Unit testing code with a file system dependency

Asked 16 years, 3 months ago Modified 8 years ago Viewed 63k times

161

I am writing a component that, given a ZIP file, needs to:

1. Unzip the file.
2. Find a specific dll among the unzipped files.
3. Load that dll through reflection and invoke a method on it.

I'd like to unit test this component.

I'm tempted to write code that deals directly with the file system:

```
void DoIt()
{
   Zip.Unzip(theZipFile, "C:\\foo\\Unzipped");
   System.IO.File myDll = File.Open("C:\\foo\\Unzipped\\SuperSecret.bar");
   myDll.InvokeSomeSpecialMethod();
}
```

But folks often say, "Don't write unit tests that rely on the file system, database, network, etc."

If I were to write this in a unit-test friendly way, I suppose it would look like this:

```
void DoIt(IZipper zipper, IFileSystem fileSystem, IDllRunner runner)
{
   string path = zipper.Unzip(theZipFile);
   IFakeFile file = fileSystem.Open(path);
   runner.Run(file);
}
```

Yay! Now it's testable; I can feed in test doubles (mocks) to the DoIt method. But at what cost? I've now had to define 3 new interfaces just to make this testable. And what, exactly, am I testing? I'm testing that my DoIt function properly interacts with its dependencies. It doesn't test that the zip file was unzipped properly, etc.

It doesn't feel like I'm testing functionality anymore. It feels like I'm just testing class interactions.

My question is this: what's the proper way to unit test something that is dependent on the file system?

edit I'm using .NET, but the concept could apply Java or native code too.

unit-testing dependency-injection dependencies

Share
Improve this question
Follow

edited Dec 7, 2016 at 13:06
Lii
12.1k ● 9 ● 68 ● 89

asked Sep 24, 2008 at 18:46
Judah Gabriel Himango
60k ● 39 ● 161 ● 215

9 Folks say don't write to the file system in a unit test because if you're tempted to write to the file system you aren't understanding what constitutes a unit test. A unit test usually interacts with a single *real* object (the unit under test) and all the other dependencies are mocked and passed in. The test class then consists of test methods that validate the logical paths through the object's methods and ONLY the logical paths in the unit under test. – Christopher Perry Oct 24, 2013 at 0:59

1 in your situation the only part that needs unit testing would be `myDll.InvokeSomeSpecialMethod();` where you would check that it works correctly in both success and fail situations so i wouldn't unit test `DoIt` but `DllRunner.Run` that said misusing a UNIT test to double check that the entire process works would be an acceptable misuse and as it would be an integration test masquerading a unit test the normal unit test rules don't need to be applied as strictly – MikeT Apr 9, 2018 at 12:46

11 Answers

Sorted by: Highest score (default)

77

> Yay! Now it's testable; I can feed in test doubles (mocks) to the DoIt method. But at what cost? I've now had to define 3 new interfaces just to make this testable. And what, exactly, am I testing? I'm testing that my DoIt function properly interacts with its dependencies. It doesn't test that the zip file was unzipped properly, etc.

You have hit the nail right on its head. What you want to test is the logic of your method, not necessarily whether a true file can be addressed. You don´t need to test (in this unit test) whether a file is correctly unzipped, your method takes that for granted. The interfaces are valuable by itself because they provide abstractions that you can program against, rather than implicitly or explicitly relying on one concrete implementation.

Share Improve this answer Follow

answered Nov 2, 2010 at 7:15
andreas buykx
12.9k ● 11 ● 64 ● 76

15 The testable `DoIt` function as stated doesn't even need testing. As the questioner rightly pointed out there's nothing of significance left to test. Now it's the implementation of

`IZipper`, `IFileSystem`, and `IDllRunner` that needs testing, but they are the very things that have been mocked out for the test! – Ian Goldby Feb 21, 2014 at 15:02

71

Your question exposes one of the hardest parts of testing for developers just getting into it:

"What the hell do I test?"

Your example isn't very interesting because it just glues some API calls together so if you were to write a unit test for it you would end up just asserting that methods were called. Tests like this tightly couple your implementation details to the test. This is bad because now you have to change the test every time you change the implementation details of your method because changing the implementation details breaks your test(s)!

