SUBJECT7
QUALITY IS A CULTURE

Test Approaches and Types Distilled

By Matthew Heusser Matt@xndev.com for Subject7

A few years ago, a friend emailed me a survey about software testing. The survey asked us to specify our roles, along with which kind of testing we thought was most important. As I recall, the question was, "If you could only do one type of testing, which would you pick?" Surprisingly, the point of the survey was to see if there was a correlation between role and preference -- and there was. Patrick Bailey, a professor at Calvin College, found that developers overwhelmingly valued unit testing, analysts valued system testing, project managers and other customer-focused roles valued user acceptance testing.

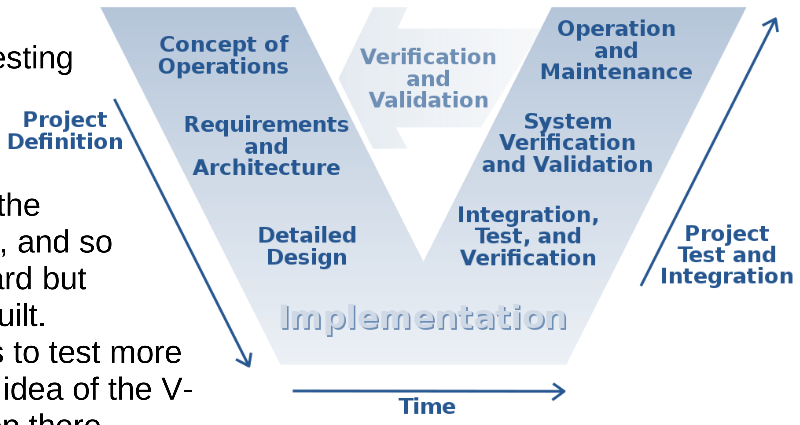
As it turns out, there are many more types of testing than just those three. So when we say the word "test," it is possible that we mean entirely different things.

And when we say "test strategy," we probably mean the combination of test approaches and effort that offers the most value. Today, I will offer a smattering description of different types of testing, along with their pros and cons. After reading this article, software testing novices should have enough familiarity to have a clear discussion. If you are a testing expert, you can forward this article to create a shared understanding or have fun nit-picking at my definitions. Hopefully both! I'll run through the definition, pros and cons, and best use of each, along with a bit on strategy. This is a vast topic, and my goal is to write a version of those "if you only ever read one article on functional software testing" articles.

Here goes, in rough order from a customer to developer.

Types of Functional Testing

The V-Model. This is an older idea in software engineering that every level of testing had a corresponding level of the test. Changes at the code level would be tested with unit tests; at the design level, there would be integration tests; at the requirements level, we have system tests, and so on. The idea is intuitive and straightforward but does not correspond to how software is built. Instead, organizations look for more ways to test more things as often as possible. Knowing the idea of the V-model can be a good start -- just don't stop there.


Concept of Operations
Verification and Validation
Operation and Maintenance
Project Definition
Requirements and Architecture
System Verification and Validation
Detailed Design
Integration, Test, and Verification
Project Test and Integration
Implementation
Time

Acceptance Testing. This is a testing classic, usually defined as User Acceptance Testing (UAT), and implies having someone who is (or acts like) a customer do actual hands-on keyboard testing while using the software. UAT intends less to find bugs and more to get the users to "accept" the system as "good enough" and provide ideas for future features or user interface improvements. Of course, UAT results can send the software back to the programmers for fixing, but most UAT groups intend that the software would be well-tested by other means before even getting to UAT. Getting customers into a zoom meeting to conduct a UAT then finding that they are unable to even login to the software makes the programmers look bad and wastes everyone's time.

End to End (E2E) Testing. This involves testing the entire customer experience of using an application from a customer's perspective. In eCommerce, that might be the entire user journey from creating an account to finding a product, adding it to the cart, and ordering it. When we do this sort of testing, it is tempting to stop before exercising the parts of checkout where API calls take place. Generally, the checkout process is where APIs may validate inventory, return shipping and delivery dates, and validate the credit card information. Ron Jeffries, one of the authors of the Agile Manifesto, is credited with the phrase "end to end is farther than you think." Not running through the complete customer journey can run afoul of the true intent of end-to-end testing.

