MANAGING THE PROPORTION OF TESTERS TO (OTHER)[1] DEVELOPERS

Cem Kaner, J.D., Ph.D.
Florida Institute of Technology

Elisabeth Hendrickson
Quality Tree Software, Inc.

Jennifer Smith-Brock
Ajilon Software Quality Partners

Pacific Northwest Software Quality Conference[2]

October 2001

ABSTRACT

One of the common test management questions is what is the right ratio of testers to other developers. Perhaps a credible benchmark number can provide convenience and bargaining power to the test manager working with an executive who has uninformed ideas about testing or whose objective is to spend the minimum necessary to conform to an industry standard.

We focused on staffing ratios and related issues for two days at the Fall 2000 meeting of the Software Test Managers Roundtable (STMR 3).[3] This paper is a report of our thinking. We assert the following:

---

[1] In most companies, testers work in the product development organization and they are part of the technological team that develops software products. Testers *are* developers. The ratios that we are interested are the ratio of testers to the *other* developers on the project.

[2] An earlier version of this paper appeared in the 2001 Proceedings of the International Software Quality Week.

[3] Software Test Managers Roundtable (STMR) meets twice yearly to discuss test management problems. A typical meeting has 15 experienced test managers, a facilitator and a recorder. There is no charge to attend the meetings, but attendance must be kept small to make the meetings manageable. If you are an experienced test manager and want to join in these discussions, please contact Cem Kaner, kaner@kaner.com. The meeting that is the basis for the present paper was STMR3, in San Jose, CA. Participants included Sue Bartlett, Laura Anneker, Fran McKain, Elisabeth Hendrickson, Bret Pettichord, Chris DeNardis, George Hamblen, Jim Williams, Brian Lawrence, Cem Kaner, Jennifer Smith-Brock, Kathy Iberle, Hung Quoc Nguyen, and Neal Reizer.

- One of the common answers is 1-to-1 (1 tester per programmer) or that 1-to-1 is the common ratio in leading edge companies[4] and is therefore desirable. Our experience has been that (to the extent that we can speak meaningfully about ratios at all) 1-to-1 has sometimes been a good ratio and sometimes a poor one.
- Ratios are calculated so differently from project to project that they probably incomparable.
- Project-specific factors will drive you toward different ratios, and toward different ratios at different times in the project. Such factors include (for example) the incoming reliability of the product, the extent to which the project involves new code that was written in-house, the extent to which the code was subjected to early analysis and review, the breadth of configurations that must be tested, the testability of the software, the availability of tools, the experience of the testers and other developers, corporate quality standards, and the allocation of work to testers and other developers.
- More is not necessarily better. A high ratio of testers to programmers may reflect a serious misallocation of resources and may do more harm than good.
- Across companies, testers do a wide variety of tasks. The more tasks that testers do, the more tester-time is needed to get the job done. We list and categorize many of the tasks that testers perform.
- The set of tasks undertaken by a test group should be determined by the group's mission. We examine a few different possible missions to illustrate this point.
- A ratio focuses executives on the wrong thing. The ratio is a comparison of body counts. For this many programmers you need that many testers. The ratio abstraction focuses attention away from the task list that drives up the staffing cost. It conveys nothing about what the testers will do. Instead it focuses attention onto a pair of numbers that have no direct link to anything but each other. At this level of abstraction, it even sounds meaningful to say, "Last time, your staffing ratio was at 1.2 to 1. You need to work smarter, and so I am setting you an objective of 10% greater efficiency. Go forth and become 1.08 to 1." The ratio is likely to distract attention from the important questions, such as: What will you *not do* in order to achieve that 10% reduction? And what task list is appropriate for you in the context of this project?

THESE RATIOS ARE INCOMPARABLE

What do we mean when we refer to a 1-to-1 ratio of testers to other developers? Across groups or even across projects by the same groups, these ratios can have wildly different meanings.

Consider the following stories:

---

[4] We thank Ross Collard (1999) for providing us with a summary of his interviews of senior testing staff at 18 companies that he classed as "leading-edge," such as BMC Software, Cisco, Global Village, Lucent, Microsoft. Six companies reported ratios of 1-to-1 or more, and the median ratio was 1-to-2.