Having bad tests is actually worse than having no tests at all.

In your example:

```
void DoIt(IZipper zipper, IFileSystem fileSystem, IDllRunner runner)
{
   string path = zipper.Unzip(theZipFile);
   IFakeFile file = fileSystem.Open(path);
   runner.Run(file);
}
```

While you can pass in mocks, there's no logic in the method to test. If you were to attempt a unit test for this it might look something like this:

```
// Assuming that zipper, fileSystem, and runner are mocks
void testDoIt()
{
  // mock behavior of the mock objects
  when(zipper.Unzip(any(File.class)).thenReturn("some path");
  when(fileSystem.Open("some path")).thenReturn(mock(IFakeFile.class));

  // run the test
  someObject.DoIt(zipper, fileSystem, runner);

  // verify things were called
  verify(zipper).Unzip(any(File.class));
  verify(fileSystem).Open("some path"));
  verify(runner).Run(file);
}
```

Congratulations, you basically copy-pasted the implementation details of your `DoIt()` method into a test. Happy maintaining.

When you write tests you want to test the *WHAT* and not the *HOW*. See Black Box Testing for more.

The **WHAT** is the name of your method (or at least it should be). The **HOW** are all the little implementation details that live inside your method. Good tests allow you to swap out the **HOW** without breaking the **WHAT**.

Think about it this way, ask yourself:

"If I change the implementation details of this method (without altering the public contract) will it break my test(s)?"

If the answer is yes, you are testing the **HOW** and not the **WHAT**.

To answer your specific question about testing code with file system dependencies, let's say you had something a bit more interesting going on with a file and you wanted to save the Base64 encoded contents of a `byte[]` to a file. You can use streams for this to test that your code does the right thing without having to check **how** it does it. One example might be something like this (in Java):

```
interface StreamFactory {
    OutputStream outStream();
    InputStream inStream();
}

class Base64FileWriter {
    public void write(byte[] contents, StreamFactory streamFactory) {
        OutputStream outputStream = streamFactory.outStream();
        outputStream.write(Base64.encodeBase64(contents));
    }
}

@Test
public void save_shouldBase64EncodeContents() {
    OutputStream outputStream = new ByteArrayOutputStream();
    StreamFactory streamFactory = mock(StreamFactory.class);
    when(streamFactory.outStream()).thenReturn(outputStream);

    // Run the method under test
    Base64FileWriter fileWriter = new Base64FileWriter();
    fileWriter.write("Man".getBytes(), streamFactory);

    // Assert we saved the base64 encoded contents
    assertThat(outputStream.toString()).isEqualTo("TWFu");
}
```

The test uses a `ByteArrayOutputStream` but in the application (using dependency injection) the real StreamFactory (perhaps called FileStreamFactory) would return `FileOutputStream` from `outputStream()` and would write to a `File`.

What was interesting about the `write` method here is that it was writing the contents out Base64 encoded, so that's what we tested for. For your `DoIt()` method, this would be more appropriately tested with an integration test.

Share

Improve this answer

Follow

edited Dec 7, 2016 at 13:07
Lii
12.1k ● 9 ● 68 ● 89

answered Apr 17, 2014 at 4:52
Christopher Perry
39.2k ● 45 ● 147 ● 188

1 I'm not sure I agree with your message here. Are you saying that there is no need to unit test this kind of method? So you are basically saying TDD is bad? As if you do TDD then you can't write this method without writing a test first. Or are you to rely on a hunch that your method will not require a test? The reason that ALL unit testing frameworks include a "verify" feature, is that it is OK to use it. "This is bad because now you have to change the test every time you change the implementation details of your method"... welcome to the world of unit testing. – Ronnie Nov 12, 2015 at 15:28

6 You are supposed to test the CONTRACT of a method, not it's implementation. If you have to change your test every time your implementation of that contract changes then you are in for a horrible time maintaining both your app code base and the test code base. – Christopher Perry Dec 21, 2016 at 2:19

