How are Mocks meant to be used?

Asked 16 years, 3 months ago Modified 14 years, 5 months ago

Viewed 3k times

24

When I originally was introduced to Mocks I felt the primary purpose was to mock up objects that come from external sources of data. This way I did not have to maintain an automated unit testing test database, I could just fake it.

But now I am starting to think of it differently. I am wondering if Mocks are more effective used to completely isolate the tested method from anything outside of itself. The image that keeps coming to mind is the backdrop you use when painting. You want to keep the paint from getting all over everything. I am only testing that method, and I only want to know how it reacts to these faked up external factors?

It seems incredibly tedious to do it this way but the advantage I am seeing is when the test fails it is because it is screwed up and not 16 layers down. But now I have to have 16 tests to get the same testing coverage because each piece would be tested in isolation. Plus each test becomes more complicated and more deeply tied to the method it is testing.

It feels right to me but it also seems brutal so I kind of want to know what others think.

unit-testing mocking

Share

Improve this question

Follow

edited Oct 17, 2008 at 14:17

Rontologist

3,558 ● 1 ● 22 ● 24

asked Sep 12, 2008 at 14:59

Flory

2,849 ● 20 ● 31

1 It is pretty tedious, but you should be reusing your mock setup code just like you would production code. That will ease the pain somewhat. – jodonnell Sep 12, 2008 at 15:41

It can be made less tedious with tools like EasyMock. Also some of Gishu's objections don't pan out if you organize your mocks correctly. Google this: "Mocks Aren't Stubs" – Justin Standard Sep 12, 2008 at 16:33

Let me see if I can write up a blog post.. articulating my feelings a bit better :) Watch this space... – Gishu Sep 13, 2008 at 7:00

9 Answers

Sorted by: Highest score (default)

20

I recommend you take a look at Martin Fowler's article *Mocks Aren't Stubs* for a more authoritative treatment of Mocks than I can give you.

The purpose of mocks is to unit test your code in isolation of dependencies so you can truly test a piece of code at

the "unit" level. The code under test is the real deal, and every other piece of code it relies on (via parameters or dependency injection, etc) is a "Mock" (an empty implementation that always returns expected values when one of its methods is called.)

Mocks may seem tedious at first, but they make Unit Testing far easier and more robust once you get the hang of using them. Most languages have Mock libraries which make mocking relatively trivial. If you are using Java, I'll recommend my personal favorite: EasyMock.

Let me finish with this thought: you need integration tests too, but having a good volume of unit tests helps you find out which component contains a bug, when one exists.

Share Improve this answer Follow

edited Sep 12, 2008 at 15:28

answered Sep 12, 2008 at 15:20

Justin Standard

21.5k ● 22 ● 82 ● 90

I am halfway through the Growing Object Oriented Software book. The approach is something that was new to me and I gave it a try and was pleased with the results.. However the approach is not what is prevalent in the large. The mock-based approach is more prone to abuse if I may say so. My blog post is way over due:) maybe as soon as I am done with the GOOS book, I'll finally get to it. Thanks for challenging my opinions and driving me to learn more... – Gishu Jul 19, 2010 at 5:48

17

Don't go down the dark path Master Luke. :) Don't mock everything. You could but you shouldn't... here's why.

- If you continue to test each method in isolation, you have surprises and work cut out for you when you bring them all together ala the **BIG BANG**. We build objects so that they can work together to solve a bigger problem.. By themselves they are insignificant. You **need to know if all the collaborators are working as expected**.
- Mocks **make tests brittle** by introducing duplication - Yes I know that sounds alarming. For every mock expect you setup, there are n places where your method signature exists. The actual code and your mock expectations (in multiple tests). Changing actual code is easier... updating all the mock expectations is tedious.
- Your **test is now privy to insider implementation information**. So your test depends on how you chose to implement the solution... bad. Tests should be a independent spec that can be met by multiple solutions. I should have the freedom to just press delete on a block of code and reimplement **without** having to rewrite the test suite.. coz the requirements still stay the same.