Jane manages a project with the following personnel:

Staff	Counted as
4 programmers	programmers
1 development manager	programmer
1 test lead	Tester
1 black box tester	Tester
2 test automation engineers	Testers
1 buildmeister	Tester

According to the numbers, there's a 1-to-1 ratio between programmers and testers. However, when a new build comes into the test group, only one person is available to test it full time—the black box tester. Because of the apparent 1-to-1 ratio, management is puzzled by how long it takes the test group to do even simple tasks, like accept or reject a build. Jane is hard-pressed to explain the bottleneck to management—they keep coming back to the 1-to-1 ratio and insisting that means there are enough testers. The testers must be goofing off.

Now consider Carl's dilemma. His staff looks like this:

Staff	Counted as
1 programmer	programmer
1 toolsmith	programmer
1 buildmeister	programmer
1 development lead	programmer
1 development manager	programmer
1 test lead	Tester
4 black box testers	Testers

According to these numbers, there are 5 programmers and 5 testers, a comfortable 1-to-1 ratio. The testers report dozens of bugs per week. However, because they have no access to the source code (they test at the black box level), they cannot isolate the bugs they report. It takes the programmers significant time to understand and fix each reported issue. Carl is hard-pressed to explain why the testers can find bugs faster than his staff can fix them. Are his programmers lazy?

Sandy's department provides even more counting challenges:

Staff	Counted as
5 programmers	programmer
5 on-site consultants (doing programming)	???
1 project team (10 people of various specializations) who are under contract with Sandy's company to write and deliver a series of components to be used in Sandra's product.	???
1 full-time, on-staff test engineer	tester
3 technicians (they work for Sandry's company, are supervised by the engineer, but have limited discretion and experience)	???
3 temporary technicians (they work for a contracting agency, not Sandy's company, they report to the test engineer, but are not counted in the company's headcount)	???
3 testers who work offsite in an independent test lab	???

Should we count consultants as programmers? What about programmers who work for other companies and are simply selling code to Sandy's company? Should we count technicians as testers? What about technicians or other testers who work for other companies and provide testing services under contract? We don't know the "right" answer to these questions. We do know that different companies answer them differently and so they would calculate different ratios (ranging from 1-to-10 through 10-to-1) for the same situation.

Here are even more of the classification ambiguities in determining the ratio of testers to programmers:

- Are test managers testers? Are project managers developers? What about test leads and project leads? If a test lead sometimes runs test cases, should we count her as a tester for the hours that she is hunting for bugs? What about the hours she spends reviewing the test plans of the other testers?

- When programmers do code reviews, they find defects. Should we count them as testers? Imagine a six-month project that has one officially designated tester and ten officially designated programmers. In the first four months, the programmers spend 60% of their time critically analyzing requirements, specifications, and code, doing various types of walkthroughs and inspections. They find lots of problems. (In the other 40% of their time, they write code.) The tester also spends 60% of her time reading and participating in the meetings. Her other 40% is spent on the test plan. For these four months, should we count the ratio of testers to programmers as 1-to-10 or as 7-to-4? (After all, didn't the programmers spend 6 person-months doing bug hunting and only 4 person-months writing code?)
- If the testers write diagnostic code or tools that will make the programmers' lives easier as well as their own, are they working as testers or programmers?
- Imagine a six-month project that starts with four months of coding by ten programmers. During this part of the project, there are no testers. In the last two months, there are ten testers. Should we count this as 10-to-10 ratio or 60-to-20? (After all, there *were* 60 programmer-months on the project and only 20 tester-months.)
- Suppose that your company spends $1,000,000 licensing software components. These components required 36 programmer-months (and an unknown number of tester-months) to develop. Your company uses one programmer for 6 months to write an application that is primarily based on these components. It assigns one tester for 6 months. Is the ratio of testers to programmers 1-to-1 or 1-to-7 or something in between?
- How should we count technical writers, tech support staff, human factors analysts, systems analysts, system architects, executives, secretaries, testing interns, programming interns, marketeers, consultants to the programmers, consultants to the testers, and beta testers?
- If the programmers dump one of their incompetents into the testing group and one of the testers has to work half-time to babysit him, did the ratio of testers to programmers just go up or down? In general, if one group is consistently more (or less) productive than industry norm should we count them as if there were more (fewer) of them?