2 @Ronnie blindly applying unit testing is not helpful. There are projects of widely-varying nature, and unit testing is not effective in all of them. As an example, I'm working on a project in which 95% of the code is *about* the side-effects (note, this side-effect-heavy nature is *by requirement*, it's *essential complexity, not incidental*, since it gathers data from a wide variety of stateful sources and presents it with very little manipulation, so there's hardly any pure logic). Unit testing is *not* effective here, integration testing is. – Vicky Chijwani Feb 7, 2017 at 17:36

1 Side effects should be pushed to the edges of your system, they shouldn't be intertwined throughout the layers. At the edges you test the side effects, which are behaviors. Everywhere else you should try to have pure functions without side effects, which are easily tested and easy to reason about, reuse, and compose. – Christopher Perry Dec 20, 2017 at 19:03

Nice explanation but "***For your DoIt() method, this would be more appropriately tested with an integration test.***" assuming that you'll change your tests based on the implementation you're still in this situation when doing integration tests: "***Congratulations, you basically copy-pasted the implementation details of your DoIt() method into a test. Happy maintaining.***". So what is the point of doing integration tests if they're going to break after implementation has changed? The situation became something similar to "***Having bad tests is actually worse than having no tests at all***" – Farid Nov 5, 2021 at 12:50

56

There's really nothing wrong with this, it's just a question of whether you call it a unit test or an integration test. You just have to make sure that if you do interact with the file system, there are no unintended side effects. Specifically, make sure that you clean up after youself -- delete any temporary files you created -- and that you don't accidentally overwrite an existing file that happened to have the same filename as a temporary file you were using. Always use relative paths and not absolute paths.

It would also be a good idea to `chdir()` into a temporary directory before running your test, and `chdir()` back afterwards.

Share Improve this answer Follow

answered Sep 24, 2008 at 19:09
Adam Rosenfield
399k ●101 ●522 ●597

33 +1, however note that `chdir()` is process-wide so you might break the ability to run your tests in parallel, if your test framework or a future version of it supports that. – user23743 Oct 12, 2010 at 13:12

26

I am reticent to pollute my code with types and concepts that exist only to facilitate unit testing. Sure, if it makes the design cleaner and better then great, but I think that is often not the case.

My take on this is that your unit tests would do as much as they can which may not be 100% coverage. In fact, it may only be 10%. The point is, your unit tests should be fast and have no external dependencies. They might test cases like "this method throws an ArgumentNullException when you pass in null for this parameter".

I would then add integration tests (also automated and probably using the same unit testing framework) that can have external dependencies and test end-to-end scenarios such as these.

When measuring code coverage, I measure both unit and integration tests.

Share
Improve this answer
Follow

edited Sep 27, 2011 at 15:05
user142162

answered Sep 24, 2008 at 18:51
Kent Boogaart
179k ●37 ●398 ●395

6 Yeah, I hear you. There's this bizarre world you reach where you've decoupled so much, that all your left with is method invocations on abstract objects. Airy fluff. When you reach this point, it doesn't feel like you're really testing anything real. You're just testing interactions between classes. – Judah Gabriel Himango Sep 24, 2008 at 20:23

10 This answer is misguided. Unit testing is not like frosting, it's more like sugar. It's baked into the cake. It's part of writing your code... a design activity. Therefore, you never "pollute" your code with anything that would "facilitate testing" because the testing is what facilitates you writing your code. 99% of the time a test is hard to write because the developer wrote the code before the test, and ended up writing evil untestable code – Christopher Perry Oct 24, 2013 at 0:56

1 @Christopher: to extend your analogy, I don't want my cake to end up resembling a vanilla slice just so I can use sugar. All I'm advocating is pragmatism. – Kent Boogaart Oct 24, 2013 at 2:59

1 @Christopher: your bio says it all: "I'm a TDD zealot". I, on the other hand, am pragmatic. I do TDD where it fits and not where it doesn't - nothing in my answer suggests I don't do TDD, though you seem to think it does. And whether it's TDD or not, I won't introduce large amounts of complexity for the sake of facilitating testing. – Kent Boogaart Oct 24, 2013 at 6:16

3 @ChristopherPerry Can you explain how to solve the OP's original issue then in a TDD way? I run into this all the time; I need to write a function whose sole purpose is to perform an action with an external dependency, like in this question. So even in the write-the-test-first scenario, what would that test even be? – Dax Fohl Apr 17, 2014 at 1:20

8

There's nothing wrong with hitting the file system, just consider it an integration test rather than a unit test. I'd swap the hard coded path with a relative path and create a TestData subfolder to contain the zips for the unit tests.

