Mocks or real classes? [duplicate]

Asked 16 years, 2 months ago Modified 15 years, 6 months ago

Viewed 4k times

13

This question already has answers here:

When should I mock? (6 answers)

Closed 11 years ago.

Classes that use other classes (as members, or as arguments to methods) need instances that behave properly for unit test. If you have these classes available and they introduce no additional dependencies, isn't it better to use the real thing instead of a mock?

unit-testing mocking

Share

Improve this question

Follow

asked Oct 7, 2008 at 21:04

andreas buykx

12.9k ● 11 ● 64 ● 76

11 Answers

Sorted by: Highest score (default)

I say use real classes whenever you can.

24

I'm a big believer in expanding the boundaries of "unit" tests as much as possible. At this point they aren't really unit tests in the traditional sense, but rather just an automated regression suite for your application. I still practice TDD and write all my tests first, but my tests are a little bigger than most people's and my green-red-green cycles take a little longer. But now that I've been doing this for a little while I'm completely convinced that unit tests in the traditional sense aren't all they're cracked up to be.

In my experience writing a bunch of tiny unit tests ends up being an impediment to refactoring in the future. If I have a class A that uses B and I unit test it by mocking out B, when I decide to move some functionality from A to B or vice versa all of my tests and mocks have to change. Now if I have tests that verify that the end to end flow through the system works as expected then my tests actually help me to identify places where my refactorings might have caused a change in the external behavior of the system.

The bottom line is that mocks codify the contract of a particular class and often end up actually specifying some of the implementation details too. If you use mocks extensively throughout your test suite your code base ends up with a lot of extra inertia that will resist any future refactoring efforts.

answered Oct 7, 2008 at 21:38

Mike Deck

2 I also like such approach, but 1. That is not a UNIT test, but seems like INTEGRATION test; 2. I like, but I understand that this is not a best approach. – Budda Jun 6, 2010 at 18:01

5 "I like, but I understand that this is not a best approach." According to whom? To a certain extent it depends on what your goals are. If you're trying to write very small tests that isolate a particular method so that you can wrap your head around the best way to implement that one unit then mocks are helpful, but if you're trying to build a regression suite that will codify the behavior of your system and alert you when it changes unexpectedly, it's been my experience that extensive mock usage is definitely not the best approach. – Mike Deck Jun 7, 2010 at 14:07

yes to "unit tests in the traditional sense aren't all they're cracked up to be" – Kemoda Jun 18, 2013 at 12:40

5

It is fine to use the "real thing" as long as you have absolute control over the object. For example if you have an object that just has properties and accessors you're probably fine. If there is logic in the object you want to use, you could run into problems.

If a unit test for class a uses an instance of class b and an change introduced to b breaks b, then the tests for class a are also broken. This is where you can run into problems where as with a mock object you could always return the correct value. Using "the real thing" Can kind of convolute tests and hide the real problem.

Mocks can have downsides too, I think there is a balance with some mocks and some real objects you will have to find for yourself.

Share Improve this answer Follow

answered Oct 7, 2008 at 21:09

Steve g

2,489 ● 17 ● 17

> Changing the class API when you refactor class b can also break the tests for class a since class a will change also, so you have 2 test classes that will break when you change class b. It does not matter if you use mocks or the real thing really, as long as you watch out for side effects (IO, DB etc). You can't really create "isolated" unit tests, there will always be a chain reaction when you refactor/change the code under test. – Geo C. Jul 17, 2020 at 8:50

5

There is **one really good reason** why you want to use stubs/mocks instead of real classes. I.e. to make your unit test's (pure unit test) class under test **isolated** from everything else. This property is extremely useful and the benefits for keeping tests isolated are plentiful:

- Tests run faster because they don't need to call the real class implementation. If the implementation is to run against file system or relational database then the tests will become sluggish. Slow tests make developers not run unit tests as often. If you're doing Test Driven Development then time hogging tests are together a devastating waste of developers time.

- It will be easier to track down problems if the test is isolated to the class under test. In contrast to a system test it will be much more difficult to track down nasty bugs that are not apparently visible in stack traces or what not.
- Tests are less fragile on changes done on external classes/interfaces because you're purely testing the class that is under test. Low fragility is also an indication of low coupling, which is a good software engineering.
- You're testing against external behaviour of a class rather than the internal implementation which is more useful when deciding code design.

Now if you want to use real class in your test, that's fine but then it is **NOT** a unit test. You're doing a integration test instead, which is useful for the purpose of **validating requirements and overall sanity check**. Integration tests are not run as often as unit tests, in practice it is mostly done before committing to favorite code repository, but is equally important.

The only thing you need to have in mind is the following:

- **Mocks and stubs** are for unit tests.
- **Real classes** are for integration/system tests.

answered Oct 8, 2008 at 9:07

Spoike

122k ● 45 ● 142 ● 158

Sometimes it's much simpler to use the real thing in your tests than to mock/stub all the methods and do a shallow copy of the API. Side effects should live in a special layer. The tests for that layer will mostly be integration tests. They key part is to not deal with functionality that could bring side effects to your tests in your core layers. I use hexagonal architecture so it's easy to keep IO and the like in an outer layer. Also the slow argument is not valid since mocking/stubing will use reflections which may be in many cases more expensive than the real thing. – Geo C. Jul 17, 2020 at 8:38

4

Extracted and extended from an answer of mine How do I unit-test inheriting objects?">here:

You should always use real objects where possible.

You should only use mock objects if the real objects do something you dont want to set up (like use sockets, serial ports, get user input, retrieve bulky data etc). Essentially, mock objects are for when the estimated effort to implement and maintain a test using a real object is greater than that to implement and maintain a test using a mock object.