The answers to these questions might seem to be obvious to you, but whatever *your* answers are, someone respectable in a respectable company would answer them quite differently. At STMR 3, we marveled at the variety of ways that we counted tester-units for comparison with programmer-units. Because of the undefined counting rules, when two companies (or different groups in the same company) report their tester-to-programmer ratios, we can't tell from the ratios whether a reported ratio of 1 (tester) to 3 (programmers) involves more or less actual quality control than a ratio of 3 (testers) to 1 (programmer).

To put this more pointedly, when you hear someone claim in a conference talk that their ratio of testers to programmers is 1-to-1, you will probably have no idea what that means. Oh, you might have an idea, but it will be based on *your* assumptions and *not their*

situation. Whatever your impression of the staffing and work-sharing arrangements at that company is, it will probably be wrong.

ARE LARGER RATIOS BETTER OR WORSE?

Most of the participants at STMR 3 (including us) had worked on projects with high ratios of testers to programmers, as many as 5 testers per programmer. Most of us had also worked on projects involving very low ratios, as few as 0-to-7 and 1-to-8. Some of the projects with high ratios had been successful, some not. Some of the projects with low ratios had been successful, some not. This corresponds with what we've been told by other managers, outside of STMR.

Why are some projects successful with very few testers while others need so many more?

Low Ratios of Testers to Programmers

Most testers have seen or worked in a test group that was flooded with work and pushed up against tight deadlines. These groups typically staff projects with relatively few testers per programmer. The work is high stress and the overall product quality will probably be low.

However, some projects are correctly staffed with low ratios of testers to programmers. In our experience, these projects generally involved programmers (and managers) who had high quality standards and who didn't rely on the test group to get the product right.

Projects with low ratios of testers to programmers might occur:

- routinely, in a company with a healthy culture whose projects normally succeed
- routinely, under challenging circumstances
- on a project-by-project basis based on special circumstances of that project

Healthy Culture

Healthy cultures that have successful projects with relatively few testers often have characteristics like these:

- The test group has low noise-to-work ratios. "Noise" includes wasted time arising out of organizational chaos or an oppressive work environment.
- Staff turnover in the testing and programming groups is probably low. It takes time for testers to become efficient with a product--time, for example, to gain expertise and to build trust with the programmers.
- The company focuses on hiring skilled, experienced testers rather than "bodies."
- There is a shared agreement on the role of the test group, and little need for ongoing reevaluation or justification of the role.
- There is trust and respect between programmers and testers, and members of either group will help the other become more productive (for example, by helping them build tools).

- Quality is seen as everyone's business. The company emphasizes individual accountability. The person who makes a bug is expected to fix it and to learn something from the experience. There is a low churn rate for bugs--they don't ping-pong between programmers and testers ("Can't reproduce this bug", "I can", "It's not a bug anyway", "Marketing says it is", etc.)
- The code coming into testing is clean, designed for testability, and has good debug support.
- There may be an extensive unit test library that the programmers rerun whenever they update the product with their changes. The result is that the code they give to testers has fewer regression errors and needs less regression testing (Beck, 2000)
- In general, there is an emphasis on prevention of defects and/or on early discovery of them in technical reviews (such as inspections).
- In general, there is an emphasis on reuse of reusable test materials and on intelligent use of test tools
- The expectation is that reproducible coding errors will be fixed. Testers spend relatively little time justifying their test cases or doing extensive troubleshooting and market research just to convince the programmers that an error is worth fixing.
- The culture is more solution-oriented than blame-oriented.

Challenging Circumstances

Some companies need much more testing than they conduct, but they might not do it because:

- The product might be so complex that it is extremely expensive to train new testers. New testers won't understand how to test the product, and they'll waste too much programmer time on unimportant bugs and misunderstandings.
- They may have decided that time to market is more important than finding and fixing defects.
- They are in a dominant market position in their niche and their customers will pay extra for maintenance and support. There is thus (until competitors appear) relatively little incentive to the company to find and fix defects before release.
- They believe that it's right and natural for people in high tech to work 80+ hour weeks for 6+ months. In their view, adding staff will reduce the free overtime without increasing total productivity.
- The testing group may be perceived as not contributing because they aren't writing code.
- Other members of the project team might believe that testing is easy. ("What's so tough about testing? Just run the program! I can find bugs just by installing it! In fact, we should just bring in a bunch of Kelly temps to do this.")

