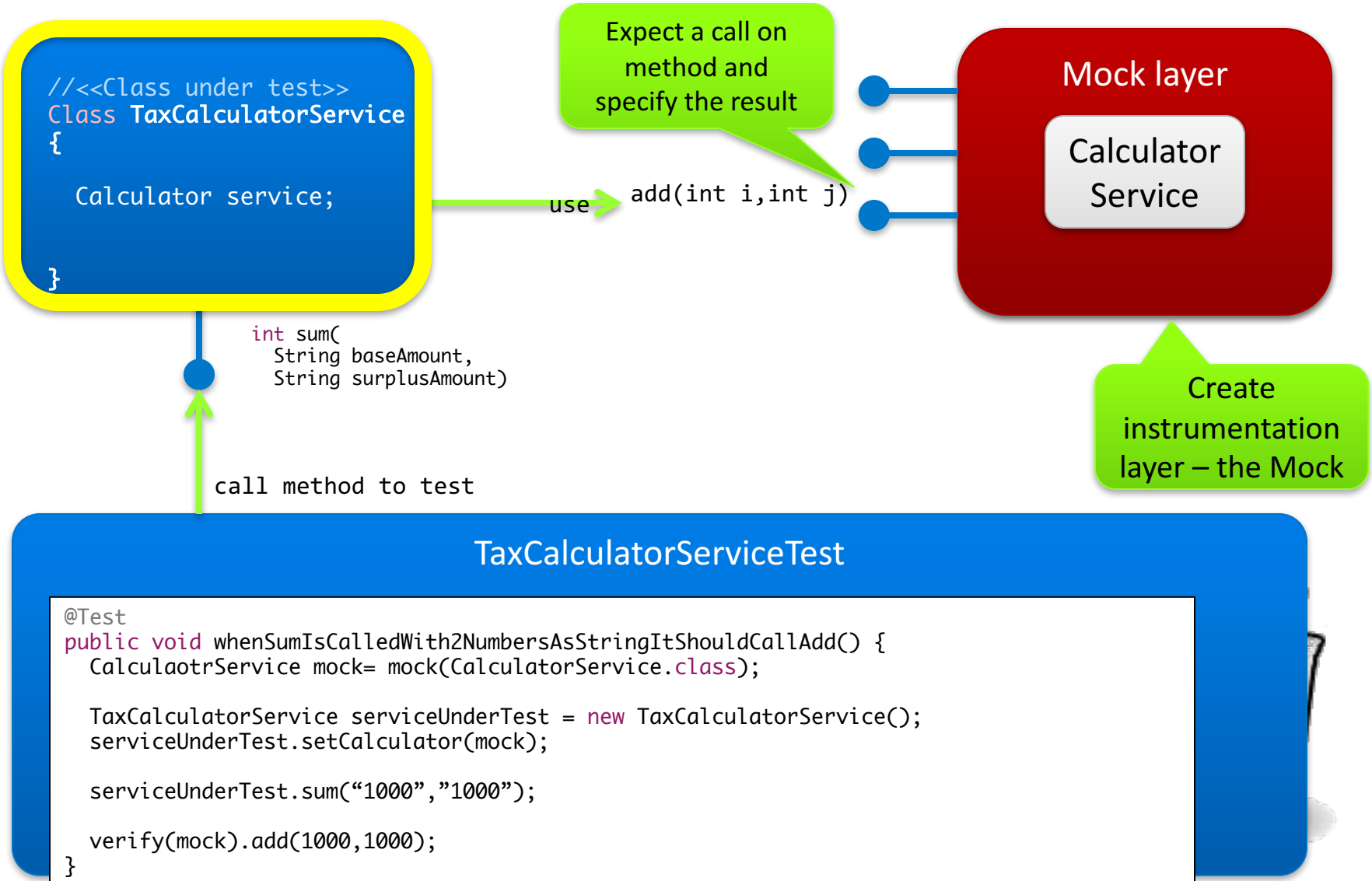


Units are hard to test



Units are easy to test

What is mocking?



A really Easy mocking framework

- Open Source
- Based on EasyMock
- Easier than EasyMock
  - No replay status
  - No Strict/Nice/Default distinction
- It tastes good!