I dont buy into the "dependant test failure" argument. If a test fails because a depended-on class broke, the test did exactly what it should have done. This is not a smell! If a depended-on interface changes, I want to know!

Highly mocked testing environments are very high-maintenance, particularly early in a project when

interfaces are in flux. Ive always found it better to start integration testing ASAP.

Share Improve this answer

Follow

answered Oct 8, 2008 at 9:37

metao

2,844 ● 1 ● 19 ● 16

> "If a test fails because a depended-on class broke, the test did exactly what it should have done." Exactly. The problem is when you have dozens of parts of your code where you mocked the behavior. You will have to go in and modify the mock's behavior to fit the new class's behavior whenever it changes. – Henry Obiaraije Dec 11, 2022 at 11:43

3

I always use a mock version of a dependency if the dependency accesses an external system like a database or web service.

If that isn't the case, then it depends on the complexity of the two objects. Testing the object under test with the real dependency is essentially multiplying the two sets of complexities. Mocking out the dependency lets me isolate the object under test. If either object is reasonably simple, then the combined complexity is still workable and I don't need a mock version.

As others have said, defining an interface on the dependency and injecting it into the object under test makes it much easier to mock out.

Personally, I'm undecided about whether it's worth it to use strict mocks and validate every call to the dependency. I usually do, but it's mostly habit.

You may also find these related questions helpful:

- What is object mocking and when do I need it?
- When should I mock?
- How are mocks meant to be used?
- And perhaps even, Is it just me, or are interfaces overused?

Share Improve this answer

Follow

edited May 23, 2017 at 12:01

Community Bot

1 • 1

answered Oct 8, 2008 at 1:08

Don Kirkby

56.5k • 27 • 219 • 301

2

Use the real thing only if it has been unit tested itself first. If it introduces dependencies that prevent that (circular dependencies or if it requires certain other measures to be in place first) then use a 'mock' class (typically referred to as a "stub" object).

Share Improve this answer

Follow

answered Oct 7, 2008 at 21:19

Ron Warholic

10.1k • 33 • 47

1 Aren't mock classes and stub objects different things? – Oddmund Oct 8, 2008 at 1:11

1 There may be a debate about sematics out there. I like Fowler's distinctions in his paper: martinfowler.com/articles/mocksArentStubs.html The gist is that stubs simply serve up a pre-determined return value, and that mocks are more for verifying that objects are interacting as expected. – Michael Lang Oct 10, 2008 at 0:10

2

If your 'real things' are simply value objects like JavaBeans then thats fine.

For anything more complex I would worry as mocks generated from mocking frameworks can be given precise expectations about how they will be used e.g. the number of methods called, the precise sequence and the parameters expected each time. Your real objects cannot do this for you so you risk losing depth in your tests.

Share Improve this answer

Follow

answered Oct 7, 2008 at 21:35

Garth Gilmour

11.2k ● 5 ● 28 ● 36

2

I've been very leery of mocked objects since I've been bitten by them a number of times. They're great when you want isolated unit tests, but they have a couple of issues. The major issue is that if the Order class needs a a collection of OrderItem objects and you mock them, it's almost impossible to verify that the behavior of of the

mocked OrderItem class matches the real-world example (duplicating the methods with appropriate signatures is generally not enough). More than once I've seen systems fail because the mocked classes don't match the real ones and there weren't enough integration tests in place to catch the edge cases.

I generally program in dynamic languages and I prefer merely overriding the specific methods which are problematic. Unfortunately, this is sometimes hard to do in static languages. The downside of this approach is that you're using integration tests rather than unit tests and bugs are sometimes harder to track down. The upside is that you're using the actual code that is written, rather than a mocked version of that code.

Share Improve this answer

Follow

answered Oct 8, 2008 at 8:50

Ovid

11.7k ● 9 ● 49 ● 76

1

If you don't care for verifying expectations on how your UnitUnderTest should interact with the Thing, and interactions with the RealThing have no other side-effects (or you can mock these away) then it is in my opinion perfectly fine to just let your UnitUnderTest use the RealThing.

That the test then covers more of your code base is a bonus.

I generally find it is easy to tell when I should use a ThingMock instead of a RealThing:

- When I want to verify expectations in the interaction with the Thing.
- When using the RealThing would bring unwanted side-effects.
- Or when the RealThing is simply too hard/troublesome to use in a test setting.

Share Improve this answer

Follow

answered Oct 7, 2008 at 22:12

Chris Vest

8,672 ● 3 ● 38 ● 43

0

If you write your code in terms of interfaces, then unit testing becomes a joy because you can simply inject a fake version of any class into the class you are testing.

For example, if your database server is down for whatever reason, you can still conduct unit testing by writing a fake data access class that contains some cooked data stored in memory in a hash map or something.

Share Improve this answer

Follow

answered Oct 7, 2008 at 21:40

moffdub

5,314 ● 2 ● 37 ● 32

It depends on your **coding style**, **what** you are doing, your **experience** and other things.

Given all that, there's nothing stopping you from **using both**.

I know I use the term **unit test** way too often. Much of what I do might be better called integration test, but better still is to **just think of it as testing**.

So I suggest using all the testing techniques where they fit. The overall aim being to **test well**, take **little time** doing it and personally have a **solid feeling that it's right**.

Having said that, depending on how you program, you might want to consider using techniques (like interfaces) that make mocking less intrusive a bit more often. But don't use Interfaces and injection where it's wrong. Also if the mock needs to be fairly complex there is probably less reason to use it. (You can see a lot of good guidance, in the answers here, to what fits when.)

Put another way: No answer works always. Keep your wits about you, observe what works what doesn't and why.

edited Oct 9, 2008 at 23:37

answered Oct 9, 2008 at 23:22

Mike Gale
311 ● 2 ● 8