When should I mock?

Asked 16 years, 3 months ago Modified 1 year, 2 months ago

Viewed 92k times

186

I have a basic understanding of mock and fake objects, but I'm not sure I have a feeling about when/where to use mocking - especially as it would apply to this scenario here.

unit-testing language-agnostic mocking

Share Follow

edited May 23, 2017 at 11:55

Community Bot

1 • 1

asked Sep 1, 2008 at 17:07

Esteban Araya

29.6k • 25 • 111 • 141

1 I recommend only mocking out-of-process dependencies and only those of them, interactions with which are observable externally (SMTP server, message bus, etc). Don't mock the database, it's an implementation detail. More about it here: enterprisecraftsmanship.com/posts/when-to-mock – Vladimir Apr 19, 2020 at 13:03

6 Answers

Sorted by: Highest score (default)

Mock objects are useful when you want to **test interactions** between a class under test and a particular interface.

For example, we want to test that method `sendInvitations(MailServer mailServer)` calls `MailServer.createMessage()` exactly once, and also calls `MailServer.sendMessage(m)` exactly once, and no other methods are called on the `MailServer` interface. This is when we can use mock objects.

With mock objects, instead of passing a real `MailServerImpl`, or a test `TestMailServer`, we can pass a mock implementation of the `MailServer` interface. Before we pass a mock `MailServer`, we "train" it, so that it knows what method calls to expect and what return values to return. At the end, the mock object asserts, that all expected methods were called as expected.

This sounds good in theory, but there are also some downsides.

Mock shortcomings

If you have a mock framework in place, you are tempted to use mock object *every time* you need to pass an interface to the class under the test. This way you end up **testing interactions even when it is not necessary**. Unfortunately, unwanted (accidental) testing of interactions is bad, because then you're testing that a particular requirement is implemented in a particular way,

instead of that the implementation produced the required result.

Here's an example in pseudocode. Let's suppose we've created a `MySorter` class and we want to test it:

```
// the correct way of testing
testSort() {
    testList = [1, 7, 3, 8, 2]
    MySorter.sort(testList)

    assert testList equals [1, 2, 3, 7, 8]
}


// incorrect, testing implementation
testSort() {
    testList = [1, 7, 3, 8, 2]
    MySorter.sort(testList)

    assert that compare(1, 2) was called once
    assert that compare(1, 3) was not called
    assert that compare(2, 3) was called once
    ....
}
```

(In this example we assume that it's not a particular sorting algorithm, such as quick sort, that we want to test; in that case, the latter test would actually be valid.)

In such an extreme example it's obvious why the latter example is wrong. When we change the implementation of `MySorter`, the first test does a great job of making sure we still sort correctly, which is the whole point of tests - they allow us to change the code safely. On the other

hand, the latter test *always* breaks and it is actively harmful; it hinders refactoring.

Mocks as stubs

Mock frameworks often allow also less strict usage, where we don't have to specify exactly how many times methods should be called and what parameters are expected; they allow creating mock objects that are used as stubs.

Let's suppose we have a method `sendInvitations(PdfFormatter pdfFormatter, MailServer mailServer)` that we want to test. The `PdfFormatter` object can be used to create the invitation. Here's the test:

```
testInvitations() {
   // train as stub
   pdfFormatter = create mock of PdfFormatter
   let pdfFormatter.getCanvasWidth() returns 100
   let pdfFormatter.getCanvasHeight() returns 300
   let pdfFormatter.addText(x, y, text) returns
true
   let pdfFormatter.drawLine(line) does nothing

   // train as mock
   mailServer = create mock of MailServer
   expect mailServer.sendMail() called exactly
once

   // do the test
   sendInvitations(pdfFormatter, mailServer)

   assert that all pdfFormatter expectations are
met
```

```
    assert that all mailServer expectations are met
}
```

In this example, we don't really care about the `PdfFormatter` object so we just train it to quietly accept any call and return some sensible canned return values for all methods that `sendInvitation()` happens to call at this point. How did we come up with exactly this list of methods to train? We simply ran the test and kept adding the methods until the test passed. Notice, that we trained the stub to respond to a method without having a clue why it needs to call it, we simply added everything that the test complained about. We are happy, the test passes.

But what happens later, when we change `sendInvitations()`, or some other class that `sendInvitations()` uses, to create more fancy pdfs? Our test suddenly fails because now more methods of `PdfFormatter` are called and we didn't train our stub to expect them. And usually it's not only one test that fails in situations like this, it's any test that happens to use, directly or indirectly, the `sendInvitations()` method. We have to fix all those tests by adding more trainings. Also notice, that we can't remove methods no longer needed, because we don't know which of them are not needed. Again, it hinders refactoring.

