FullStack.Cafe - Kill Your Tech Interview

Q1: What is *Mocking*? ☆☆

Topics: Software Testing Unit Testing

Answer:

Mocking is primarily used in unit testing. An object under test may have dependencies on other (complex) objects. To isolate the behavior of the object you want to replace the other objects by mocks that simulate the behavior of the real objects. This is useful if the real objects are impractical to incorporate into the unit test.

In short, **mocking **is creating objects that simulate the behavior of real objects.

Q2: What is the difference between *Unit Tests* and *Functional Tests*? ☆☆

Topics: Unit Testing

Answer:

- **Unit Test** - testing an individual unit, such as a method (function) in a class, with all dependencies mocked up.
- **Functional Test** - AKA Integration Test, testing a slice of functionality in a system. This will test many methods and may interact with dependencies like Databases or Web Services.

Q3: Should unit tests be written for *Getter* and *Setters*? ☆☆

Topics: Unit Testing

Answer:

Properties (getters/setters) are good examples of code that usually doesn’t contain any logic, and doesn’t require testing. But watch out: once you add any check inside the property, you’ll want to make sure that logic is being tested.

In other words, if your getters and setters do more than just get and set (i.e. they're properly complex methods), then yes, they should be tested. But don't write a unit test case just to test a getter or setters, that's a waste of time.

Q4: How to unit test an object with *database* queries? ☆☆

Topics: Unit Testing Software Testing

Answer:

- If your objects are tightly coupled to your data layer, it is difficult to do proper unit testing. Your objects should be *persistent ignorant*. You should have a *data access layer*, that you would make requests to, that would return objects.

- Then *mock* out your calls to the database. This way, you can leave that DB dependants part out of your unit tests, test them in isolation or write *integration tests*.

Q5: How would you unit test *private* methods? ☆☆☆

Topics: Unit Testing

Answer:

If you want to unit test a private method, something may be wrong. Unit tests are (generally speaking) meant to test the interface of a class, meaning its public (and protected) methods. You can of course "hack" a solution to this (even if just by making the methods public), but you may also want to consider:

1. If the method you'd like to test is really worth testing, it may be worth to move it into its own class.
2. Add more tests to the public methods that call the private method, testing the private method's functionality. (As the commentators indicated, you should only do this if these private methods's functionality is really a part in with the public interface. If they actually perform functions that are hidden from the user (i.e. the unit test), this is probably bad).

Q6: What is a reasonable *Code Coverage* `%` for unit tests (and why)? ☆☆☆

Topics: Software Testing Unit Testing

Answer:

Code coverage is great, but **functionality coverage** is even better. I don't believe in covering every single line I write. But I do believe in writing 100% test coverage of all the functionality I want to provide (even for the extra cool features I came with myself and which were not discussed during the meetings).

I don't care if I would have code which is not covered in tests, but I would care if I would refactor my code and end up having a different behaviour. Therefore, 100% functionality coverage is my only target.

Q7: Is writing Unit Tests worth it for already exciting functionality? ☆☆☆

Topics: Software Testing Unit Testing

Answer:

It's absolutely worth it. Our app has complex cross-validation rules, and we recently had to make significant changes to the business rules. We ended up with conflicts that prevented the user from saving. I realized it would take forever to sort it out in the application (it takes several minutes just to get to the point where the problems were). I'd wanted to introduce automated unit tests and had the framework installed, but I hadn't done anything beyond a couple of dummy tests to make sure things were working. With the new business rules in hand, I started writing tests. *The tests quickly identified the conditions that caused the conflicts, and we were able to get the rules clarified.*

If you write tests that cover the functionality you're adding or modifying, you'll get an immediate benefit. If you wait for a re-write, you may never have automated tests.

You shouldn't spend a lot of time writing tests for existing things that already work. Most of the time, you don't have a specification for the existing code, so the main thing you're testing is your reverse-engineering ability. On the other hand, if you're going to modify something, you need to cover that functionality with tests so you'll

know you made the changes correctly. And of course, for new functionality, write tests that fail, then implement the missing functionality.

Q8: How can I unit test a *GUI*? ☆☆☆

Topics: Unit Testing

Answer:

The answer is to use MVC and move as much logic out of the GUI as possible. For the graphic, you should test the value that you supply to the code that generates the graphic.

I heard from a coworker a long time ago that when SGI was porting OpenGL to new hardware, they had a bunch of unit tests that would draw a set of primatives to the screen and then compute an MD5 sum of the frame buffer. This value could then be compared to known good hash values to quickly determine if the API is per pixel-accurate.