- Testers might be perceived as too expensive.
- Testers might be perceived as incompetent, counterproductive twits.
- The test manager might be perceived as a whiner who should use his staff more effectively.
- The test group's work might be perceived as poor, with an overemphasis on unimportant issues ("corner cases") and the superficial aspects of the product.
- The testing group may have little credibility. They are seen as politically motivated and being preoccupied with irrelevant tests (e.g. some extreme corner-case tests). Therefore they are not sufficiently funded.
- The relation between testers and programmers may be toxic, resulting in excessive turnover in the testing group.

Some companies will never develop respect for their testing staff, and will never staff the test groups appropriately, no matter how good the testers or test managers. But in many other companies, testing groups build their own reputations over time. Some testing groups work too hard to increase their power and control in a company and not hard enough to improve their credibility and their technical contribution. Down that road, we think, tight staffing, high turnover, and layoffs are inevitable.

Project Factors

To some degree independently of the corporate culture, some ***projects*** are likely to succeed with few testers because of factors specific to those projects. For example:

- The product might involve low risk. No one expects it to work well and failures won't harm anyone.
- There might be little time-to-market pressure.
- The product might come to the test team with few defects (perhaps because this particular project team paid a lot of attention to the design, did paired programming or did a lot of inspections, etc.)
- The code might be particularly easy to test or relevant test tools that the testers are familiar with might be readily available.
- There might be no need to certify this product, no need for extensive documentation of the tests or (except for bug reports) the test results, and no requirement for detailed evaluations of the final quality of this product.
- The testers might simply not have much work to do on this project because it is easy, reliable, intuitive, testable, etc.

High Ratios of Testers to Programmers

We've met testers who respond enthusiastically when they hear of a group that has a very high ratio of testers to programmers. The impression that they have expressed to us is that such a high ratio must indicate a corporate commitment to quality, and a healthier lifestyle (less stress, less grinding overtime) for the testers.

In many cases, though, a high number of testers results from (and contributes to) dysfunction in the product development effort.

One of us worked on a project that had roughly three times as many testers as programmers by the end of the project. The programmers were under intense time pressure—and they couldn't help but notice the large pool of people next door just waiting to catch their mistakes. The result? The bug introduction rate skyrocketed. One programmer commented about a particularly buggy area of the program under test, "Oh, yeah. I knew there would be bugs there—I just didn't have time to look for them myself."

Programmers find the vast majority of defects in their own code before they turn it over for testing. When a programmer finds a bug in her own code, she can usually isolate it quickly. She doesn't have to spend much time documenting the bug, replicating the bug, tracking it, or arguing that it should be fixed. When programmers skimp on testing, testers must spend much more time per bug to find, isolate, report, track, and advocate the fix. And then the programmer wastes time translating a black box test result back to code.

We suggest that there is a significant waste of project resources whenever an error is found by a black box tester that could have been easily found by the programmer using traditional glass box unit testing techniques. Some of these errors will inevitably creep through to testing, but we think staffing and lifecycle models that encourage over-reliance on black box testers are pathological.

Having an army of testers can encourage a spiraling drop in productivity and quality. (We talk more about this in Hendrickson, 2001, and Kaner, Falk & Nguyen, 1993, Chapter 15).

The best solution for severely buggy code is not to add testers. The best solution might be to freeze (or even reduce) the size of the testing group while adding programmers. The programmers should fix and test code, not add even more buggy features.

Healthy Cultures

Some companies need more testers because of the market they are in or the technology they use. The examples below might describe the culture of the company or the circumstances of a particular project. Examples:

- Much more formal planning, documentation, and archiving of all artifacts of the testing effort is needed when developing safety-critical software. Heavy documentation might be required for other software because of regulatory agency interest or high litigation risk.
- Extensive documentation may also be needed for software that will be sold in its entirety to a customer, with the expectation that the customer will assume responsibility for future maintenance, support and enhancement.
- Some markets are particularly picky about fit and finish errors or are more likely to expect / demand technical support for problems that customers in other markets

might seem small or easy to solve. If you are selling into that market, you'll probably do much more user interface testing and much more scenario testing.

