Software Quality for a Warfare

Zlatko Michailov
31-Mar-2005

This article explains how to achieve a high quality in software product development. It uses examples and techniques from the software industry. However, the concept is easily applicable to products from any other industry with only minor (if any at all) modifications.

Introduction

Every software project manager knows that a product must be rigorously tested before released to market. So does every software project stakeholder. And, most likely a product is tested according to its quality plan proposed by the project manager and approved by the project stakeholders.

Then why do software products still misbehave – cannot sustain extra workload, cannot work for too long, work too slowly, or don't do certain things the way they are supposed to? Apparently project stakeholders know *what* is needed – a quality plan. What is missing is *how* to create and follow a good quality plan.

This article outlines the concept of a solid quality plan that covers four aspects of a software product, which in most cases should be sufficient to guarantee the behavior of the product according to its specification. It also provides a procedure for applying the designed quality plan.

Concept

When a product is released to market it has two main objectives:

- Minimum – *defend* the market share held by its company.
- Maximum – *conquer* some rival market share.

It is as simple as that. Every business has the main objective of generating profit for its owners/shareholders.

Once a software product is deployed to a user's computer, it is on its own - no more quick fixes or configuration tweaks. This is no demo. This is *action*. The battle to win a customer's heart is fiercer than the battles between competitors. Legislation may regulate business competition but no legislation dictates how customers should spend their money. That is a battle with no rules. The only way to win is to sustain any attack – fair or not.

The above software deployment pattern matches the pattern of deploying a *soldier on a mission*. Assuming the latter has a long history in our civilization and puts real human lives at stake, we can conclude that the military *development process* is suitable for adoption by the software industry.

There are four properties a soldier must develop in order to be ready for combat:

- Accuracy – besides shooting this includes knowledge, memory, analytical skills, etc.
- Strength - the maximum weight the soldier can lift/carry.
- Fastness - the soldier's time on 100m, 1 mile, a cross-country track, etc.
- Endurance - how long the soldier can operate carrying combat equipment without taking a rest?

All of those four military properties map easily to software terms as follows:

- Conformance - does the product's behavior match its specification?
- Load - the maximum load the product can sustain without crashing.
- Speed - the time it takes to perform certain operations.
- Endurance - how long the product can sustain a moderate load without crashing?

Let's call the sum of those four military/software properties the *Breadth Performance Profile* of a soldier/product.

Now, let's look at the situation from a slightly different angle – in what types of environment a soldier must train and be tested? There are three main ones:

- Lab/Gym - purified environment for best brain/muscle performance.
- Park - environment similar to Field but without any real danger.
- Field - stressful environment with real combat elements.

Let's call the soldier's performance in all those three types of environment his *Depth Performance Profile*. The purpose of maintaining a depth performance profile is to effectively troubleshoot unsatisfactory performance. For instance, if a soldier's swimming performance is not good enough in the sea but is excellent in an indoor pool, most likely the problem has nothing to do with his swimming skills. Perhaps he cannot handle the water temperature, the waves, the fear of drowning, or something else related to the outdoor environment.

Respectively a software product's depth performance profile consists of:

- Kernel - database transactions, I/O operations, computations, etc.
- Business Logic- whole workflows and parts of them at the middle tier.
- User Interface- complete end-user workflows.

Again, as in the military case, we are looking for performance deviations. It is possible to have a poor end-user performance while at the middle-tier it is satisfactory. That means we may be generating too much network traffic, or the UI rendering engine is too slow. It is very important to know where to look for a problem before starting to destroy things that work fine.

The explanation so far should have clarified the concept of adopting the military development process. What is left is to create the breadth- and depth performance profile templates of a software product, and a procedure that verifies the four breadth properties have been developed at each depth level. Simply put – *training* and *testing*.

Training

There is no recipe how to achieve a high quality. The main idea is *to remember that tests will be performed and the product must pass*. In that respect it is a good idea

to start testing as early as possible in the lifecycle, before everybody has forgotten, and to schedule testing on a regular basis, e.g. every day/week, as a reminder.

Testing

Evolution

Testing any of the four breadth properties at any depth level consists of executing multiple test workflows. It is recommended to keep adding new workflows along the way to constantly increase test coverage. Do not remove a test workflow unless the feature has been dropped – that is to detect regression.