Q9: What's the difference between *Mock* an object or *Spy* on it? ☆☆☆

Topics: Unit Testing

Answer:

- **Mock object** replace mocked class entirely, returning recorded or default values. You can create a mock out of *thin air*. This is what is mostly used during unit testing.
- When **spying**, you take an existing object and *replace* only some methods. This is useful when you have a huge class and only want to *mock* certain methods (*partial mocking*). We *spy* real objects meaning that we can instruct which method to be *stubbed*. In a nutshell, you will *spy* **real object** and *stub* **some of the methods**.

When in doubt, use mocks.

Q10: *When* and *where* should I use *Mocking*? ☆☆☆

Topics: Unit Testing

Answer:

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**. Mock objects allow you to set up test scenarios without bringing to bear large, unwieldy resources such as databases. Mock objects are simulated objects that mimic the behaviour of real objects in controlled ways.

Typically you write a mock object if:

- The real object is too complex to incorporate it in a unit testing (For example a networking communication, you can have a mock object that simulate been the other peer)
- The result of your object is non-deterministic
- The real object is not yet available

For example:

Let's say 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. In the end, the mock object asserts, that all expected methods were called as expected.

Note:

A Unit Test should test a single code path 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.

Q11: Name some Unit Testing benefits for devs that you personally experienced ☆☆☆

Topics: Unit Testing

Answer:

1. Unit Tests allow you to make big changes to code quickly. You know it works now because you've run the tests, when you make the changes you need to make, you need to get the tests working again. This saves hours.
2. TDD helps you to realise when to stop coding. Your tests give you confidence that you've done enough for now and can stop tweaking and move on to the next thing.
3. The tests and the code work together to achieve better code. Your code could be bad / buggy. Your TEST could be bad / buggy. In TDD you are banking on the chances of **both** being bad / buggy being low. Often it's the test that needs fixing but that's still a good outcome.
4. TDD helps with coding constipation. When faced with a large and daunting piece of work ahead writing the tests will get you moving quickly.
5. Unit Tests help you really understand the design of the code you are working on. Instead of writing code to do something, you are starting by outlining all the conditions you are subjecting the code to and what outputs you'd expect from that.
6. Unit Tests give you instant visual feedback, we all like the feeling of all those green lights when we've done. It's very satisfying. It's also much easier to pick up where you left off after an interruption because you can see where you got to - that next red light that needs fixing.
7. Contrary to popular belief unit testing does not mean writing twice as much code or coding slower. It's faster and more robust than coding without tests once you've got the hang of it. Test code itself is usually relatively trivial and doesn't add a big overhead to what you're doing. This is one you'll only believe when you're doing it :)
8. I think it was Fowler who said: "Imperfect tests, run frequently, are much better than perfect tests that are never written at all". I interpret this as giving me permission to write tests where I think they'll be most useful even if the rest of my code coverage is woefully incomplete.
9. Good unit tests can help document and define what something is supposed to do
10. Unit tests help with code re-use. Migrate both your code **and** your tests to your new project. Tweak the code till the tests run again.

Q12: What's the difference between *Unit Tests* and *Integration Tests*? ☆☆☆

Topics: Unit Testing

Answer:

- A **Unit Test** is a test written for one single testable unit of code. The purpose of unit test is to check the behaviour of the tested function in a well-defined context. . A unit test should have no dependencies on code outside the unit tested. You decide what the unit is by looking for the smallest testable part. Where there are dependencies they should be replaced by false objects, mocks or stubs.
- An **Integration Test** is done to demonstrate that different pieces of the system work together. Integration tests can cover whole applications, and they require much more effort to put together. They usually require resources like database instances and hardware to be allocated for them. When false objects are replaced by real objects and the test execution thread crosses into other testable units, you have an integration test.

