# SOEN390 - Testing Plan

This testing plan suggests what should be done during the development of the system. It is based on the recent book by Mauricio Aniche - Effective Software Testing (2021).

You’re required to write test in **all levels**:

* Unit tests
* Integration tests
* System tests

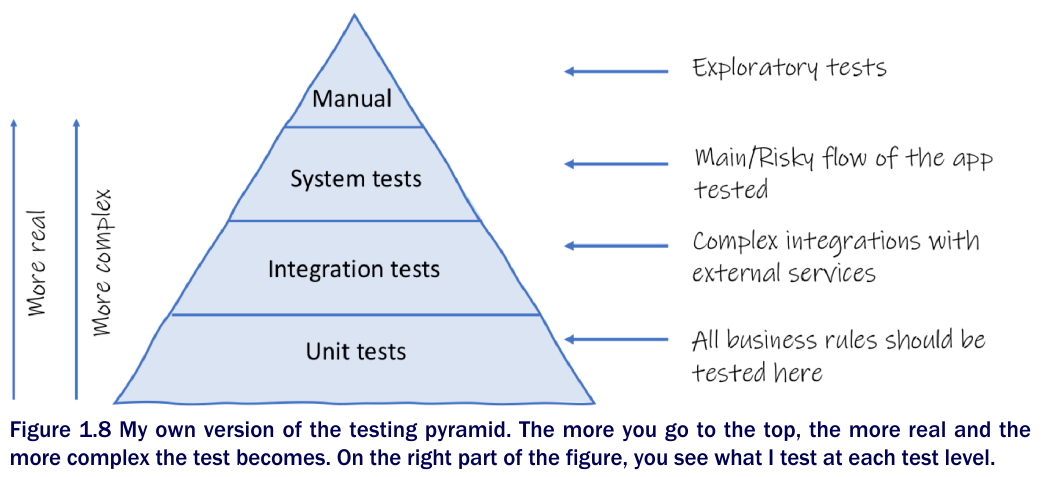
Also, create any form of **automation** for executing these tests. For example, using Continuous Integration (CI) pipeline that triggers every new commit or merge.

As for manual testing, that also could be (partially) automated using UI tools and scripts.

Finally, the toolset is up to the team to choose.

## Overview

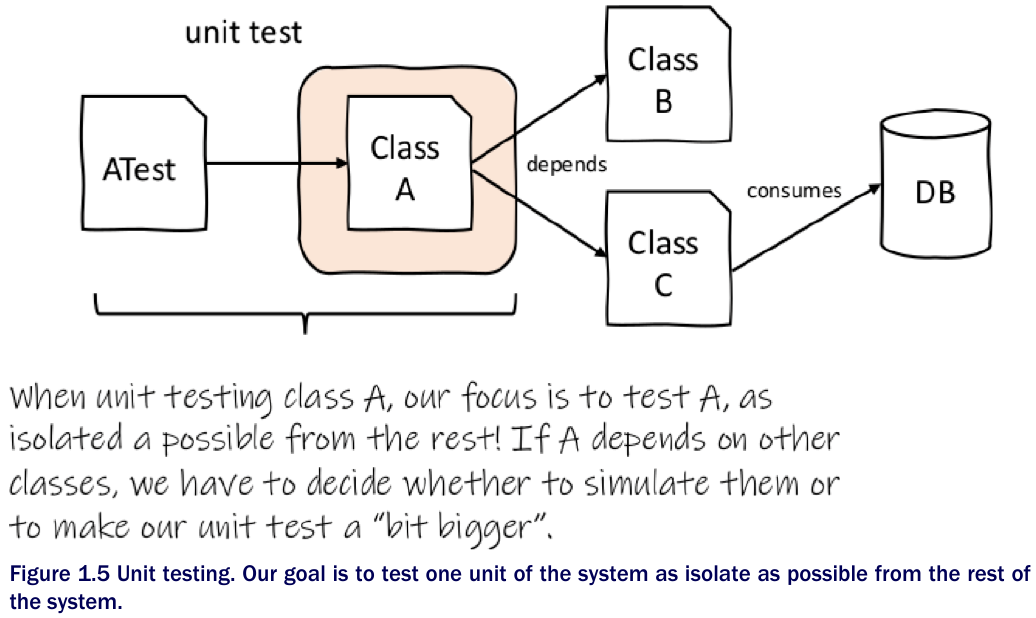
* The size of the slice in the pyramid basically represents the number of tests one should carry out at each test level.
* Unit testing is at the bottom of the pyramid and has the largest area of them all. This means that such developers favor unit testing (i.e., write more unit tests). The reasons are: they are fast, require less effort to be written, and give developers more control.
* As we climb up the levels on the diagram, we see that the next level is integration testing. The area is a bit smaller, indicating that, in practice, these developers write integration tests less than unit tests. Given the extra effort that integration tests require, testers make sure to write tests only for the integrations they really need.
* The diagram continues, showing that testers then favor system tests less than integration tests and have even fewer manual tests.