- Extensive configuration testing is needed for software that must work on many platforms or support many different technologies or types of software or peripherals.
- Load testing is needed for software that is subject to bursts of peak usage.
- Some companies hire domain experts into testing or train several testers into domain expertise. These testers become knowledgeable advocates for customer satisfaction improvements and are particularly important in projects whose designs emerge over time.
- Some testing groups have a broad charter. Along with testing, they provide several other development services such as debugging, specification writing, benchmarking competing products, participation in code reviews, and so on. The broader the charter, the more people are needed to do the work.
- If the company relies on outsourced testing (this is sometimes a requirement of the customer's), there is substantial communication cost. The external testers need time to understand the product, the market, and the risks. They also need significant support (people to answer questions and documentation) from in-house testing staff.
- Software that involves a large number of components can be very complex and requires more testing than a simpler architecture.
- The development project might involve relatively little fresh code, but a large end product. The product might be knitted together from many externally written components or it might be an upgrade of an existing product. The testers will still have to do system testing (the less you trust the external code or the modification process, the more testing is needed). When the external components come from many sources, the test group may have to research, design and execute many different usage scenario tests in order to see how well the components work together to meet actual customer needs.

Challenging Circumstances

Some companies or projects back themselves into excessive testing staff sizes. For example:

- Some test groups don't understand domain testing or combinatorial testing, so they try to test too many values of too many variables in too many combinations.
- Some test groups rely on large numbers of low-skill testers. Manual execution of large sets of fully scripted test cases can be extremely labor-intensive, mind-numbing for testers and test case maintainers, and not very effective as a method of finding defects.
- Test groups that suffer high turnover are constantly in training mode. The staff may never get fully proficient with base technologies, available tools, or the

software under test. Tasks that would be easy for a locally experienced tester might take a newcomer tremendously longer to understand and do.

- Testers may be given inadequate tools. Most testers need at least two computers, access to a configuration or replication lab, a decent bug tracking system, and various test automation tools. To the extent that the software under test runs on platforms for which there are few test tools, the testers have less opportunity to become efficient.
- Testing can be inefficient because the team doesn't use basic control procedures such as smoke tests and configuration management software.
- Software that was not designed for testability will be more difficult and thus more time consuming to test.
- Some corporate metrics projects waste time on the data collection, the data fudging (see Kaner, 2001; Hoffman, 2000), and the gossiping about the dummies in head office who rely on these stupid metrics. We are not suggesting that metrics efforts are necessarily worthless. We are saying that we have seen several such worthless efforts, and they create a lot of distraction.
- Programmers might focus entirely on implementing features. In some companies, testers write installers, do builds, write all the documentation, etc. This is not necessarily a bad thing. Instead it reflects a division of labor that might be wise under the circumstances but that must be factored into the budgets and staffing of both groups.
- Programming teams might send excessively buggy code into testing, perhaps because they are untrained in base technologies, or new to the project, or managed to implement features as quickly as possible, leaving the testing to testers. The worst case of this reflects a conscious decision that they don't have to test the code because they can count on the testers to find everything. Add more testers and the programmers do even less checking of their work. This can become a vicious spiral of increasing testing costs paired with declining quality (Hendrickson, 2001).

One-to-One Ratios of Testers to Programmers

Sometimes groups that describe their work in terms of one-to-one ratios really mean that they use paired teams of programmers and testers. For a given type of feature, a specific tester and a specific programmer work together, perhaps for several years.

There are many advantages to this approach. In particular, the tester is in a position to become very knowledgeable about the types of features this programmer works on and the types of errors this programmer makes. If the programmer and tester get along well, their communication about product risks and bugs will probably get very efficient.

On the other hand, a programmer-tester pair sometimes develops an idiosyncratic model of what things are acceptable to customers or reasonable to report and fix.

FACTORS THAT INFLUENCE STAFFING RATIOS AND LEVELS

Suppose that your boss calls you into a meeting and says, "We have just been assigned Project Whizbang. It will take 20 programmer-years. They started last week and will be done in 6 months. How many testers do you need?"

How do you come up with an answer?