To close, I'll say "If it quacks like a duck, walks like a duck, then it probably is a duck" - If it feels wrong.. it probably is. *Use mocks to abstract out problem children

like IO operations, databases, third party components and the like.. Like salt, some of it is necessary.. too much and :x *
This is the holy war of State based vs Iteraction based testing.. Googling will give you deeper insight.

Clarification: I'm hitting some resistance w.r.t. integration tests here :) So to clarify my stand..

- Mocks do not figure in the 'Acceptance tests'/Integration realm. You'll only find them in the Unit Testing world.. and that is my focus here.
- Acceptance tests are different and are **very much** needed - not belittling them. But Unit tests and Acceptance tests are different and should be kept different.
- All collaborators within a component or package do not need to be isolated from each other.. Like micro-optimization that is Overkill. They exist to solve a problem **together**.. cohesion.

edited Sep 12, 2008 at 15:33

answered Sep 12, 2008 at 15:14

Gishu
137k ● 47 ● 226 ● 311

This the difference between unit tests and integration tests. You need both. I shouldn't need a database to test how my code reacts to an incoming record set, but I do need to also

test that my SQL brings back the right result. – jodonnell Sep 12, 2008 at 15:18

1 So you need a SQLDataAccessLayer like component. Unit tests for those have to hit the DB because DB-access is its reason for being. But components accessing IDataAccessLayer need to mock to keep their tests running sub-second fast. Testing end to end is more of in the realm of acceptance tests-FIT – Gishu Sep 12, 2008 at 15:27

What you are saying about not testing everything in isolation is incorrect. You must do unit testing AND integration testing. When you do, and your integration test fails or a bug is encountered, isolated unit tests quickly point to the problem by eliminating tested code which is not suspect. – Justin Standard Sep 12, 2008 at 15:27

As usual, moderatin is key. I think the tradeoff on how "isolated" you make your unit tests is always about ROI. Good answer Gishu, would be great to see you blog about these OOAD/SE topics. – Troy DeMonbreun Sep 12, 2008 at 15:42

1 Fallacy: Fallacy is spelled with two "l"s. ;-) – Epaga Oct 24, 2008 at 14:32

2

Yes, I agree. I see mocking as sometimes painful, but often necessary, for your tests to truly become **unit** tests, i.e. only the smallest unit that you can make your test concerned with is under test. This allows you to eliminate any other factors that could potentially affect the outcome of the test. You do end up with a lot more small tests, but it becomes so much easier to work out where a problem is with your code.

Share Improve this answer

Follow

answered Sep 12, 2008 at 15:05

Sam Wessel

8,848 ● 8 ● 42 ● 44

1

My philosophy is that you should write testable code to fit the tests,
not write tests to fit the code.

As for complexity, my opinion is that tests should be simple to write, simply because you write more tests if they are.

I might agree that could be a good idea if the classes you're mocking doesn't have a test suite, because if they did have a proper test suite, you would know where the problem is without isolation.

Most of them time I've had use for mock objects is when the code I'm writing tests for is so tightly coupled (read: bad design), that I have to write mock objects when classes they depend on is not available. Sure there are valid uses for mock objects, but if your code **requires** their usage, I would take another look at the design.

Share Improve this answer

Follow

answered Sep 12, 2008 at 15:17

asksol

19.5k ● 5 ● 64 ● 68

Yes, that is the downside of testing with mocks. There is a lot of work that you need to put in that it feels brutal. But

1

that is the essence of unit testing. How can you test something in isolation if you don't mock external resources?

On the other hand, you're mocking away slow functionality (such as databases and i/o operations). If the tests run faster then that will keep programmers happy. There is nothing much more painful than waiting for really slow tests, that take more than 10 seconds to finish running, while you're trying to implement one feature.

If every developer in your project spent time writing unit tests, then those 16 layers (of indirection) wouldn't be that much of a problem. Hopefully you should have that test coverage from the beginning, right? :)

Also, don't forget to write a function/integration test between objects in collaboration. Or else you might miss something out. These tests won't need to be run often, but are still important.

