Testing considerations

Introduction

The purpose of testing is to assure that the code, that we deliver, will do what we and the user expect. There are different ways how you can achieve this goal. They vary, as many other products, by cost and the result that you can expect.

You can consider certain characteristics of what you can expect from testing. Think about it as a service that your order (just for this moment, otherwise testability and testing needs to be part of the design).

Reliability: How much you can rely that running one or set of tests will ensure that the product will functioning as you expected.

Speed of execution: How long does it take to execute all tests.

Diagnostics: If a test fails, how detail information you will get.

Cost: How much time it takes to implements these tests.

Maintenance cost: if you change internal implementation, how much time you need to spend with updating tests.

You cannot get all of the characteristics with the best expected value possible, usually for the highest cost. Best diagnostics requires writing more tests that validate components or method behavior. More tests will run longer. If they test product internals, they will need to be updated if the product implementation changes. If you want to get high reliability, you need to run the product as close possible to its production environment. This may require to setup external system like databases as part of the test, which will add to the execution time. You need to choose case by for specific situation.

In terms of priority, you could consider this as guideline:

Rank tests for scenarios by priority:

- How often are they used
- How critical is the failure

Have at least one high reliable test (end to end test) for each scenario. If output variations encapsulate one method, you can write unit tests for the method to validate the variations.

Test automation:

When it makes sense to automate tests.

Some people are in favor to try to automate as much as possible for various reasons: they may hate manual work and it is better for them to spend 1 day to write test automation for something what can be tested in 10 minutes of manual work and will not likely be needed to be tested again, or very rarely.

As the previous sentence suggests, if your primary business reason is to ensure that a feature of your product works properly after first release and after future product updates, you can apply basic math and calculate.

Automation cost < manual test cost * number of execution during estimated product lifecycle

The equation seems simple, but the last part is tricky. How do you know the "number of execution during estimated product lifecycle". You may try to estimate how often you release (monthly, quarterly, yearly) and how long this feature could be part of your product.

Another way to think about it: As in life, there are situation where you want to pay insurance and where the insurance does not make much sense. If you live uphill, flooding insurance is less important than for somebody living at river bank. Based on where you live, you need appropriate insurance – test automation.

How to decide what to test.

You can often hear that you need to test everything. In reality, situation may be quite opposite, because what matter most, and reasonably, is the product. So many project schedules are tight, last few weeks before release people concentrate of product delivery as it becomes more important than test automation. I am saying specifically automation, not testing.

Code coverage:

Simple rule of testing states to run all the code before you ship it. Test coverage helps you find out which part of the code was exercised by test and which not.

In old days, people could put break point at each line of the code and run the code and each time when the code stopped in the debugger, they would remove the break point. The breakpoints left after code execution would showed you, which code has not been exercised.

Today, tools like Visual Studio help us to figure out which part of the code was exercised and which was not.

Code coverage is not test coverage:

Exercising every line of the code does not quarantine that the product will work as we expect in all situations, because the code may produce different results base on different input parameters. In this case, you need different test cases that will run the same scenario with different input parameters.

Mock testing

Mock testing helps with the "Speed of execution" characteristics and sometimes with product code implementation itself, because it allows you to run code even before external systems are ready for use. On the other hand you do not test complete system, so you need to plan for integration testing as well.

Frameworks:

You can use different framework like Microsoft Fakes, NMock, Moq and others.

Unit test vs. end-to-end test characteristics

The table below shows different aspects of unit versus end-to-end testing.

1 Good
2 Neutral
3 bad

	Unit testing	**End to end testing**
Reliability	2 if you have a lot of tests 3 the same amount	1
Speed of execution	1	3 less tests running slowly
Diagnostics	1	3
Cost	3	1*
Maintenance cost	3	1

* assuming external dependencies framework is ready