Improving Test Data Collection and Management

By **Michael Stahl** - December 23, 2019
https://www.stickyminds.com/article/improving-test-data-collection-and-management

Summary:
There is much published about the data we generate to assess product quality. There is less discussion about the data testers generate for our own use that can help us improve our work—and even less is said about recommended practices for data collection. Test data collection, management, and use all call for upfront planning and ongoing maintenance. Here's how testers can improve these practices.

It would be a safe bet to say that all testers are sometimes jealous of developers. I am not talking about the usual moaning about the lack of respect or being considered a second-tier profession; I am talking about the fact that development is a productive activity.

Code is written, a product is created, and when the product succeeds, there is a direct line between the developers' work to the existence of the product. And what about us testers? What is it that we create? At dark moments, when melancholy sets in, it seems we don't create anything. But is that true?

I think we create two main products: bug reports and data. These are the tangible outcomes of our work. They can be counted, categorized, and analyzed. All the rest comes under the vague and not easily measured term of "product quality."

Quality can only be measured by indirect metrics, and even when these metrics improve, it's hard to prove that the improvement was due to the tester's work. Maybe it was because of the architects who designed such a wonderful product. Or maybe it was the developers, who excelled in writing high-quality code (I refer, of course, to the code that fixed the bugs we found ...).

There is much published on the importance of writing effective bug reports and about the data we generate to assess product quality. There is less discussion about data we generate for our own use—data that can be used to help us improve our work—and even less is said about recommended practices for data collection.

Data Types

At the most basic level, test execution generates one main data type: test results of pass or fail. However, there is a lot of other information that can collected:

- Measurements: Power consumption, execution speed, CPU load, or memory usage
- Details of the test environment: Processor type, memory size, screen resolution, type and size of the hard drive, network connection details, or versions of the OS, drivers, and firmware
- Details of the involved people: Who set up the environment, wrote the test case, executed it, reviewed the results, and updated the results

- Details of the test automation system: Versions of all the framework's ingredients, details of the controller PC when executing tests remotely, or number of retests
- Details generated by the tests: Test time or test logs

And that's just a partial list. This is where the dilemmas start: What makes sense to collect, how often should we collect it, and how much effort should we invest in the collection?

The first rule is to collect and keep only data we have an idea of what to do with later; otherwise, the options are limitless.

Indeed, there is some risk here: We may miss those unusual cases where seemingly unimportant and obscure data is critical for a certain analysis. But this is the exception. Generally, it does not make sense to collect piles of numbers just because we hope that someday we will do something with them. Let's be honest: We don't have enough time to analyze the data we clearly know how to use, so what are the chances that we will find the time to analyze data that we have no idea what to do with?

Having said that, if collecting a piece of information is easy and cheap, we may just as well collect it. "Easy and cheap" means data that does not take a lot of space to keep and the collection of which does not impact the test time, does not load the system under test, and does not call for an additional system to save and retrieve the data.

There are many such data available for collection during test execution, and it's just a matter of deciding to save them in an orderly fashion. A good example is test execution time—it's easy to collect and can be saved in the same database record that holds the test result.

An opposite example is from testing computer vision systems. Images arrive from the video camera at 30 frames per second, and sometimes even faster. As part of the test case, we calculate some metric on each frame; for this example, let's say we calculate the processing time of the frame. Should we collect this per-frame information, as we already have it available during the test? Just one test case, running a video of one minute, generates 3,600 results. And surely we run more than just one test. Saving the many thousands of data points calls for installation of a high-speed database and for writing code to help load the data efficiently.

Once we have all these numbers available, what would we do with them? If all we will use them for is to calculate the average, minimum, and maximum frame processing time, we may as well calculate these on the fly while running the test and save only these statistics. This is, of course, a compromise. If we did save all the numbers, we could analyze how the processing time changes while processing a long series of frames. This is something we won't be able to note when only looking at the average. Analysis may show that our algorithm has a hard time coping with a specific type of image data, and this may lead to an improvement in the algorithm. This sounds really great—if we ever get to do this analysis.

The decision about what to collect and save is not an easy one. Too little and we will regret it later; too much and we will pay dearly for collecting information that just accumulates in ever-increasing piles.

Data Quality

It's important to ensure the data is correct. This seems like an obvious requirement, but it is not always easily achieved.

To begin with, when collecting data, there is some software in the loop. And software, as we all know, does not work perfectly. Moreover, it's software that we wrote, and for some reason, once we are talking about *our* code, we tend to forget that it needs testing. Even when we do test our code, we are never as creatively destructive as when we test the product. The result is that in many cases, our product (the data) suffers from quality issues.