If you use the ratio approach, you might say something like, "In the last 10 projects, we averaged 1.2 testers for every programmer. We were pretty understaffed, though. We had to work lots of overtime, and the product still went out with bugs. So I think we need a ratio of 1.5 testers per programmer, or 30 tester-years. How soon can we start?

A few colleagues of ours, who have significant and successful management experience, have had good experiences with this approach. They don't expect to stop at 30-tester years. They use this number to open the negotiations, to give them a reasonable place to start from.

We agree, *we strongly agree,* that historical data is useful. But we are concerned that historical data, *summarized this way*, can be counterproductive.

A ratio focuses on a *relationship* between two numbers.

These numbers have no direct link to anything but each other. The relationship conveys nothing about the task list, about what the testers will do or what the programmers will do.

At this level of abstraction, it might sound reasonable to say something like, "Last time, your staffing ratio was at 1.2 to 1. I am setting you an objective of 10% greater efficiency. Go forth and become 1.08 to 1."

So, how can we use historical data but still steer the conversation to our needs and responsibilities?

We think we can use the same information that we'd use to justify a ratio, to justify a predicted staffing level. The difference is that when someone asks about the staffing level, we'd talk about the tasks that the testers do (and the ones they won't do if they run out of time) rather than the proportion of testers to programmers.

A Two-Factor Model for Predicting Staffing Requirements

Here's a simple approach for estimating the testing needs for a project. Create a table that shows project size and risk:

		Project Size				
		Very Small	Small	Medium	Large	Very Large
Project Risk	Very low					
	Low					
	Medium					
	High					
	Very high					

As you complete projects, fill in the staff size you had and the staff size that (at the end of the project, with the benefit of hindsight) you think you needed.

Here's an example. The project is maintenance of a product that has been reasonably stable in the field. There are several bug fixes, totaling about 1.5 programmer-months of work. The programmers involved are familiar with the program and have a good track record. However, some of the bug fixes involve device-handling, so extra configuration testing is needed. Therefore, you spend 3 tester-months on the project and at the end of the project, you feel that this was about the right number. You might class this as a small project with low risk—enter 3 months into the Small/Low cell.

Project size is partially determined by the number of programmer-hours, but there are many other factors. For example, adding a hundred new components makes the project large (from the viewpoint of what will have to be tested) even if the programmers took them from a library and spent almost no time on them. Similarly, the project size is increased by an extensive documentation requirement.

The level of risk is affected by such factors as the technical difficulty of the programming task, the skills of the programmers, the expectations of the customers, and the types of harm that errors might cause. The more risk, the more thoroughly you'll have to test, and the more times you'll probably have to retest.

As you gain experience (yours or colleagues'), you'll fill the table with values.

When someone asks you to estimate a new project, use the table. Ask questions to get a sense of the size and level of risk, and then you can say, "This is like these projects, and they took so much staff."

This two-factor model is quite useful.

Rothman (2000) proposed a useful three-factor model, that considered the product (some are harder to test than others), the project and its process (some projects employ better processes than others), and the people and their skills. As with the two-factor model, this folds quite a few considerations into a few variables.

We think it is useful to think explicitly in terms of a longer list of factors, such as the following ten:

- Mission of the testing group.
- Allocation of labor (responsibility for different tasks) between testers and programmers.
- People and their skills.
- Partnerships between testers and other stakeholders.
- Product under test.
- Market expectations
- Project details (e.g. what resources are available when.
- Process (principles and procedures intended to govern the running of the project)
- Methodology (principles and procedures intended to govern the detailed implementation of the product or the development of product artifacts)
- Test infrastructure

We don't think this is the ultimate list. You might do well to generate your own. Our point is that if you are trying to understand your staffing situation, it can help to start by listing several different dimensions to consider. Considering them each in turn, alone or preferably in a brainstorming session with a small group, can lead to a broad and useful set of issues to consider.

Here are additional thoughts about two of these factors, the group mission and the allocation of labor.

Mission of the Testing Group

Different testing groups, even within the same company, have different missions. For example, these are all common missions (although some of them might not be possible):

- Find defects.
- Maximize the number of bugs found.
- Block premature product releases.
- Help managers make ship / no-ship decisions.
- Assess quality.
- Minimize technical support costs.
- Conform to regulations.
- Minimize safety-related lawsuit risk.