Share Improve this answer

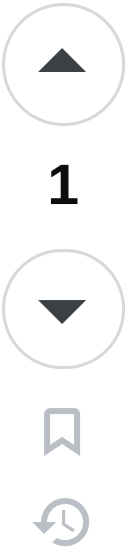
Follow

answered Sep 12, 2008 at 15:27

Spoike

122k ● 45 ● 142 ● 158

> The slow tests you are talking about are integration tests. This is why you should keep your unit and integration tests separate (though they often rely on the same testing tools). Run unit tests very often because they are fast. Run integration tests at intervals to get a reality check.
> – Justin Standard Sep 14, 2008 at 6:33



On one scale, yes, mocks are meant to be used to simulate external data sources such as a database or a web service. On a more finely grained scale however if you're designing loosely coupled code then you can draw lines throughout your code almost arbitrarily as to what might at any point be an 'outside system'. Take a project I'm working on currently:

When someone attempts to check in, the **CheckInUi** sends a **CheckInInfo** object to a **CheckInMediator** object which validates it using a **CheckInValidator**, then if it is ok, it fills a domain object named **Transaction** with **CheckInInfo** using **CheckInInfoAdapter** then passes the **Transaction** to an instance of **ITransactionDao.SaveTransaction()** for persistence.

I am right now writing some automated *integration tests* and obviously the **CheckInUi** and **ITransactionDao** are windows unto external systems and they're the ones which should be mocked. However, whose to say that at some point **CheckInValidator** won't be making a call to a web service? That is why when you write *unit tests* you assume that everything other than the specific functionality of your class is an external system. Therefore in my unit test of **CheckInMediator** I mock out all the objects that it talks to.

EDIT: Gishu is technically correct, not everything needs to be mocked, I don't for example mock **CheckInInfo** since it is simply a container for data. However anything that you could ever see as an external service (and it is

almost anything that transforms data or has side-effects) should be mocked.

An analogy that I like is to think of a properly loosely coupled design as a field with people standing around it playing a game of catch. When someone is passed the ball he might throw a completely different ball to the next person, he might even throw a multiple balls in succession to different people or throw a ball and wait to receive it back before throwing it to yet another person. It is a strange game.

Now as their coach and manager, you of course want to check how your team works as a whole so you have team practice (integration tests) but you also have each player practice on his own against backstops and ball-pitching machines (unit tests with mocks). The only piece that this picture is missing is mock expectations and so we have our balls smeared with black tar so they stain the backstop when they hit it. Each backstop has a 'target area' that the person is aiming for and if at the end of a practice run there is no black mark within the target area you know that something is wrong and the person needs his technique tuned.

Really take the time to learn it properly, the day I understood Mocks was a huge a-ha moment. Combine it with an inversion of control container and I'm never going back.

On a side note, one of our IT people just came in and gave me a free laptop!

Share Improve this answer

Follow

answered Sep 12, 2008 at 15:43

George Mauer

122k ● 139 ● 395 ● 626

1

As someone said before, if you mock everything to isolate more granular than the class you are testing, you give up enforcing cohesion in you code that is under test.

Keep in mind that mocking has a fundamental advantage, behavior verification. This is something that stubs don't provide and is the other reason that makes the test more brittle (but can improve code coverage).

Share Improve this answer

Follow

answered Sep 23, 2008 at 15:39

Korbin

1,808 ● 2 ● 19 ● 29

1

Mocks were invented in part to answer the question: How would you unit test objects if they had no getters or setters?

These days, recommended practice is to mock roles not objects. Use Mocks as a design tool to talk about collaboration and separation of responsibilities, and not as "smart stubs".

Share Improve this answer

Follow

answered Oct 18, 2008 at 14:40

keithb

103 ● 4

0

Mock objects are 1) often used as a means to isolate the code under test, BUT 2) as keithb already pointed out, are important to "focus on the relationships between collaborating objects". This article gives some insights and history related to the subject: Responsibility Driven Design with Mock Objects.

Share Improve this answer

Follow

answered Jul 18, 2010 at 16:27

koen

13.7k ● 10 ● 48 ● 51