Q13: What is the fundamental value of *Unit Tests* vs *Integration Tests*? ☆☆☆

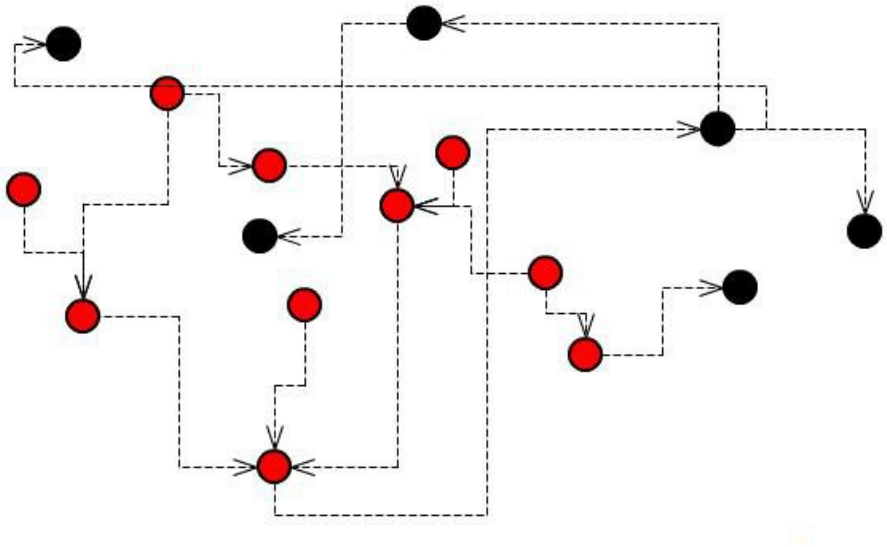
Topics: Unit Testing Software Testing

Answer:

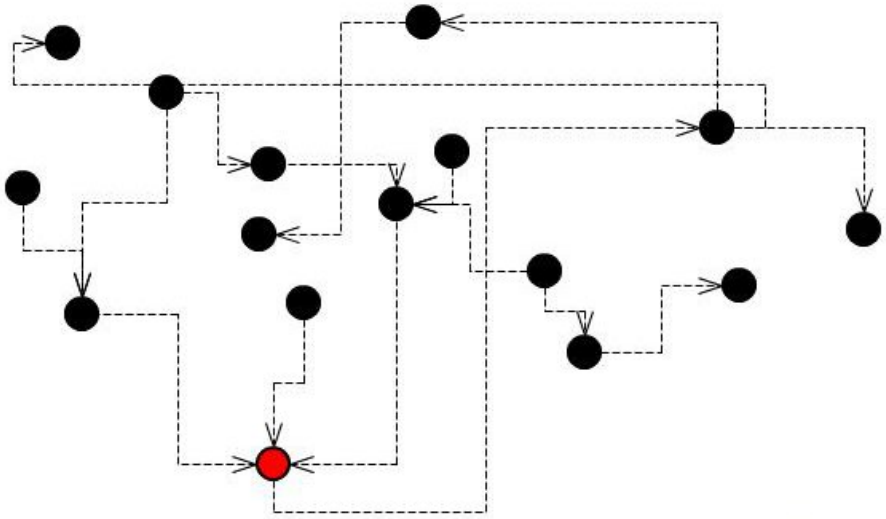
- **Integration tests** tell **what**'s not working. But they are of no use in **guessing where** the problem could be.
- **Unit tests** are the sole tests that tell you **where** exactly the bug is. To draw this information, they must run the method in a mocked environment, where all other dependencies are supposed to correctly work.

Consider:

A single bug will break several features, and several integration tests will fail.



On the other hand, the same bug will break just one unit test.



Q14: Should I Unit Test *private* methods or only *public* ones? ☆☆☆

Topics: Unit Testing

Answer:

In general, you don't want to break any encapsulation for the sake of testing (or as Mom used to say, "don't expose your privates!"). **Most of the time, you should be able to test a class by exercising its public methods**. If there is significant functionality that is hidden behind private or protected access, that might be a warning sign that there's another class in there struggling to get out.

You might not like testing private functionality for a couple of reasons:

1. Typically when you're tempted to test a class's private method, it's a design smell.
2. You can test them through the public interface (which is how you want to test them, because that's how the client will call/use them). You can get a false sense of security by seeing the green light on all the passing tests for your private methods. It is much better/safer to test edge cases on your private functions through your public interface.
3. You risk severe test duplication (tests that look/feel very similar) by testing private methods. This has major consequences when requirements change, as many more tests than necessary will break. It can also put you in a position where it is hard to refactor because of your test suite...which is the ultimate irony, because the test suite is there to help you safely redesign and refactor!

Q15: In a nutshell: what do I *lose* by adoptingTDD? What are the disadvantages of Test Driven Development? ☆☆☆

Topics: Unit Testing Software Testing

Answer:

- **Big-time investment + knowledge required of knowing how to write testable code.**. For the simple case, you lose about 20% of the actual implementation, but for complicated cases, you lose much more.
- Where you have a large number of tests, **changing the system might require re-writing** some or all of your **tests**, depending on which ones got invalidated by the changes. This could turn a relatively quick modification into a very time-consuming one.
- You might **start making design decisions based more on TDD** than on actually good design principles. As a result, you now have a much more complex system that is actually more prone to mistakes (and other devs and devs after you shall know how to support it).