Scorecard

A scorecard must be maintained where every test workflow has an entry. It is recommended that the test workflows are grouped by breadth property and depth level. It is up to the project team to choose which dimension should be leading. Next to each test workflow the current best result is recorded as a *Bottom Line*. Previous bottom-line results may optionally be kept to track progress. When a test workflow is executed, the result is compared with its corresponding bottom line:

- If the absolute difference is less than a certain threshold, the bottom line remains unchanged and the test is considered to have passed.
- If the result is better than the bottom line with a difference bigger than a certain threshold, the bottom line is updated to the new result and the test is considered to have passed.
- If the result is worse than the bottom line with a difference bigger than a certain threshold, the bottom line remains unchanged and the test is considered to have failed.

In exceptional cases when a newly added important feature worsens the results of one or more other features, a manager may sign off a worse bottom line for the affected feature judging that having the new feature is worth the worse performance.

The threshold is a percentage – the value is the same for all tests. It should be chosen based on the nature of the product. It should be big enough to filter out noise fluctuations, and small enough to detect misbehavior as early as possible. *10%* is recommended as a starter.

Iterations

To avoid random factors, each test workflow must be executed multiple times, e.g. 10, 20, or more. The final result should be the average of all iteration results. It is acceptable to skip the result of the first iteration – that is when program modules are loaded and/or memory buffers are initialized.

Always watch for a trend among the iteration results. It is possible that garbage never gets disposed of and the system looses resources over time.

Environment

All tests are performed on the same hardware- and software platform isolated from any network traffic. If a platform component has to be changed, all test workflows are executed against the *Latest Good Build* and the new results are recorded as bottom lines.

The latest good build is the one that has passed all test workflows last.

Breadth Testing

Conformance Testing

Conformance testing verifies whether a product behaves as it is specified. Conformance testing is unique with regard to the test results – they are all binary – a product's behavior in a test workflow either matches its specification or not. Conformance to the specification is a prerequisite to any other quality property. That's why there may be no deviation within any threshold – the match must be exact.

Load Testing

Load testing tries to find the load boundaries of the system. It simulates different kinds of load until the system crashes or makes a mistake of any sort. Examples of load are: number of concurrent users, number of requests per second, number of bytes per second, etc.

Tracking the load limits of a software product has two benefits:

- We verify whether the product meets its load requirements or not. If so, we may also estimate how far we could eventually expand the business without modifying the code.
- We detect bad algorithms as soon as they are submitted to the baseline, since algorithm modifications are very likely to affect performance results. Equally well we detect algorithm improvements and recognize their authors' good job.

Speed Testing

Speed testing is logically equivalent to load testing but in a different dimension. It measures the time it takes for a software product to perform certain operations. For instance: 1 user to perform operation A 10 times sequentially, 10 users to perform operation B concurrently, etc.

Again, the benefits are virtually equivalent to load testing:

- Verify speed-related requirements and estimate possibilities for business expansion.
- Justify algorithm modifications.

Endurance Testing

A software system's endurance is crucial to its operational cost – a factor project stakeholders are very sensitive to. The ultimate goal is to make a given software

system maintain a sufficient level of resources at all times and even repair itself after abnormal events without any human intervention.
Normally endurance testing consists of one long and complex test workflow for each depth level. That test workflow should resemble a production environment as close as possible – it should generate load slightly above expected/existing in the same peak/dip pattern. The test workflow should run for a period of time long enough to make a human intervention (financially) acceptable. If it is impossible to run the test for such a long time, the duration may be shortened and the load increased respectively.

Depth Testing

Besides verification, depth testing has an architectural advantage – it pushes the system design towards well-defined layers with public APIs. In fact, layering a system properly is more challenging than designing test workflows for each layer.

The test workflows at each level should repeat the operations generated for workflows of the adjacent outer level and eventually add some extra operations specific to the tested level. That way, if performance problems are discovered, they can be localized immediately. So, it is a good idea to start designing test workflows from the user-level going deeper.

What may turn out challenging is finding a technology that drives and recognizes GUI at user-level. The non-UI levels are easier to test using any scripting technology.

Conclusion

Developing a software product is often compared to construction. The two *processes* are indeed quite similar. However, this article is about the final *product*, which is much closer related to a soldier on a mission. Like construction, warfare is a business with a long history and a solid process. Unlike construction products, warfare products operate with the highest possible cost at stake – human life – and therefore must not fail. If not perfect, the military approach to software quality is at least suitable for adoption with a very good chance to improve the robustness of the final products.