Homework

1- **Regression Testing** is defined as a type of software testing to confirm that a recent program or code change has not adversely affected existing features. Regression Testing is a full or partial selection of already executed test cases which are re-executed to ensure existing functionalities work fine.

2- **A/B testing** (also known as split testing or bucket testing) is a method of comparing two versions of a webpage or app against each other to determine which one performs better. A/B testing is essentially an experiment where two or more variants of a page are shown to users at random, and statistical analysis is used to determine which variation performs better for a given conversion goal.

3- In **Black-box testing**, a tester doesn't have any information about the internal working of the software system. Black box testing is a high level of testing that focuses on the behavior of the software. It involves testing from an external or end-user perspective. Black box testing can be applied to virtually every level of software testing: unit, integration, system, and acceptance.

White-box testing is a testing technique which checks the internal functioning of the system. In this method, testing is based on coverage of code statements, branches, paths or conditions. White-Box testing is considered as low-level testing. It is also called glass box, transparent box, clear box or code base testing. The white-box Testing method assumes that the path of the logic in a unit or program is known.

4- **Mutation Testing** is a type of Software Testing that is performed to design new software tests and also evaluate the quality of already existing software tests. Mutation testing is related to modification a program in small ways. It focuses to help the tester develop effective tests or locate weaknesses in the test data used for the program.

5- **Behavior-driven development (BDD)** is an Agile software development methodology in which an application is documented and designed around the behavior a user expects to experience when interacting with it. By encouraging developers to focus only on the requested behaviors of an app or program, BDD helps to avoid bloat, excessive code, unnecessary features or lack of focus. This methodology combines, augments and refines the practices used in test-driven development (TDD) and acceptance testing.

6- **Agile Testing Quadrants** are a visual tool for understanding different QA tests. They differentiate between business- and technology-facing tests, and those that support programming or 'critique' the product. Testing types are sorted into these four categories on a grid.

   - **Q1 :** Unit and component tests are performed throughout your application's development. They provide feedback to your developers on the quality of their code on an ongoing basis, usually through repeated, automated processes.
   - **Q2 :** With a combination of manual and automated tests, these 'business-facing' tests are more customer-focused, but they support your application build as well. These include functional tests, which ensure the product does what it is meant to do.
   - **Q3 :** User Acceptance Tests (UAT), usability and exploratory testing all belong in this quadrant. These involve manual testing by experienced QA engineers and end-user testing. Your aim here is to gain feedback and improve the quality of the product, ensuring it is fit for the designed purpose.
   - **Q4 :** These are technology-facing performance tests, like load testing and checking the data security of your software application. There are many tools available to automate this type of testing, such as Selenium used alongside JMeter.

7- Abstract of(https://martinfowler.com/articles/practical-test-pyramid.html)

Practical Test Pyramid

The "Test Pyramid" is a metaphor that tells us to group software tests into buckets of different granularity. It also gives an idea of how many tests we should have in each of these groups.
Having an effective software testing approach allows teams to move fast and with confidence.

The Importance of Test Automation

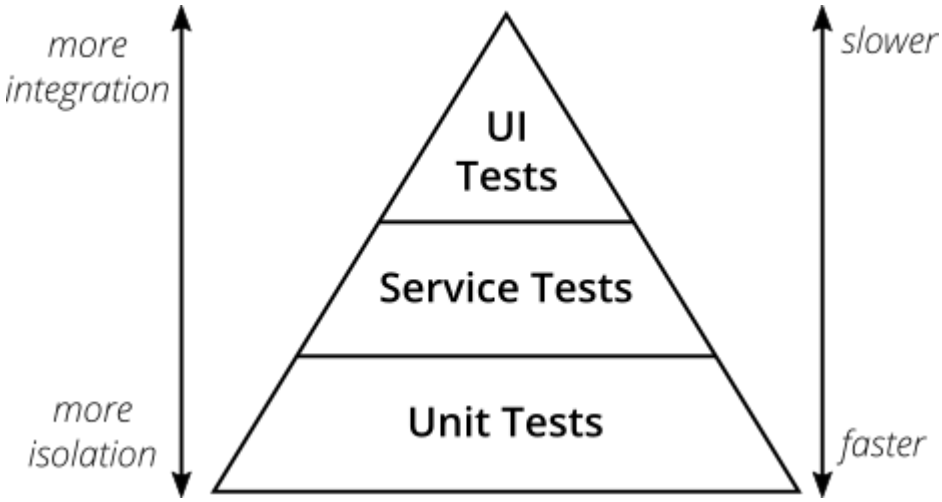
With continuous delivery you use a build pipeline to automatically test your software and deploy it to your testing and production environments. Automating everything is your only way forward.

It's obvious that testing all changes manually is time-consuming, repetitive and tedious. Repetitive is boring, boring leads to mistakes and makes you look for a different job by the end of the week.