If your integration tests take too long to run then separate them out so they aren't running as often as your quick unit tests.

I agree, sometimes I think interaction based testing can cause too much coupling and often ends up not providing enough value. You really want to test unzipping the file here not just verify you are calling the right methods.

Share Improve this answer Follow

answered Sep 24, 2008 at 18:57
JC.
11.8k ● 11 ● 42 ● 50

How often they run is of little concern; we use a continuous integration server that automatically runs them for us. We don't really care how long they take. If "how long to run" isn't a concern, is there any reason to distiguish between unit and integration tests? – Judah Gabriel Himango Sep 24, 2008 at 20:07

4 Not really. But if developers want to quickly run all the unit tests locally its nice to have an easy way to do that. – JC. Sep 24, 2008 at 20:40

6

One way would be to write the unzip method to take InputStreams. Then the unit test could construct such an InputStream from a byte array using ByteArrayInputStream. The contents of that byte array could be a constant in the unit test code.

Share Improve this answer Follow

answered Sep 24, 2008 at 18:49
nsayer
17k ● 4 ● 36 ● 52

Ok, so that allows for injection of the stream. Dependency injection/IOC. How about the part of unzipping the stream into files, loading a dll among those files, and calling a method in that dll? – Judah Gabriel Himango Sep 24, 2008 at 19:57

3

This seems to be more of an integration test as you are depending on a specific detail (the file system) that could change, in theory.

I would abstract the code that deals with the OS into it's own module (class, assembly, jar, whatever). In your case you want to load a specific DLL if found, so make an IDllLoader interface and DllLoader class. Have your app acquire the DLL from the DllLoader using the interface and test that .. you're not responsible for the unzip code afterall right?

Share Improve this answer Follow

answered Sep 24, 2008 at 19:00

tap
201 ●1 ●3 ●8

2

Assuming that "file system interactions" are well tested in the framework itself, create your method to work with streams, and test this. Opening a FileStream and passing it to the method can be left out of your tests, as FileStream.Open is well tested by the framework creators.

Share Improve this answer Follow

answered Sep 24, 2008 at 18:50

Sunny Milenov
22.3k ●6 ●82 ●107

You and nsayer have essentially the same suggestion: make my code work with streams. How about the part about unzipping the stream contents into dll files, opening that dll and calling a function in it? What would you do there? – Judah Gabriel Himango Sep 24, 2008 at 20:01

3 @JudahHimango. Those parts may not necessarily be testable. You can't test everything. Abstract the un-testable components into their own functional blocks, and assume that they will work. When you come across a bug with the way this block works, then devise a test for it, and voila. Unit testing does NOT mean you have to test everything. 100% code coverage is unrealistic in some scenarios. – Zoran Pavlovic Oct 26, 2012 at 14:55

1

You should not test class interaction and function calling. instead you should consider integration testing. Test the required result and not the file loading operation.

Share Improve this answer Follow

answered Sep 24, 2008 at 18:52

Dror Helper
30.7k ● 15 ● 82 ● 129

1

As others have said, the first is fine as an integration test. The second tests only what the function is supposed to actually do, which is all a unit test should do.

As shown, the second example looks a little pointless, but it does give you the opportunity to test how the function responds to errors in any of the steps. You don't have any error checking in the example, but in the real system you may have, and the dependency injection would let you test all the responses to any errors. Then the cost will have been worth it.

Share Improve this answer Follow

answered Dec 16, 2008 at 13:52
David Sykes
49.7k ● 18 ● 74 ● 81

1

For unit test I would suggest that you include the test file in your project(EAR file or equivalent) then use a relative path in the unit tests i.e. "../testdata/testfile".

As long as your project is correctly exported/imported than your unit test should work.

Share Improve this answer Follow

answered Dec 16, 2008 at 14:00
James Anderson
27.4k ● 7 ● 54 ● 80