Sins of mocking

How to mock correctly?

Jürg Weilenmann, 24.06.2021

Problem

- Had to add some ‚non functional‘ extension to existing code
- code base +10 years
- large class
- no tests and not testable (at least for me)

Plan

- extract the part to change into a separate class with ‚low-risk‘ changes to the existing code
- write a test
- mock all ‚external‘ services
- add the new functionality

Actual Class

```
int largeMethod(int operation){  // in a large class
    switch(operation){
        case 0:
            return doSomething0("some data");
            // ....
        case 999:
           return doSomething999("some data");
    }
}
```

Extracting command

```
int largeMethod(int operation){
    switch(operation){
        case 0:
            return new DoSomething0Command().execute("some data");
        // ....
}

class DoSomething0Command {
   public int execute(String data) {
       createService().callIt(data);
       return 0;
    }

    private ExternalService createService() {
        return ExternalServiceFactory.createService();
    }
}
```

First Test

```
void test(){
    DoSomething0Command command = new DoSomething0Command();

    int result = command.execute("some data");

    assertEquals(0, result);
}
```

- failed for wrong reason
- ExternalServiceFactory.createService() with JEE lookup magic
- had to mock it

Option 1 (my favourite ;-) , seen quite often in our legacy code)

```
void test(){
    ExternalService externalService = createMock();
    DoSomething0Command command = new DoSomething0Command(){
        @Override
        ExternalService createService() {
            return externalService;
        }
    };

    int result = command.execute("some data");

    assertEquals(0, result);
}
```

- changing visibility of createService() just for testing
- not testing the real class but something different
- have to know what method to override

Option 2: passing mock into function call

```
void test(){
    ExternalService externalServiceMock = createMock();
    DoSomething0Command command = new DoSomething0Command()

    int result = command.execute("some data“,
                                 externalServiceMock);

    assertEquals(0, result);
}
```

- the parameter gives me a hint what to mock
- changes signature of the command method and probably each command needs different services
- service is now created at an earlier phase -> not exactly what we had before, might be a problem when JEE looking up the bean

Option 3: create an additional constructor for testing

```
void test(){
    ExternalService externalService = createMock();
    DoSomething0Command command = new
                                  DoSomething0Command(externalService)
    int result = command.execute("some data");

    assertEquals(0, result);
}
```

- code used only for testing
- service class must handle both cases

Option 4: create an additional constructor passing in a factory method

```
public class DoSomething0Command {
    private final Supplier<ExternalService> externalServiceSupplier;

    public DoSomething0Command() {
        this(DoSomething0Command::createService);
    }

    DoSomething0Command(Supplier<ExternalService> externalServiceSupplier) {
        this.externalServiceSupplier = externalServiceSupplier;
    }

    public int execute(String data) {
        externalServiceSupplier.get().callIt(data);
        return 0;
    }

    private static ExternalService createService() {
        return ExternalServiceFactory.createService();
    }
```

- same code for test and production, but still a constructor that is used only for testing
- no change in the interface

Option 5: CDI way

```
public class DoSomething0Command {
    private final Instance<ExternalService> externalService;

    @Inject
    public DoSomething0Command2(Instance<ExternalService> externalService) {
        this.externalService = externalService;
    }

    public int execute(String data) {
        externalService.get().callIt(data);
        return 0;
    }
```

- no special code for testing anymore
- creating the factory is delegated to a producer
- looks good, but our legacy code is not managed by CDI

Other Options

- use reflection to override internals (but needs knowledge of the internals)
- …

Let’s go ‚flying‘ to learn how to do it right

Merçi for listening

Contact:

Mail: juerg.weilenmann@css.ch

Twitter: -

Facebook: -

LinkedIn: -

Instagram: -

WhatsApp: -

Git: -

BeachBar: 18:00 - 20:00