Also, the readability of test suffered terribly, there's lots of code there that we didn't write because of we wanted to, but because we had to; it's not us who want that code

there. Tests that use mock objects look very complex and are often difficult to read. The tests should help the reader understand, how the class under the test should be used, thus they should be simple and straightforward. If they are not readable, nobody is going to maintain them; in fact, it's easier to delete them than to maintain them.

How to fix that? Easily:

- Try using real classes instead of mocks whenever possible. Use the real `PdfFormatterImpl`. If it's not possible, change the real classes to make it possible. Not being able to use a class in tests usually points to some problems with the class. Fixing the problems is a win-win situation - you fixed the class and you have a simpler test. On the other hand, not fixing it and using mocks is a no-win situation - you didn't fix the real class and you have more complex, less readable tests that hinder further refactorings.
- Try creating a simple test implementation of the interface instead of mocking it in each test, and use this test class in all your tests. Create `TestPdfFormatter` that does nothing. That way you can change it once for all tests and your tests are not cluttered with lengthy setups where you train your stubs.

All in all, mock objects have their use, but when not used carefully, **they often encourage bad practices, testing implementation details, hinder refactoring and**

produce difficult to read and difficult to maintain tests.

For some more details on shortcomings of mocks see also Mock Objects: Shortcomings and Use Cases.

Share Follow

edited Jun 20, 2020 at 9:12

Community Bot

1 ● 1

answered Sep 1, 2008 at 18:34

Jan Soltis

4,419 ● 2 ● 27 ● 24

1 A well thought out answer, and I mostly agree. I would say that since unit tests are white-box testing, having to change the tests when you change the implementation to send fancier PDFs may not be an unreasonably burden. Sometimes mocks can be useful way to quickly implement stubs instead of having a lot of boiler plate. In practice it does seem that their use is not retricted to these simple cases, however. – Draemon Mar 24, 2012 at 13:46

2 isn't the whole point of a mock is that your tests are consistent, that you don't have to worry about mocking on objects whose implementations are continually changing possibly by other programmers each time you run your test and you get consistent test results? – PositiveGuy Apr 1, 2013 at 19:27

3 Very good and relevant points (especially about tests fragility). I used to use mocks a lot when i was younger, but now i consider unit test that heavilly depend on mocks as potentially disposable and focus more on integration testing (with actual components) – Kemoda Jun 18, 2013 at 12:35

9 "Not being able to use a class in tests usually points to some problems with the class." If the class is a service(e.g. access to database or proxy to web service), it should be considered as an external dependancy and mocked/stubbed – Michael Freidgeim Jul 6, 2013 at 3:40

3 @MichaelFreidgeim Why mock database? Why not use an embedded database to test the same thing? That way you don't have separate unit and integration tests. Also your tests perform faster as required because the database is embedded. – Narendra Pathai Nov 16, 2014 at 8:43

167

A unit test should test a single codepath through a single method. When the execution of a method passes outside of that method, into another object, and back again, you have a dependency.

When you test that code path with the actual dependency, you are not unit testing; you are integration testing. While that's good and necessary, it isn't unit testing.

If your dependency is buggy, your test may be affected in such a way to return a false positive. For instance, you may pass the dependency an unexpected null, and the dependency may not throw on null as it is documented to do. Your test does not encounter a null argument exception as it should have, and the test passes.

Also, you may find its hard, if not impossible, to reliably get the dependent object to return exactly what you want during a test. That also includes throwing expected exceptions within tests.

A mock replaces that dependency. You set expectations on calls to the dependent object, set the exact return values it should give you to perform the test you want, and/or what exceptions to throw so that you can test your exception handling code. In this way you can test the unit in question easily.

TL;DR: Mock every dependency your unit test touches.

Share Follow

edited Jan 2, 2021 at 10:09

Michał Krzywański

16.8k ● 5 ● 43 ● 67

answered Sep 1, 2008 at 18:27

user1228

243 This answer is too radical. Unit tests can and should exercise more than a single method, as long as it all belongs to the same cohesive unit. Doing otherwise would require way too much mocking/faking, leading to complicated and fragile tests. Only the dependencies that don't really belong to the unit under test should be replaced through mocking. – Rogério Aug 26, 2010 at 13:58

12 This answer is also too optimistic. It would be better if it incorporated @Jan's shortcomings of mock objects. – Jeff Axelrod May 26, 2011 at 19:45

2 Isn't this more of an argument for injecting dependencies for tests rather than mocks specifically? You could pretty much replace "mock" with "stub" in your answer. I agree that you should either mock or stub the significant dependencies. I've seen a lot of mock-heavy code which basically ends up reimplementing parts of the mocked

objects; mocks certainly aren't a silver bullet. – Draemon Mar 24, 2012 at 13:35

4 Mock every dependency your unit test touches. This explains everything. – Teoman shipahi Jan 21, 2015 at 17:02

13 TL;DR: Mock every dependency your unit test touches. - this is not really a great approach, mockito itself says - don't mock everything. (downvoted) – Praveen Tiwari Jan 8, 2019 at 9:44

78

Rule of thumb:

If the function you are testing needs a complicated object as a parameter, and it would be a pain to simply instantiate this object (if, for example it tries to establish a TCP connection), use a mock.