Steps when using mocking

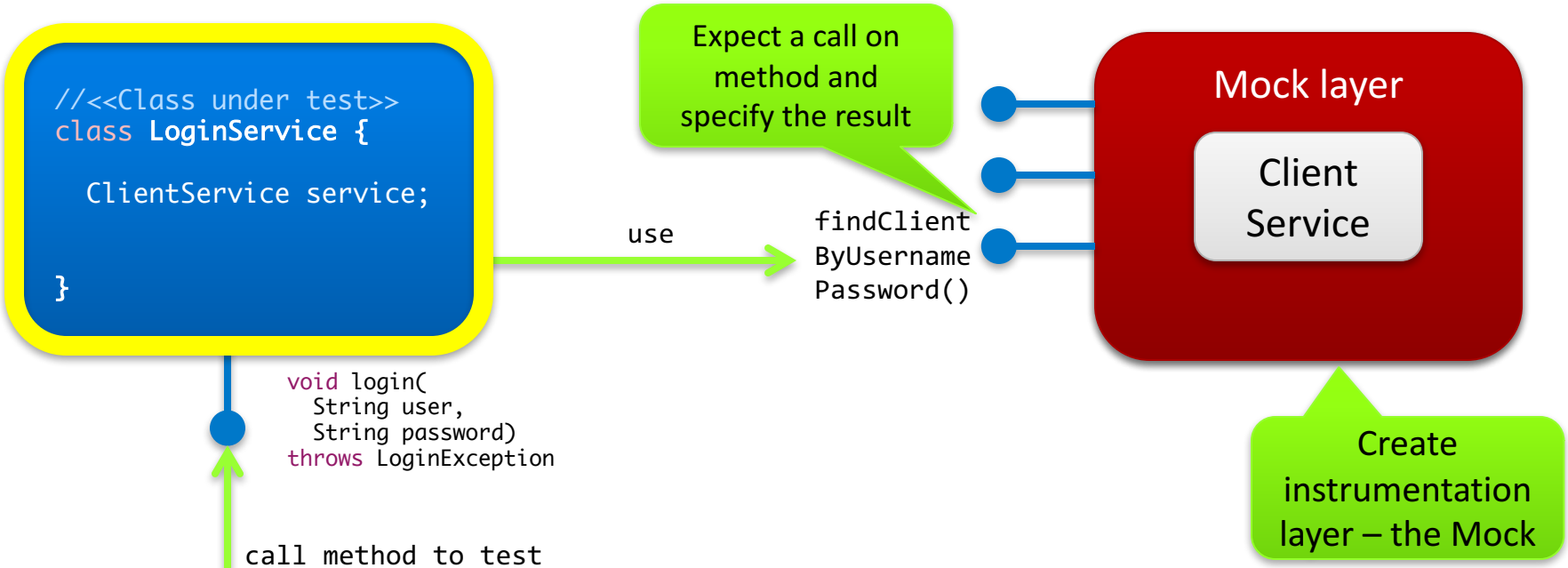
Step 1: Create the mock

Step 2: program the mock for stubbing

Step 3: Use the mock

Step 4: Verify the mock

What is mocking?



LoginServiceTest

```
@Test(expected=LoginException.class)
public void loginServiceThrowsLoginExceptionWithUnknownUser() {
  ClientService clientServiceMock = mock(ClientService.class);
  when(clientServiceMock.findClientByUsernameAndPassword(
                    "unknownuser", "")).thenReturn(null);
  LoginService loginService = new LoginService();
  loginService.setClientService(clientServiceMock);

  loginService.login("unknownuser", "");
}
```

Let's start

- To create a Mock:

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

Calculator mock = mock(Calculator.class);
```

- After this statement, you can use the mock

```
mock.add(1, 2);
```

- Verify the mock

```
verify(mock).add(1, 2);
```

More about mock creation

- Default way to create a mock:

```
Calculator mock = mock(Calculator.class);
```

- Other option is to use annotations:

```
@Mock List mockedList;
```

  - Don't forget to init your class!

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

  - Or use `@RunWith(MockitoJUnitRunner.class)`

More about verifying

- Default way to verify

```
verify(mock).add(1, 2);
```

  - Don't forget this!

- Other verify options:

```
verify(mock, times(5)).add(1, 2);

verify(mock, atLeast(2)).add(3, 4);

verify(mock, atLeastOnce()).add(0, 0);

verify(mock, never()).add(-1, -1);
```

More about verifying

- Verify for no more interaction

```
mock.add(0, 0);
mock.min(7, 1);

verify(mock, atLeastOnce()).add(0, 0);

verifyNoMoreInteractions(mock);
```

More about verifying cont.

- If order of calls is important
- Use *InOrder* object

```
Calculator mock= mock(Calculator.class);

mock.add(1, 2);
mock.min(8, 1);

InOrder inOrder = inOrder(mock);
inOrder.verify(mock).add(1, 2);
inOrder.verify(mock).min(8, 1);
```

Capture argument on mock call

- Check a test specific state of a parameter supplied to a mocked call
- Use an ArgumentCaptor<T> where T is the type of the parameter to capture

```
ArgumentCaptor<Cursist> arg =
                    ArgumentCaptor.forClass(Cursist.class);

verify(mock).save((arg.capture()));

assertThat(arg.getValue().isActive(), is(false));
```

Create a stub,a mock with behavior

- First, Create the mock
- Second, Use the static *WHEN* method
- Third, Program the stub return value

```
when(mock.add(5, 6)).thenReturn(11);
```

- Or throw an exception

```
when(mock.add(5, 6))
   .thenThrow(
        new IllegalStateException());
```

More about stubs

- For void methods use the *do...()* method

```
doThrow(
  new RuntimeException()
).when(objects).clear();
```

Stubs support a flexible syntax

- Declare the return value for multiple calls of hit() on the stub

```
when(counter.hit())
        .thenReturn(0)
        .thenReturn(1)
        .thenReturn(2);
```

Flexible syntax for arguments

- Use any arguments

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

when(
   mock.add(anyInt(), anyInt())
).thenThrow(
   new IllegalStateException()
);
```

But not that flexible!

- Don’t mix any and literal values!

```
when(mock.add(10, anyInt()))
```

Invalid use of argument matchers!
2 matchers expected, 1 recorded:
-> at …

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"));
```

Use matchers instead

- In that case use matchers (for example: eq)

```
when(mock.add(eq(10), anyInt()))
```

Use dynamic stub return values

- The stub must return a value depending on the parameters, i.e. our stub is dynamic
- Use Answers for dynamic stubs

```
when(mock.add(anyInt(), anyInt())).then(new Answer<Integer>() {

  @Override
  public Integer answer(
                    InvocationOnMock invocation) throws Throwable {
    return (Integer) invocation.getArguments()[0]
         + (Integer) invocation.getArguments()[1];
  }
});
```

Mock only a part of a class

- In some cases you only want to mock a part of an object
- Use the static method *spy()*

```
List<Object> objects = spy(new ArrayList<>());
when(objects.remove(anyObject())).thenThrow(
                new IllegalStateException("my spy method"));
objects.remove(new Object());
```

This doesn't work for final methods or classes!

Questions