Planning the Endgame

Fiona Charles

Originally published on stickyminds.com August 28, 2007.

We know that the elapsed time for testing will ultimately be decided not only by the number of hours we schedule for our team but also by the quality of the system we get to test, the development team's turnaround time for bug fixes, and the stakeholders' appetite for risk.

This is the endgame: the interplay of tests, builds, bug fixes, and retests plus regression tests. Unfortunately, project managers, even experienced ones, can fall into the trap of planning only for testing—forgetting to take the whole endgame into account.

A test manager can help the project manager build a credible case for the amount of testing time you need by modeling a plan for the endgame.

I start by coming up with a number representing test cases. That gives me the most important unit of measure, and a starting point for other planning assumptions.

In my planning, a "test case" is just a handy unit of measure, representing one significant thing we're going to do to test some aspect of the system.. The definition and size vary according to the system we're testing. Although the actual sizes of test cases for any system will vary widely, it's usually possible to come up with an average unit that will hold up well enough for planning purposes.

During test design, I work with my team to define what we mean by "test case," usually by analogy.

> "A test case is a thing like this, about this big."
>
> "We think it will take about this long to develop an average test case."

When we have several developed, we can say,

> "We think it will take about this long, on average, to execute an average test case, including setting up the data, taking notes, and entering bugs."

Test cases don't have to be documented. I can use the same unit to allow for a mix of predesigned and exploratory tests in the plan. Having a consistent unit means I can apply some planning assumptions to undocumented tests. I add contingency to the number, and for test execution I also raise it by some percentage for exploratory testing.

For test execution, I first try to estimate the number of test cases we can attempt in a week. Several assumptions go into that, including average time to execute a test case, productive tester hours per day, and the number of testers. I also add some factor for environment down time, plus or including builds, depending on the disruption time expected from routine builds.

The calculations look like this:

Work days in a week	5
(times) Productive hours in a day, per tester	6
(times) Number of testers	5
(minus) Environment downtime	10%
Average hours/week available for testing	**135**
(times) Average test cases executed/hour	3
Average test cases executed/week	**405**

Let’s say we have 1,200 test cases. If we plan only for actual test time, it could appear as if we would complete testing in about three weeks.

In reality, the test team won’t reach full productivity in the first couple of weeks. If we estimate a productivity hit of 25 percent in weeks one and two, we will actually only execute about 300 test cases in the first two weeks.

Regardless, we will be finding bugs, so next I estimate how many bugs we expect to find. Let’s say I expect an average of one bug logged for every three test cases executed. A test case that finds a bug won’t pass and will have to be re-executed at least once more.

Week one starts to look like this:

WEEK 1		**Test Cases to Pass**
		1200
Average hours/week available for testing	135	
(minus) First week startup	25%	
Test cases executed	**304**	
Bugs found	**101**	
		998

Of the bugs we find, some number – say one in three -- will be severity one or two, and therefore critical to fix and retest. I also assume that around a third of the lower-severity bugs will be fixed. I ask the development leads for average times to fix bugs. If they can’t supply an estimate (which is often the case), I’ll propose one, say seven bugs per developer per week. . The development plan should tell me how many developers will be available to fix bugs, say five developers.

Now Week One looks like this.

WEEK 1		**Test Cases to Pass**	**Bugs to Fix**
TEST CASES		1200	
Average hours/week available for testing	135		
(minus) First week startup	25%		
Test cases executed	**304**		
		998	
BUGS			
Bugs found	**101**		
% of bugs that are severity 1 or 2	33%		33
% of low severity bugs to fix	33%		22
Bugs to fix			**55**
Bug fixes/developer/week	7		
(times) Developers fixing bugs	5		
Bugs fixed	**35**		**20**

We go into week two with almost 1,000 test cases still to execute and pass and twenty open bugs.

If developers aren't added to fix bugs, the number of open bugs continues to rise. Week two ends with forty open bugs; week three (when we are executing more test cases and finding more bugs) ends with seventy-nine. And so on.

By the end of week four, we will have run through all the test cases once and will be finding fewer new bugs than during the first couple of weeks, at which point the emphasis shifts almost entirely to bug fixing and retesting. When that is complete, so is the endgame.

That's the simple model, which provides a rough idea of the end date. You can refine the model by adding more variables reflecting expected reality.

For example, some of the bugs fixed, say one in ten, will fail retest and go back into fix. When planning, I include a few other assumptions, such as a number of reported non-problems that take everybody's time, and a number of environment problems, especially in the first couple of weeks.

Like all models, this one has limitations. For one, the calculations tend to be linear while projects aren't. But on several projects I have found that my models weren't far from reality. I once used the model in a phone conversation to convince a project manager and the client's CIO (whom I'd never met) that we needed three more months to test than they had planned for. Impressed by the detail in the model, they bought my plan. In the end the model was short by three weeks, but I didn't think I'd done too badly on a four-month plan—given that the development team decided to do an infrastructure upgrade in the last month that delayed everything.

When we use a planning model like this, based on assumptions built from estimates, it's important for everyone to understand that it is a heuristic device. Estimating endgame activities, including testing, is about as accurate as estimating development or any other project work. The more we know, the closer we can get to something that might work out in reality. But our numbers will always be estimates rather than exact predictions.

Merriam Webster's Online Dictionary has this definition for "heuristic":

> "Of or relating to exploratory problem-solving techniques that utilize self-educating techniques (as the evaluation of feedback) to improve performance"

This definition is a useful reminder that once we have modeled a plan, it's essential to track actuals so we can continually check every assumption as we go through the endgame and adjust the plan accordingly.

Modeling the endgame helps test managers and project managers estimate an end date that the project has a reasonable chance of achieving. That's the date when we should have passed all the test cases and fixed and retested all the bugs that must be fixed. Most importantly, because the model provides an explicit set of assumptions, it's hard for project managers and others to argue that we need less time to test, and it's easier for us to argue for more time if the assumptions turn out to be wrong.