Automating your repetitive tests can be a big game changer in your life as a software developer.

The Test Pyramid

Mike Cohn came up with this concept in his book Succeeding with Agile. It's a great visual metaphor telling you to think about different layers of testing. It also tells you how much testing to do on each layer.



From a modern point of view the test pyramid seems overly simplistic and can therefore be misleading.

Still, due to its simplicity the essence of the test pyramid serves as a good rule of thumb when it comes to establishing your own test suite. Your best bet is to remember two things from Cohn's original test pyramid:

- Write tests with different granularity
- The more high-level you get the fewer tests you should have

Stick to the pyramid shape to come up with a healthy, fast and maintainable test suite: Write lots of small and fast unit tests. Write some more coarse-grained tests and very few high-level tests that test your application from end to end.

Given the shortcomings of the original names it's totally okay to come up with other names for your test layers, as long as you keep it consistent within your codebase and your team's discussions.

Tools and Libraries

JUnit: our test runner
Mockito: for mocking dependencies
Wiremock: for stubbing out external services
Pact: for writing CDC tests
Selenium: for writing UI-driven end-to-end tests
REST-assured: for writing REST API-driven end-to-end tests

Unit Tests

The foundation of your test suite will be made up of unit tests. Your unit tests make sure that a certain unit (your subject under test) of your codebase works as

intended. Unit tests have the narrowest scope of all the tests in your test suite. The number of unit tests in your test suite will largely outnumber any other type of test.

What's a Unit ? : If you're working in a functional language a unit will most likely be a single function. Your unit tests will call a function with different parameters and ensure that it returns the expected values. In an object-oriented language a unit can range from a single method to an entire class.

Sociable and solitary : At the end of the day it's not important to decide if you go for solitary or sociable unit tests. Writing automated tests is what's important. Personally, I find myself using both approaches all the time. If it becomes awkward to use real collaborators I will use mocks and stubs generously. If I feel like involving the real collaborator gives me more confidence in a test I'll only stub the outermost parts of my service.

Mocking and stubbing : In plain words it means that you replace a real thing (e.g. a class, module or function) with a fake version of that thing. The fake version looks and acts like the real thing (answers to the same method calls) but answers with canned responses that you define yourself at the beginning of your unit test.
Using test doubles is not specific to unit testing. More elaborate test doubles can be used to simulate entire parts of your system in a controlled way. However, in unit testing you're most likely to encounter a lot of mocks and stubs simply because lots of modern languages and libraries make it easy and comfortable to set up mocks and stubs.

Your unit tests will run very fast. On a decent machine you can expect to run thousands of unit tests within a few minutes. Test small pieces of your codebase in isolation and avoid hitting databases, the filesystem or firing HTTP queries (by using mocks and stubs for these parts) to keep your tests fast.

Once you got a hang of writing unit tests you will become more and more fluent in writing them. Stub out external collaborators, set up some input data, call your subject under test and check that the returned value is what you expected. Look into Test-Driven Development and let your unit tests guide your development; if applied correctly it can help you get into a great flow and come up with a good and maintainable design while automatically producing a comprehensive and fully automated test suite. Still, it's no silver bullet. Go ahead, give it a real chance and see if it feels right for you.

What to Test ? : A unit test class should at least test the public interface of the class. Private methods can't be tested anyways since you simply can't call them from a different test class. Protected or package-private are accessible from a test class (given the package structure of your test class is the same as with the production class) but testing these methods could already go too far.

There's a fine line when it comes to writing unit tests: They should ensure that all your non-trivial code paths are tested (including happy path and edge cases). At the same time they shouldn't be tied to your implementation too closely.

Test Structure :

A good structure for all your tests is this one:

1) Setup the test data
2) Call your method under test
3) Assert that the expected results are returned

There's a nice mnemonic to remember this structure: "Arrange, Act, Assert". Another one that you can use takes inspiration from BDD . It's the "given", "when", "then" triad, where given reflects the setup, when the method call and then the assertion part.

This pattern can be applied to other, more high-level tests as well. In every case they ensure that your tests remain easy and consistent to read. On top of that tests written with this structure in mind tend to be shorter and more expressive.

Implementing a Unit Test :

We're writing the unit tests using JUnit, the de-facto standard testing framework for Java. We use Mockito to replace the real Repository class with a stub for our test. This stub allows us to define canned responses the stubbed method should return in this test. Stubbing makes our test more simple, predictable and allows us to easily setup test data.

