**What is unit test? (A test that purely tests a single unit of functionality)**

**What is component test?**

**What is integration test? (Examine several parts of a system to make sure that when integrated, these parts behave as expected)**

**What is user acceptance test? BDD?**

*Depending on where you look, you'll get slightly different answers. I've read about the subject a lot, and here's my distillation; again, these are slightly wooly and others may disagree.*

*Unit Tests*

*Tests the smallest unit of functionality, typically a method/function (e.g. given a class with a particular state, calling x method on the class should cause y to happen). Unit tests should be focussed on one particular feature (e.g., calling the pop method when the stack is empty should throw an InvalidOperationException). Everything it touches should be done in memory; this means that the test code and the code under test shouldn't:*

*Call out into (non-trivial) collaborators*

*Access the network*

*Hit a database*

*Use the file system*

*Spin up a thread*

*etc.*

*Any kind of dependency that is slow / hard to understand / initialise / manipulate should be stubbed/mocked/whatevered using the appropriate techniques so you can focus on what the unit of code is doing, not what its dependencies do.*

*In short, unit tests are as simple as possible, easy to debug, reliable (due to reduced external factors), fast to execute and help to prove that the smallest building blocks of your program function as intended before they're put together. The caveat is that, although you can prove they work perfectly in isolation, the units of code may blow up when combined which brings us to ...*

*Integration Tests*

*Integration tests build on unit tests by combining the units of code and testing that the resulting combination functions correctly. This can be either the innards of one system, or combining multiple systems together to do something useful. Also, another thing that differentiates integration tests from unit tests is the environment. Integration tests can and will use threads, access the database or do whatever is required to ensure that all of the code and the different environment changes will work correctly.*

*If you've built some serialization code and unit tested its innards without touching the disk, how do you know that it'll work when you are loading and saving to disk? Maybe you forgot to flush and dispose filestreams. Maybe your file permissions are incorrect and you've tested the innards using in memory streams. The only way to find out for sure is to test it 'for real' using an environment that is closest to production.*

*The main advantage is that they will find bugs that unit tests can't such as wiring bugs (e.g. an instance of class A unexpectedly receives a null instance of B) and environment bugs (it runs fine on my single-CPU machine, but my colleague's 4 core machine can't pass the tests). The main disadvantage is that integration tests touch more code, are less reliable, failures are harder to diagnose and the tests are harder to maintain.*

*Also, integration tests don't necessarily prove that a complete feature works. The user may not care about the internal details of my programs, but I do!*

*Functional Tests*

*Functional tests check a particular feature for correctness by comparing the results for a given input against the specification. Functional tests don't concern themselves with intermediate results or side-effects, just the result (they don't care that after doing x, object y has state z). They are written to test part of the specification such as, "calling function Square(x) with the argument of 2 returns 4".*

*Acceptance Tests*

*Acceptance testing seems to be split into two types:*

*Standard acceptance testing involves performing tests on the full system (e.g. using your web page via a web browser) to see whether the application's functionality satisfies the specification. E.g. "clicking a zoom icon should enlarge the document view by 25%." There is no real continuum of results, just a pass or fail outcome.*

*The advantage is that the tests are described in plain English and ensures the software, as a whole, is feature complete. The disadvantage is that you've moved another level up the testing pyramid. Acceptance tests touch mountains of code, so tracking down a failure can be tricky.*

*Also, in agile software development, user acceptance testing involves creating tests to mirror the user stories created by/for the software's customer during development. If the tests pass, it means the software should meet the customer's requirements and the stories can be considered complete. An acceptance test suite is basically an executable specification written in a domain specific language that describes the tests in the language used by the users of the system.*

*Conclusion*

*They're all complementary. Sometimes it's advantageous to focus on one type or to eschew them entirely. The main difference for me is that some of the tests look at things from a programmer's perspective, whereas others use a customer/end user focus.*

***Why Unite Test***

*Every day in our office there is an exchange which goes something like this:*

*"Man, I just love unit tests, I've just been able to make a bunch of changes to the way something works, and then was able to confirm I hadn't broken anything by running the test over it again..."*

*The details change daily, but the sentiment doesn't. Unit tests and test-driven development (TDD) have so many hidden and personal benefits as well as the obvious ones that you just can't really explain to somebody until they're doing it themselves.*

*But, ignoring that, here's my attempt!*

*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.*

*A lot of work I'm involved with doesn't Unit Test well (web application user interactions etc.), but even so we're all test infected in this shop, and happiest when we've got our tests tied down. I can't recommend the approach highly enough.*

***Types of tests: acceptance testing, functional testing, smoke testing, regression testing, unit testing, integration testing, stress testing, (Load, Performance, Sanity, Stability, Security, Feature, Progression, Installation, Business).***

***How to test multithreading code?***

*Yes you can do multi-threading in PHP with pthreads*

*From the PHP documentation:*

*pthreads is an object-orientated API that provides all of the tools needed for multi-threading in PHP. PHP applications can create, read, write, execute and synchronize with Threads, Workers and Threaded objects.*

*Warning: The pthreads extension cannot be used in a web server environment. Threading in PHP should therefore remain to CLI-based applications only.*

*Simple Test*

*#!/usr/bin/php*

*<?php*

*class AsyncOperation extends Thread {*

*public function \_\_construct($arg) {*

*$this->arg = $arg;*

*}*

*public function run() {*

*if ($this->arg) {*

*$sleep = mt\_rand(1, 10);*

*printf('%s: %s -start -sleeps %d' . "\n", date("g:i:sa"), $this->arg, $sleep);*

*sleep($sleep);*

*printf('%s: %s -finish' . "\n", date("g:i:sa"), $this->arg);*

*}*

*}*

*}*

*// Create a array*

*$stack = array();*

*//Initiate Multiple Thread*

*foreach ( range("A", "D") as $i ) {*

*$stack[] = new AsyncOperation($i);*

*}*

*// Start The Threads*

*foreach ( $stack as $t ) {*

*$t->start();*

*}*

[*https://stackoverflow.com/questions/70855/how-can-one-use-multi-threading-in-php-applications*](https://stackoverflow.com/questions/70855/how-can-one-use-multi-threading-in-php-applications)