Using a dependency injection system how do you unit test your code

Asked 15 years, 10 months ago Modified 15 years, 10 months ago

Viewed 324 times

7

As far as I can see there are two ways, both with their drawbacks.

1. Get the object you are unit testing from the dependency injection system. This is low maintenance as you don't have to manage anything when you change the framework around. But you are essentially testing the whole system from the point of view of your object, if a component fails it can blow up a lot of unit tests and it may not be apparent which one is failing.
2. is to manage the dependencies manually in the unit tests, and in some cases create test objects so that you can test each object in isolation. This keeps the unit tests discreet but dramatically increases the maintenance of the unit tests themselves. It also means that you don't pick up on bugs cause by the way the objects interact on your live system.

Is either approach right or wrong? Should a compromise be used? Has anyone had any success stories either

way.

unit-testing dependency-injection

Share

Improve this question

Follow

asked Jan 29, 2009 at 10:10

Jeremy French

12.1k ● 6 ● 47 ● 72

2 Answers

Sorted by: Highest score (default)

6

If you're writing a **unit** test you should be using mocks for your dependencies and an IoC container shouldn't come into the picture. You should instantiate your class-under-test with mocks for the dependencies injected by hand.

If you're getting your object from the IoC container already wired up then what you're writing is integration tests which are very different.

Your goal for writing a unit test should be to write your test *in isolation* from the rest of the system.

Share Improve this answer

Follow

answered Jan 29, 2009 at 10:16

tddmonkey

21.2k ● 10 ● 59 ● 69

You are right. With simple enough mocks it shouldn't hurt that much to make changes. Time for me to read up some more

on unit testing, not as simple as it seemed at first.
– Jeremy French Jan 29, 2009 at 13:03

0

You want to use an auto mocking container to manage the dependencies for you. There is one built into StructureMap and it's not that hard to roll your own.

The basic premise of an auto mocking container is that you set expectations on the dependencies you need to/care about and it creates mocks for any other dependencies for you. This means that if you add a dependency to a class you don't necessarily have to go back and alter every test for it; just those that will use the new dependency.

In my opinion the first approach is wrong as - as you you mentioned - it's not necessarily the object that you are testing that will cause a test to fail.

Share Improve this answer

Follow

answered Jan 29, 2009 at 10:13

Garry Shutler

32.7k ● 13 ● 89 ● 120