- **Complexity of code (DI, IoC, MVC/MVP__) + Design Impacts**. When you start using mocks, after a while, you will want to start using Dependency Injection (DI) and an Inversion of Control (IoC) container. To do that you need to use interfaces for everything (which have a lot of pitfalls themselves). At the end of the day, **you have to write a lot more code**, than if you just do it the "plain old way". Instead of just a customer class, you also need to write an interface, a mock class, some IoC configuration and a few tests.
- **Prototyping** can be very **difficult** with TDD. **Continuous Tweaking** of tests in the **exploration** phase of the project. For data structures and black-box algorithms, unit tests would be perfect, but for algorithms that tend to be changed, tweaked or fine-tuned, this can cause a big-time investment that one might claim is not justified. Every time you change the design of a class, now you have to also change the test cases.

Q16: What is *Unit test*, *Integration Test*, *Smoke test*, *Regression Test* and what are the differences between them? ☆☆☆☆

Topics: Software Architecture Software Testing Unit Testing

Answer:

- **Unit test**: Specify and test one point of the contract of single method of a class. This should have a very narrow and well defined scope. Complex dependencies and interactions to the outside world are stubbed or mocked.

- **Integration test**: Test the correct inter-operation of multiple subsystems. There is whole spectrum there, from testing integration between two classes, to testing integration with the production environment.

- **Smoke test (aka Sanity check)**: A simple integration test where we just check that when the system under test is invoked it returns normally and does not blow up.

    - Smoke testing is both an analogy with electronics, where the first test occurs when powering up a circuit (if it smokes, it's bad!)...
    - ... and, apparently, with plumbing, where a system of pipes is literally filled by smoke and then checked visually. If anything smokes, the system is leaky.

- **Regression test**: A test that was written when a bug was fixed. It ensures that this specific bug will not occur again. The full name is "non-regression test". It can also be a test made prior to changing an application to make sure the application provides the same outcome.

To this, I will add:

- **Acceptance test**: Test that a feature or use case is correctly implemented. It is similar to an integration test, but with a focus on the use case to provide rather than on the components involved.

- **System test**: Tests a system as a black box. Dependencies on other systems are often mocked or stubbed during the test (otherwise it would be more of an integration test).

- **Pre-flight check**: Tests that are repeated in a production-like environment, to alleviate the 'builds on my machine' syndrome. Often this is realized by doing an acceptance or smoke test in a production like environment.

- **Canary test** is an automated, non-destructive test that is **run on a regular basis** in a **LIVE** environment, such that if it ever fails, something really bad has happened. Examples might be:

    - Has data that should only ever be available in DEV/TEST appeared in LIVE.
    - Has a background process failed to run
    - Can a user logon

Q17: What are best practices for Unit Testing methods that use *cache* heavily? ☆☆☆☆

Topics: Layering & Middleware Software Testing Unit Testing

Problem:

Consider

```
IList<TObject> AllFromCache() { ... }

TObject FetchById(guid id) { ... }

IList<TObject> FilterByPropertry(int property) { ... }
```

`Fetch.` and `Filter.` would call `AllFromCache` which would populate cache and return if it isn't there and just return from it if it is. What are best practices for Unit Testing against this type of structure?

Solution:

- First of all, move `AllFromCache()` into a repository class and call it `GetAll()` to comply with **Single Responsibility Principle**. That it retrieves from the cache is an implementation detail of the repository and shouldn't be known by the calling code.
- Second, wrap the class that gets the data from the database (or wherever) in a caching wrapper. AOP is a good technique for this. It's one of the few things that it's very good at.

```
public class ProductManager
{
  private IProductRepository ProductRepository { get; set; }

  public ProductManager
  {
      ProductRepository = productRepository;
  }

  Product FetchById(guid id) { ... }

  IList<Product> FilterByPropertry(int property) { ... }
}
```

public interface IProductRepository { IList GetAll(); }

public class SqlProductRepository : IProductRepository { public IList GetAll() { // DB Connection, fetch } }

public class CachedProductRepository : IProductRepository { private IProductRepository ProductRepository { get; set; }

```
public CachedProductRepository (IProductRepository productRepository)
{
    ProductRepository = productRepository;
}

public IList<Product> GetAll()
{
    // Check cache, if exists then return,
    // if not then call GetAll() on inner repository
}
```

}

* If you want true Unit Tests, then you have to *mock the cache*: write a mock object that implements the same interface as the cache, but instead of being a cache, it keeps track of the calls it receives, and always returns what the real cache should be returning according to the test case.