Share Follow

edited Mar 6, 2014 at 2:57

answered Sep 1, 2008 at 21:04

Orion Edwards

123k ● 66 ● 245 ● 339

I agree, but a small nitpick regarding the terminology: "... use a test double." would be more accurate. Mocks test specific interactions, whereas other kinds of test doubles (stubs, fakes, dummies) are used to simplify the test setup. – Raimund Krämer Jul 7 at 10:34

7

You should mock an object when you have a dependency in a unit of code you are trying to test that needs to be "just so".

For example, when you are trying to test some logic in your unit of code but you need to get something from another object and what is returned from this dependency might affect what you are trying to test - mock that object.

A great podcast on the topic can be found here

Share Follow

edited Jan 29, 2014 at 23:07

palacsint

28.8k ● 11 ● 84 ● 110

answered Sep 1, 2008 at 17:38

Toran Billups

27.4k ● 41 ● 158 ● 272

4 The link now routes to the current episode, not the intended episode. Is the intended podcast this hanselminutes.com/32/mock-objects? – C Perkins Jun 10, 2020 at 16:34

4

1. Mock the interfaces that you don't own. When your code calls out to a third-party service, you should mock that service. This allows your test to exercise the calling code (which you do own) without being coupled to the performance characteristics of other services. It also allows you to separate issues in your application from issues in other applications. For

example, you might use a tool like WireMock to represent http-based services; and use an in-memory data store to represent an actual database.

2. Mock the services that would cause collisions among your tests. Avoid writing individual tests that are required to run sequentially, by mocking the common services where collisions would occur. This is another reason for an in-memory data store that can spin up instances in parallel: the effects of multiple tests can collide in a single database, making the output inconsistent and unpredictable. Strive for tests that can run in parallel, even if you don't actually run them in parallel today. It gives you the option to drastically improve performance in the future with a simple configuration change.
3. Mock the services whose real implementations would cause your tests to run unacceptably slowly. The slower the tests, the less often they will be run. The less often the tests run, the less feedback the developers receive. The less feedback the developers receive, the higher the risk of poor code quality.
4. Avoid mocking your own internal services. Mocking your own services couples your tests to the current implementation details of those services (including class names and method signatures, etc.). You want to be able to make minor refactorings to your codebase (such as renaming classes and methods) without breaking tests. If refactoring is easy,

developers will be encouraged to do more of it, which will increase the quality of the code over time.

5. Bonus mocking advice: mock the interfaces that are unstable or unknown. Avoid constantly rewriting tests by mocking interfaces that are under active development or even hypothetical. Mocking can facilitate a proof-of-concept.

In summary, if your tests can run quickly, consistently, and together in parallel, you're doing a good job of mocking. If your codebase can support minor refactoring without breaking those tests, you're doing a great job of mocking. That may mean you're doing very little mocking at all!

Share Follow

edited May 25, 2023 at 2:55

answered May 25, 2023 at 2:44

jaco0646

17k ● 10 ● 67 ● 95

1 This fine answer has been underappreciated. – tchrist Jul 11, 2023 at 0:29

1

TL;DR: in mainly depends on the testing style you think is more appropriate and on the characteristics of the specific method you are testing (e.g. if it depends on an object that is very difficult to instantiate). As a general rule, some developers prefer using mocks for all

dependencies and trying to verify the correct behavior (the functions calls it makes) of the method, while others prefer using real objects and trying to verify the correct state (return values). This choice has its own implications in terms of, mainly, test isolation and implementation coupling.

Depending on the testing style

Independently of if a given test should be called a unit test or not (more on this below), the most important decision is whether to use mocks (or any other test doubles) or not when testing a given SUT (system under test). In general, this decision should be made depending on the testing style you or your team have decided to follow for your project. In TDD (Test Driven Development), there are mainly two styles for testing (following the classification and terminology used in Martin's Fowlers blog):

- **Classical TDD**: advocates not using test doubles for collaborators (i.e. methods or objects the SUT depends on) as a general rule, always trying to use the real objects for tests and focusing on the state of the SUT to verify its correct implementation (e.g. check that the returned value, given certain inputs, is correct) - also called state verification.
- **Mockist TDD**: advocates always using mocks for collaborators and focusing on behavior of the SUT to verify its correct implementation (e.g. check that the

calls to collaborators, given certain inputs, are correct) - also called behavior verification.