- Assess conformance to specification.
- Find safe scenarios for use of the product (find ways to get it to work, in spite of the bugs).
- Verify correctness of the product.
- Assure quality.

A group focused on regulation will spend far more time pre-planning and documenting its work than a group focused on finding the largest number of bugs in the time available. The staffing requirements of the two testing groups will also be quite different.

Allocation of Labour

The most important driver of the ratio of testers to programmers should be the allocation of labor between the groups. If testers take on tasks that go beyond the minimum essentials of black box testing, it will take more time or more testers to finish testing the software.

To estimate how many testers you need to perform the job, you need a clear idea of what those testers are going to do. At a bare minimum, the testers will probably:

- Design tests
- Execute tests
- Report bugs

They will probably also spend time interpreting results, isolating bugs, regressing fixes, and performing other similar tasks.

In some organizations, the testers have a much broader range of responsibilities. For example, testers may also:

- Write requirements
- Participate in inspections and walkthroughs
- Compile the software
- Write installers
- Investigate bugs, analyzing the source code to discover the underlying errors
- Conduct unit tests and other glass box tests
- Configure and maintain programming-related tools, such as the source control system
- Archive the software
- Evaluate the reliability of components that the company is thinking of using in its software
- Provide technical support
- Demonstrate the product at trade shows or internal company meetings

- Train new users (or tech support or training staff) in the use of the product
- Provide risk assessments
- Collect and report statistical data (software metrics) about the project
- Build and maintain internal test-related tools such as the bug tracking system
- Benchmark competing products
- Evaluate the significance of various hardware/software configurations in the marketplace (to inform their choices of configuration tests)
- Conduct usability tests
- Lead or audit efforts to comply with regulatory or industry standards (such as those published by SEI, ISO, IEEE, FDA, etc.)
- Provide a wide range of project management services.

We do not espouse a preferred division of labor in this paper. Any of the tasks above might be appropriately assigned to a test group, depending on its charter. There is nothing wrong with that, as long as the group is appropriately staffed for its tasks.

We think that the best way to estimate your staffing level for a project is task-based. Start by listing the tasks that your staff will do and estimate, task by task, how much work is involved. (If you're not sure how to do this, Kaner, 1996, describes a task-by-task estimation approach.) The total number of staffed tester-hours should be based on this estimate. The ratio of this staff to the programming staff size will emerge as a result, not as a driver of proper staffing.

CLOSING COMMENTS

Ratios out of context are meaningless. Attempting to use industry figures for ratios is at best meaningless and more likely dangerous.

Testers often ask about industry standard ratios in order to use these numbers to justify a staff increase. To justify an increase in staff, we suggest that you argue from your tasks and your backlog of work, not for a given ratio.

Even if you have a backlog, adding testers won't necessarily help clear it (Hendrickson, 2001). Many problems that drive down a test group's productivity cannot be solved by adding testers. For example, poor source control, blocking bugs, missing features, and designs that are inconsistent and undocumented are not going to be solved by doing more testing.

References

Beck, Kent (2000), *Extreme Programming*, Addison Wesley.

Collard, Ross (1999), "Testing & QA Staffing Levels: Internal IS Organizations", Collard & Company.

Hendrickson, Elisabeth (2001), "Better Testing--Worse Quality?", Proceedings of the International Conference on Software Management, San Diego, CA. <http://www.qualitytree.com/feature/btwq.pdf>

Hoffman, Doug (2000) "The Darker Side of Metrics", Proceedings of the 18th Pacific Northwest Software Quality Conference, Portland, OR.

Kaner, Cem (1996) "Negotiating Testing Resources: A Collaborative Approach", Proceedings of the Software Quality Week conference, San Francisco, CA.

Kaner, Cem (2001) "Measurement Issues & Software Testing", QUEST Conference Proceedings, Orlando, FL.

Kaner, Cem, Jack Falk, & Hung Quoc Nguyen (1993; republished 1999) *Testing Computer Software*, John Wiley & Sons.

Rothman, Johanna (2000), "It Depends: Deciding on the Correct Ratio of Developers to Testers," <http://www.testing.com/test-patterns/index.html>.