* Of course the cache itself also needs unit testing then, for which you have to mock anything it depends on, and so on.

Q18: Explain what is `Arrange-Act-Assert` pattern? ☆☆☆☆

Topics: Node.js Unit Testing

Answer:
Arrange-Act-Assert is a great way to structure test cases. It prescribes an order of operations:

1. **_Arrange_** inputs and targets. _Arrange_ steps should set up the test case. Does the test require any objects or special settings? Does it need to prep a database? Does it need to log into a web app? Handle all of these operations at the start of the test.
2. **_Act_** on the target behaviour. _Act_ steps should cover the main thing to be tested. This could be calling a function or method, calling a REST API, or interacting with a web page. Keep actions focused on the target behaviour.
3. **_Assert_** expected outcomes. _Act_ steps should elicit some sort of response. _Assert_ steps verify the goodness or badness of that response. Sometimes, assertions are as simple as checking numeric or string values. Other times, they may require checking multiple facets of a system. Assertions will ultimately determine if the test passes or fails.

Q19: Can _Unit Testing_ be successfully added into an existing production project? If so, how and is it worth it? ☆☆☆☆

Topics: Software Testing Unit Testing

Answer:
Adding testing (especially automated testing) makes it **_much_** easier to keep the project working in the future, and it makes it significantly less likely that you'll ship stupid problems to the user. In the long run, it will take more time and cost more money to fix bugs after the fact than write tests upfront. Period.

Although adding tests to existing code is valuable, it does come at a cost. It comes at the cost of _not_ building out new features. How these two things balance out depends entirely on the project, and there are a number of variables.

- How long will it take you to put all that code under test? Days? Weeks? Months? Years?
- Who are you writing this code for? Paying customers? An open-source project?
- What is your schedule like? Do you have hard deadlines you must meet? Do you have any deadlines at all?

The problem with retrofitting unit tests is you'll realise you didn't think of injecting a dependency here or using an interface there, and before long you'll be rewriting the entire component. If you have the time to do this, you'll build yourself a nice safety net, but you could have introduced subtle bugs along the way.

_If_ the code is split up reasonably well from the beginning, it's fairly easy to test it. The problem comes when you've got code that is messily all stitched together and where the only test points exposed are for full integration tests.

Q20: What's the best strategy for Unit-Testing _database-driven applications_? ☆☆☆☆

Topics: Unit Testing

Answer:
Follow these steps:

1. Keep the entire schema and scripts for creating it in source control so that anyone can create the current database schema after a checkout. It helps enormously with the quality of the DDL files.
2. In addition, keep sample data in data files that get loaded by part of the build process. The data is loaded from SQL, a template DB or a dump/backup. It also makes you think about which data to keep in your test DB and why.
3. Use a continuous integration server to build the database schema, load the sample data, and run tests. This is how you keep our test database in sync (rebuilding it at every test run).
4. After most tests, invoke `ROLLBACK` to keep data in the test database stable. ALWAYS keep the data in the test DB stable! If the data changes all the time, you can't test.

```
4. You can run tests against an in-memory DB (HSQLDB or Derby) if speed is an issue + you can run tests
locally
```

FullStack.Cafe - Kill Your Tech Interview

Q1: Can you provide and example of a common *Mocking* scenario?

☆☆☆

Topics: Unit Testing

Answer:

Mocks are basically objects that look like the object you are trying to call a method on, in the sense that they have the same properties, methods, etc. available to the caller. But instead of performing whatever action they are programmed to do when a particular method is called, it skips that altogether and just returns a result. That result is typically defined by you ahead of time.