These are the general principles for each style. However, depending on the nature of the SUT and of its collaborators, it is common to make exceptions (more on this below). Also, note that it is also possible to adopt a more in-between posture or to follow other development processes apart from TDD, but I think this distinction is quite illustrative.

Tradeoffs

Each style has its own advantages and disadvantages:

- **Classical TDD**: its main disadvantage is bad test isolation - if a method that is highly used fails, it can cause many tests to fail, making it difficult to spot the root error (this is the problem of not having 'real' unit tests for all methods). Other disadvantages are that it can require complex fixture setups and that the tests will tend to take longer to run (using mocks is faster than actually using the real objects).
- **Mockist TDD**: its main disadvantage is high implementation coupling. Tests are more coupled to the implementation of a method rather to its correctness, as changing the nature of calls to collaborators usually causes a behavior-verification test to break, even if the end results (states) don't change and keep being correct.

Apart from these tradeoffs, the choice of testing style also impacts the coding workflow (how features are implemented along the tests) and the coding style (e.g., mockist TDD developers ease away from methods that return values in favor of methods that act upon an object that is passed as an argument, as it makes it easier to test them).

Depending on the characteristics of the SUT

As I said, for each style, it is common to make exceptions for certain SUTs.

Reasons for using test doubles in classic TDD

- **It is hard to use real collaborators in tests**: when collaborators are hard to use as real objects, it is more appropriate to use test doubles. The appropriate test double type depends on the characteristics of the SUT:
    - Stub / fake: when the SUT is better verified by its state - it is more natural to describe the function in terms of its expected output (its state after exercising it) rather than the actions (functions calls) it should perform. *Example*: to test a method that makes a simple database query and then makes complex computations based on it to return an end result, a stub could be

used as a replacement to the database call, verifying if the result returned by the SUT is correct given certain values the database query could return.

    - Mock: when the SUT is better verified by its behavior - it is more natural to describe the SUT in terms of the actions (function calls and passed arguments) it should perform (i.e. in terms of its interactions). *Example*: to test a method that should send invitations via email using an external email service, it is better to use mocks and verify that the method makes the correct calls to the mocked email service.

- **It is hard to test by verifying its state**: when it is awkward/hard to verify the correct implementation of the SUT by verifying its behavior, even if it isn't difficult to use real objects for its collaborators. In this case, it can be more appropriate to use behavior verification and test doubles. *Example*: in caching methods, you can't tell from the cache state whether it hit or missed. Behavior verification with mocks would be the best choice.
- **One of the downsides of classic TDD is very acute**: the disadvantages of classic TDD (bad test isolation, complex fixture setups, slower tests) can push developers to use test doubles instead of real objects for certain systems which are specially affected by one (or various) of these. Example: if there is a method that often fails during testing, but it is difficult to find the root problem when it does

because it has many dependencies, creating unit tests for it can be a wise chose (in addition or replacement of its existing integration tests). This would achieve better test isolation and would help when trying to find the system that is causing the errors. This can be achieved by mocking its collaborators and performing behavior verification for it, or by using other types of tests doubles.

Reasons for using real objects in mockist TDD

- **It is hard to test by verifying its behavior**: when it is awkward/hard to verify the correct implementation of the SUT by verifying its behavior (its function calls to collaborators). This is common in methods that invoke collaborators iteratively or recursively in contrast with simpler control-flows (e.g. sorting algorithms, graph algorithms…). In this case, it can be more appropriate to use state verification and real objects. *Example*: the sorting example provided by @Jan Soltis.
- **One of the downsides of mockist TDD is very acute**: mainly when you do not want the test to be too coupled with the implementation, especially for those systems whose implementations are highly likely to change often.

Regarding unit tests

Finally, I would like to address the question in terms of the terminology (as the question you have linked raises this doubt in the context of unit tests). As @user1228 said, if you do not mock all collaborators (or use any other type of test doubles for them), the test cannot be called a unit test. The definition of unit, however, can be relaxed in some cases to refer to all methods that form a fairly low-level and loosely coupled abstraction / cohesive unit instead of strictly to refer to a single method (as @rogerio commented). However, discussing when a certain test should be called a unit or integration test is not that important. Sometimes it is appropriate to only verify if a certain SUT works correctly with integration tests and no unit tests (specially for high level ones which are more naturally described / verified by their state or output) depending on the testing style you favor.

Share Follow

edited Sep 26, 2023 at 7:25

answered Sep 25, 2023 at 23:40

jaimeaz

21 ● 4