Here are some such examples:

- Measurements: Measuring the wrong thing, having the wrong definition of the measurement (e.g., average instead of maximum), measuring at the wrong frequency, or creating a "probe effect"
- Details of the test environment: Missing the important data (e.g., saving the CPU type but not the L1 cache size, or not saving the screen resolution in tests that are impacted by resolution)
- Details of the involved people: Recording who *should* execute the test, but not who *actually* executed it
- Details of the test automation system: Not considering any of it as worth saving
- Details generated by the tests: Overly verbose or overly cryptic logs, or logs that are difficult to analyze automatically

Data Collection

Let's assume we successfully executed the first two challenges: All the important data were collected, and it's of good quality. The next stage is to save this data.

I am skipping over the discussions (which may take weeks of heated arguments) about what type of database to use and which data collection technology is right for the job. Instead of trying to figure out the fine details of data storage technologies, I recommend investing in achieving clear and detailed understanding of the expected data flow and accurately defining what we will do with the data. Then let the system people (or DevOps) figure out the technical details that will best support our needs.

A good definition of intended use comes before the considerations of the technical aspects. Here are some questions to ask:

- Where do the data arrive from and who will load them to the database? What should happen when something goes wrong? How can we avoid data loss? Can we tolerate some level of data loss?
- After loading to the database, will we ever need to make changes to the data? Who will be allowed to do that? How frequently do we expect this to happen? Do we need a meticulous recording of the changes? What history do we need to keep?

- How would we extract the data? How many times a day? Will we have manual or automated extracts? What volume? How critical is the extract speed?

Another aspect of planning data storage solution is the connectivity to other systems and databases and avoiding inconsistencies. For example:

- Data is duplicated in a number of systems and can be modified in each of these systems. Does the solution need to provide synchronization?
- Data is spread across a number of systems. Would we have a need to extract and join data from a number of sources? Can this be done efficiently with the proposed storage design?

Having the data spread across a number of related systems opens the field for many small and avoidable small mistakes, which, unless prevented right from the start, will annoy and hamper users for years. Here are two examples I encountered.

In one, test results were collected in one database, then ported to another. In the first database, the results field was called "Results," but in the second, it was called "State." The naming difference caused ongoing misunderstandings and communication mistakes.

In another situation, a field in the bug reporting database was used to record the name of the module where the bug was found. The field used a choice list for the module names. In the test case management system, each test was associated with the module it covered, using again a field with a choice list of module names. I will give you one chance to guess if the lists were synchronized, with a single, common name for each module.

The system people who build and maintain the support systems are not always aware of the connections needed between the systems. As users, it's our job to pay attention to these details.

Data Access

Data collection is creating our "product," the information. But the information by itself is of little value unless we do something useful with it. To be able to do so, we need easy access to the data. I think it is safe to say that data that is hard to access will not be used.

The decision on how the data is stored impacts how easy it is to access it. An implementation that is efficient from one aspect (e.g., storage size) may be inefficient from another aspect (e.g., data retrieval time). This is yet another reason why it is critically important to define what we plan to do with the data prior to defining the technical solution.

For example, some databases support history by creating a new record each time anything is updated in the record. Others keep only one record and save a trace of all changes. The second implementation is much more efficient from a storage point of view, but re-creating the historical state of a record in the past is not trivial at all. Test yourself: How easy is it for you to find all the bugs in your database that at some point failed confirmation tests and were reopened?

Here's another example. When attempting analysis using big data methods, you need to be able to pull millions of records from your data repositories. How easy is it? In big data analysis, we look for unintuitive connections between data, so we will want to extract all available fields. Do your extract systems hide some information because the system people thought it is unimportant data?

Beyond the extraction, testers also need access to tools that allow manipulation of large quantities of information. Anyone who tried to do analysis on a million rows of data using Excel has concluded that a different tool is needed. The availability of such tools is a prerequisite for meaningful use of the data we collect.

Maintenance

Even if we did all the upfront analysis and defined our requirements for data storage in excruciating detail, we will always find that we need changes. Information we thought will be critical and important turns out to be meaningless. Data that is entered manually is not as updated or accurate as we hoped it will be. Carefully selected values in choice lists turn out to be ambiguous to the users. Other fields are consistently left empty or with the default value. Even automatically collected data can have defects—wrong definitions can cause a loss of accuracy (e.g., defining a field as Integer while the data is Real), or if some date fails to load or is frequently missing. Additionally, as time goes on, new needs appear, new ideas are suggested, and the system people are swamped with requests for additions and changes to the databases.

All this activity must be monitored and managed. It is obvious that someone needs to check that data loads correctly when the system is launched, but it needs to be monitored at some frequency to ensure the system continues to work as planned.

Someone needs to discuss all the requests—which sometimes contradict each other—and decide what to accept, what can be supported by the systems as is, and what calls for a larger modification. Someone needs to review the defined data fields and remove fields that seemed like a great idea at the time but are left unused. It is important to define this managing activity: who owns it, how requests are handled, who is the final decision maker, and who ensures the data we collect is not just digital garbage.

Test data collection, management, and use all call for upfront planning and ongoing maintenance. It won't happen by itself, and confirming it is done correctly and efficiently ensures the usefulness of the two main products of the testing activity.