Let look at the mocking out your calls to the database. In order to set up your objects for mocking, you probably need to use some sort of *inversion of control/ dependency injection pattern*, as in the following pseudo-code:

```
class Bar
{
    private FooDataProvider _dataProvider;

    public instantiate(FooDataProvider dataProvider) {
        _dataProvider = dataProvider;
    }

    public getAllFoos() {
        // instead of calling Foo.GetAll() here, we are introducing an extra layer of abstraction
        return _dataProvider.GetAllFoos();
    }
}

class FooDataProvider
{
    public Foo[] GetAllFoos() {
        return Foo.GetAll();
    }
}
```

Now in your unit test, you create a mock of `FooDataProvider`, which allows you to call the method `GetAllFoos` *without having to actually hit the database*.

```
class BarTests
{
    public TestGetAllFoos() {
        // here we set up our mock FooDataProvider
        mockRepository = MockingFramework.new()
        mockFooDataProvider = mockRepository.CreateMockOfType(FooDataProvider);

        // create a new array of Foo objects
        testFooArray = new Foo[] {Foo.new(), Foo.new(), Foo.new()}

        // the next statement will cause testFooArray to be returned every time we call
FooDAtaProvider.GetAllFoos,
        // instead of calling to the database and returning whatever is in there
        // ExpectCallTo and Returns are methods provided by our imaginary mocking framework
        ExpectCallTo(mockFooDataProvider.GetAllFoos).Returns(testFooArray)

        // now begins our actual unit test
        testBar = new Bar(mockFooDataProvider)
        baz = testBar.GetAllFoos()
```

```
        // baz should now equal the testFooArray object we created earlier
        Assert.AreEqual(3, baz.length)
    }
}
```

Q2: What is the best way to unit test a method that doesn't return anything (`void`)? ☆☆☆☆

Topics: Unit Testing

Answer:

Test its side effects. This includes:

- Does it **throw any exceptions**? (If it should check that it does. If it shouldn't, try some corner cases which might if you're not careful - null arguments being the most obvious thing.)
- Does it play nicely with its **parameters**? (If they're mutable, does it mutate them when it shouldn't and vice versa?)
  - Should it throw an exception when called with the wrong parameters?
  - Should it throw no exception when called with the right parameters?
- Does it have the right **effect on the state** of the object/type you're calling it on?
- You can also consider to **refactor your code**

Instead of a method like:

```
public void SendEmailToCustomer()
```

Make a method that follows Microsoft's `int.TryParse()` paradigm:

```
public bool TrySendEmailToCustomer()
```

Q3: How do I *test* a `private` function or a class that has `private` methods, fields or inner classes? ☆☆☆☆☆

Topics: Software Architecture Unit Testing

Answer:

The best way to test a private method is via another public method. If this cannot be done, then one of the following conditions is true:

1. The private method is dead code
2. There is a design smell near the class that you are testing
3. The method that you are trying to test should not be private

Also, by testing private methods you are testing the implementation. This defeats the purpose of unit testing, which is to test the inputs/outputs of a class' contract. A test should only know enough about the implementation to mock the methods it calls on its dependencies. Nothing more. If you can not change your implementation without having to change a test - chances are that your test strategy is poor.

Q4: Is Unit Testing *worth* the effort? ☆☆☆☆☆

Topics: Software Testing Unit Testing

Answer:

- Unit Tests allows you to make big changes to code quickly. You know it works now because you've run the tests, when you make the changes you need to make, you need to get the tests working again. This saves hours.
- TDD helps you to realise when to stop coding. Your tests give you confidence that you've done enough for now and can stop tweaking and move on to the next thing.
- The tests and the code work together to achieve better code. Your code could be bad / buggy. Your TEST could be bad / buggy. In TDD you are banking on the chances of both being bad / buggy being low. Often it's the test that needs fixing but that's still a good outcome.
- TDD helps with coding constipation. When faced with a large and daunting piece of work ahead writing the tests will get you moving quickly.
- Unit Tests help you really understand the design of the code you are working on. Instead of writing code to do something, you are starting by outlining all the conditions you are subjecting the code to and what outputs you'd expect from that.
- Unit Tests give you instant visual feedback, we all like the feeling of all those green lights when we've done. It's very satisfying. It's also much easier to pick up where you left off after an interruption because you can see where you got to - that next red light that needs fixing.
- Contrary to popular belief unit testing does not mean writing twice as much code, or coding slower. It's faster and more robust than coding without tests once you've got the hang of it. Test code itself is usually relatively trivial and doesn't add a big overhead to what you're doing. This is one you'll only believe when you're doing it :)
- I think it was Fowler who said: "Imperfect tests, run frequently, are much better than perfect tests that are never written at all". I interpret this as giving me permission to write tests where I think they'll be most useful even if the rest of my code coverage is woefully incomplete.
- Good unit tests can help document and define what something is supposed to do
- Unit tests help with code re-use. Migrate both your code and your tests to your new project. Tweak the code till the tests run again.