## 

## Unit Testing

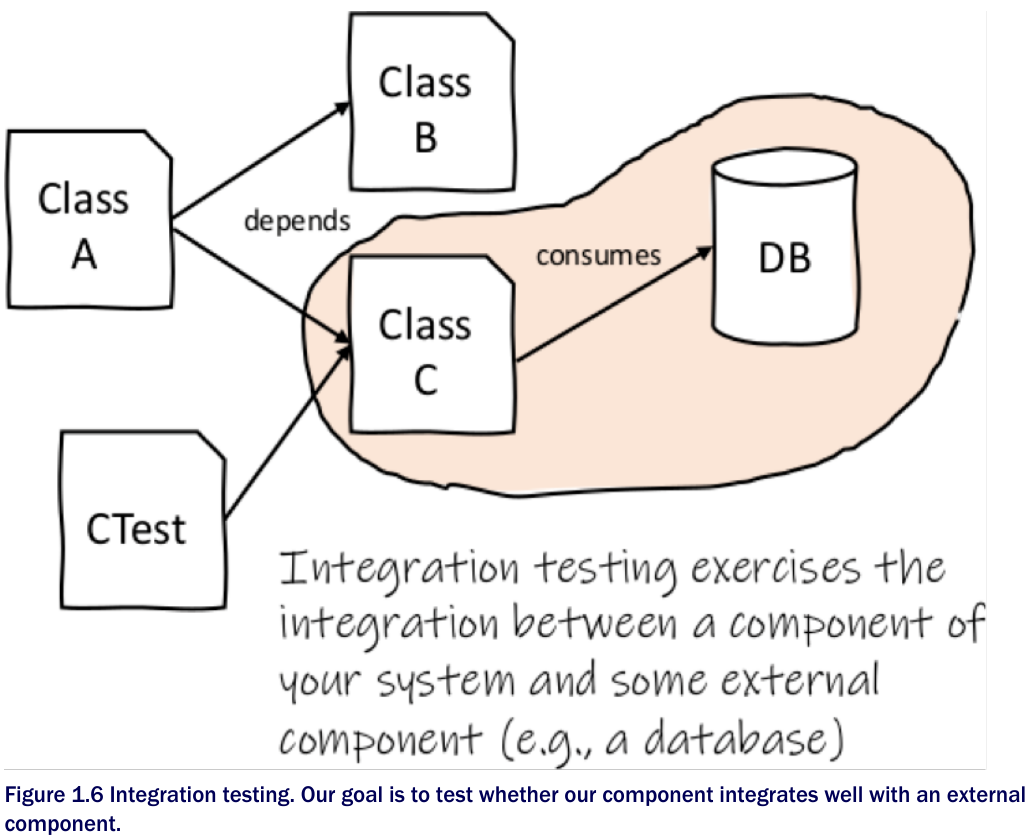
* In some situations, the goal of the tester is to test a single feature of the software, purposefully ignoring the other units of the systems.
* The goal is to test the *doSomething()* method and nothing else.
* Of course, we cared about how this method would interact with the rest of the system, but we did not really test it together with the other pieces of the system.
* What if a class I want to test depends on another class that talks to, e.g., a database? if my focus is to test some class, and this class depends on another class that depends on a database, and I know that handling databases in my automated tests is just much harder than not having to handle them, I would simulate this database class. In other words, I would create a stub that acts like the original class, but it is much simpler and much easier to use during testing.



## 

## Integration Testing

* The goal of integration testing is to test multiple components of a system together, focusing on the interactions between them instead of testing the system as a whole. Are they communicating correctly? What happens if component A sends message X to component B? Do they still present correct behaviour?
* Example: Software systems commonly rely on database systems. To communicate with the database, developers often create a class whose only responsibility is to interact with this external component (think of Data Access Objects - DAO - classes). These DAOs might contain complicated SQL code. Thus, a tester feels the need to test these SQL queries. Note that the tester does not want to test the entire system, only the integration between the DAO class and the database. The tester also does not want to test the DAO class in complete isolation; after all, the best way to know whether a SQL query works is to actually submit it to the database and see what the database returns back.
* Integration testing becomes handy whenever you want to ensure that the integration between a component of your system and an external component works correctly. Note that, because we focus on these two parts (our component and the external component), writing such a test is less complicated than writing a test that would go through the entire system and components we do not care about.
* When compared to unit testing, integration tests are more difficult to write. In the example, setting up a database for the test requires effort. Tests that involve databases usually need to make use of an isolated instance of the database just for testing purposes (as you probably do not want your tests to mess with production data), update the database schema (in fast companies, database schemas are changing all the time, and the test database needs to keep up), put the database into a state expected by the test by adding or removing rows, and clean everything afterwards (so that the next tests do not fail because of the data that was left behind by the previous test). The same effort happens to any other type of integration test you can imagine, e.g., web services, file reads and writes, etc.



## System Testing

* To get a more realistic view of the software, and thus perform more realistic tests, we should run the entire software system, with all its databases, front-end apps, and any other components it has.
* When we test the system in its entirety, we are doing what is called system testing. In practice, instead of testing small parts of the system in isolation, system tests exercise the system as a whole. We do not really care how the system works from the inside; we do not care if it was developed in Java or in Ruby, if it uses a relational database or not. We only care that, given this input, the system will provide that output.