Feature Testing. This is testing a new feature in some detail. Feature tests often include one-time "what if" questions and difficulties that may often run to confirm that the software did not fall backward or *regress*. Sadly, it is expected that a change performed on one feature may have unintended consequences and break some other feature unexpectedly. This leads to a desire to "double-check" that something outside the feature did not regress after making a change.

Regression Testing. The periodic running through of a wide-ranging battery of tests intends to reduce the risk that a regression (breakage of what worked before) escapes to customers. This can be related to a "final check" or final inspection of the software release, especially in software systems where the entire application is delivered as one "build." Most GUI test automation is considered regression testing.

API Testing. The next level down from the user interface is the Application Programming Interface or API. On the web, these are requests the web page calls to make queries. For example, a web page might call an API to return search results for "Matthew Heusser" to find the books (products) written by Matthew Heusser. API tests define a set of expectations for the API - they work as examples. This is a contract for how the API will operate. Before the API changes, the programmer can run the test to ensure the contract continues. This significantly reduces the chance of a breaking change. Breaking changes then require versioning or coordinated deploys.

Integration Testing. Somewhere between testing just a simple change in code and the entire system is the integration test. Further, this is a level of software testing where individual units and components are combined and tested as a whole. The purpose of this type of testing is to detect defects in the interactions between integrated units. Mocking and test stubs are often employed in simulation of various components while performing this type of testing.

Unit Testing or Micro-tests. Unit tests generally make no sense to end-users and are more confirmatory for developers. A unit test is written before the code provides the programmer the satisfaction that the code does what the programmer expects. That is a good start but does not provide the customer perspective.

In addition to these, then we have the "special" types of tests…

Special Categories

Black-box testing. It understands what is happening to a piece of software from the outside, without any insight into the system's state of internals. Typically, this is a user interface, but it is possible to Black-box test an API or a specific component. Imagine picking an avionic part, such as an airspeed indicator, and hooking it up to a hairdryer that blows air into the tube. The operator can measure if the speed is correct or the speed of response but knows nothing about the internals and how each component within the part functions. Getting a result that is expected is a tacit confirmation that the feature is functioning as intended.

Clear-box or White-box testing. This is testing the software with an awareness of the innards. Debuggers allow a programmer to walk through every line of code that is executed, start or stop a test on a particular line of code, review the values of the variables or even change the value of those variables. Observability is a modern practice that makes testing the software from the outside look like a transparent box by dropping those values and traces of the software to a log.

When to use Black or Clear-box testing. Most companies use Black-box testing for user acceptance testing and final checks. As observability continues to improve, the difference between the two is becoming more marginal. For that matter, as testing in production rises, the difference between testing and production monitoring is dwindling as well.

Testing in Production. This was previously throwing new code on the same system customers use, then rushing to test it before the customers do. Today, it may be possible to create a new branch of code that will only be used by testers, or people in software engineering, or employees. After a few hours, days, or weeks of waiting (and perhaps fixing), that code could be promoted to production, with an ability to identify problems quickly and rollback. This can accelerate the time to production while reducing risk. This idea of getting code to production quickly while managing the risk is sometimes called "Shift-Right Testing."

Other types of testing

I'm not a huge fan of the term "functional testing." In my mind, the purpose of testing is to reduce risk, and it seems unwise to limit the risks to just functional. Other testing disciplines, including performance, load, accessibility, scalability, and security, are all risks that can, and should, be tested for risk mitigation. Performing significant functional testing while neglecting other important testing disciplines is a formula for disaster. It's critically important to look at the big picture and ensure your test coverage aligns with the various risks that your application/customers may face.

For today, we covered functional testing. The rest? That will have to be a conversation for another time -- soon!