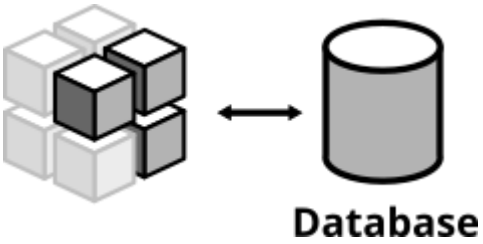
Following the arrange, act, assert structure, we write two unit tests - a positive case and a case where the searched person cannot be found. The first, positive test case creates a new object and tells the mocked repository to return this object when it's called as the value for the parameter. The test then goes on to call the method that should be tested. Finally it asserts that the response is equal to the expected response.

Integration Tests

Integration Tests are there to help testing the integration of your application with all the parts that live outside of your application.

For your automated tests this means you don't just need to run your own application but also the component you're integrating with. If you're testing the integration with a database you need to run a database when running your tests. For testing that you can read files from a disk you need to save a file to your disk and load it in your integration test.

Narrow integration tests live at the boundary of your service. Conceptually they're always about triggering an action that leads to integrating with the outside part (filesystem, database, separate service). A database integration test would look like this:



1- Start a database
2- Connect your app to the database
3- Trigger a function within your code that writes data to the database
4- Check that the expected data has been written to the database by reading the data from the database

Write integration tests for all pieces of code where you either serialize or deserialize data.

When writing narrow integration tests you should aim to run your external dependencies locally: spin up a local MySQL database, test against a local ext4 filesystem. If you're integrating with a separate service either run an instance of that service locally or build and run a fake version that mimics the behaviour of the real service.

Integrating with a service over the network is a typical characteristic of a broad integration test and makes your tests slower and usually harder to write.

With regards to the test pyramid, integration tests are on a higher level than your unit tests. Integrating slow parts like filesystems and databases tends to be much slower than running unit tests with these parts stubbed out. They can also be harder to write than small and isolated unit tests, after all you have to take care of spinning up an external part as part of your tests. Still, they have the advantage of giving you the confidence that your application can correctly work with all the external parts it needs to talk to. Unit tests can't help you with that.

Contract Tests

Splitting your system into many small services often means that these services need to communicate with each other via certain (hopefully well-defined, sometimes accidentally grown) interfaces.

UI Tests

Most applications have some sort of user interface. Typically we're talking about a web interface in the context of web applications. People often forget that a REST API or a command line interface is as much of a user interface as a fancy web user interface.

UI tests test that the user interface of your application works correctly. User input should trigger the right actions, data should be presented to the user, the UI state should change as expected.

End-to-end Tests

Testing your deployed application via its user interface is the most end-to-end way you could test your application. The previously described, webdriver driven UI tests are a good example of end-to-end tests.

Acceptance Tests

Acceptance tests can come in different levels of granularity. Most of the time they will be rather high-level and test your service through the user interface. However, it's good to understand that there's technically no need to write acceptance tests at the highest level of your test pyramid. If your application design and your scenario at hand permits that you write an acceptance test at a lower level, go for it. Having a low-level test is better than having a high-level test. The concept of acceptance tests - proving that your features work correctly for the user - is completely orthogonal to your test pyramid.

Exploratory Tests

It is a manual testing approach that emphasises the tester's freedom and creativity to spot quality issues in a running system. Simply take some time on a regular schedule, roll up your sleeves and try to break your application. Use a destructive mindset and come up with ways to provoke issues and errors in your application. Document everything you find for later. Watch out for bugs, design issues, slow response times, missing or misleading error messages and everything else that would annoy you as a user of your software.

The Confusion About Testing Terminology

Don't get too hung up on sticking to ambiguous terms. It doesn't matter if you call it end-to-end or broad stack test or functional test. It doesn't matter if your integration tests mean something different to you than to the folks at another company. Yes, it would be really nice if our profession could settle on some well-defined terms and all stick to it. Unfortunately this hasn't happened yet. And since there are many nuances when it comes to writing tests it's really more of a spectrum than a bunch of discrete buckets anyways, which makes consistent naming even harder.

Avoid Test Duplication

As with production code you should strive for simplicity and avoid duplication. In the context of implementing your test pyramid you should keep two rules of thumb in mind:

- If a higher-level test spots an error and there's no lower-level test failing, you need to write a lower-level test
- Push your tests as far down the test pyramid as you can

Writing Clean Test Code

As with writing code in general, coming up with good and clean test code takes great care. Here are some more hints for coming up with maintainable test code before you go ahead and hack away on your automated test suite:

- Test code is as important as production code. Give it the same level of care and attention. "this is only test code" is not a valid excuse to justify sloppy code
- Test one condition per test. This helps you to keep your tests short and easy to reason about
- "arrange, act, assert" or "given, when, then" are good mnemonics to keep your tests well-structured
- Readability matters. Don't try to be overly DRY . Duplication is okay, if it improves readability. Try to find a balance between DRY and DAMP code
- When in doubt use the Rule of Three to decide when to refactor. Use before reuse