Estimating Tester to Developer Ratios (or Not)

Kathy Iberle
Hewlett-Packard
kiberle@kiberle.com

Sue Bartlett
Formerly of STEP Technology
Currently at: System Harmony
sustainable@systemharmony.com

Abstract

Test managers often need to make an initial estimate of the number of people that will be required to test a particular product, before the information or the time to do a detailed task breakdown is available. One piece of data that is almost always available is the number of developers that are or will be working on the project in question. Common sense suggests that there is a relationship between the number of testers and the number of developers.

This article presents a model that can be used in describing that relationship. It is a heuristic method for predicting a ratio of testers to developers on future projects. The method uses the model to predict differences from a baseline project. A reader with some projects behind her will be able to come up with a rule-of-thumb model to suit her most common situations, to understand when the model might not give accurate answers and what additional factors might need to be taken into consideration.

Biographies

Kathy Iberle is a senior software process engineer at Hewlett-Packard. Over the past 16 years, she has been involved in software development and testing for products ranging from data management systems for medical test results to inkjet printer drivers and, most recently, e-services. Kathy has worked extensively on researching appropriate development and test methodologies for different situations, training test staff, and developing processes and templates for effective and efficient software testing.

Sue Bartlett is the QA Manager at STEP Technology, a software company that builds technically advanced, high performance, scalable, eBusiness and Enterprise applications. She has over 18 years of experience in software quality assurance and testing, software process improvement and software development. Her domain experience ranges from Computer Aided Design (CAD), color printers, and most recently, web and other business applications. Over half of her career has been at a technical lead or managerial level. Sue has been actively involved in testing education and process.

First published at the Pacific Northwest Software Quality Conference; October 2001; Portland, Oregon.

Acknowledgement

Some of the material in this paper was developed by the participants of the Software Test Managers Roundtable (STMR). The Software Test Managers Roundtable meets twice yearly to discuss test management problems. 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. Facilities and other support for STMR were provided by Software Quality Engineering, which hosts these meetings in conjunction with the STAR conferences. We also thank the members of the email discussion group for their comments in reviewing our paper.

The discussion at STMR3 started with the paper "It Depends: Deciding on the Correct Ratio of Developers to Testers" by Johanna Rothman, http://www.jrothman.com/Papers/ItDepends.html.

Introduction

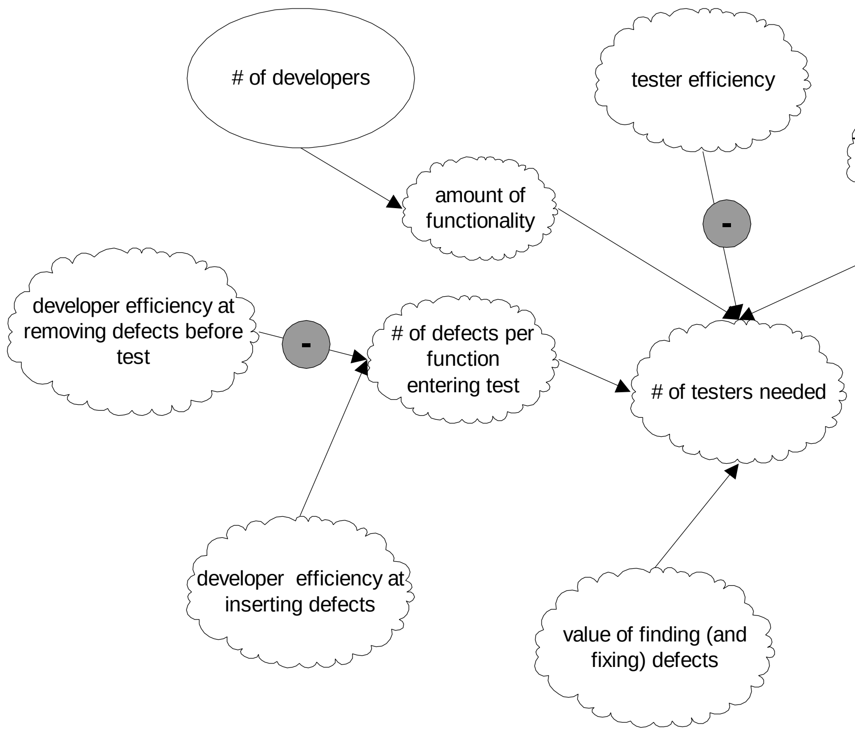
Past tester to developer ratios are often useful for making rough estimates of required test staff based on the number of the developers that will be working on the project in question. Common sense tells us that there must be *some* relationship between the amount of code and the amount of testing needed – and some relationship between the amount of code and the number of developers. This suggests that there is a relationship between the number of developers and the amount of testing needed.

The oft-quoted search for the "right" ratio of testers to developers assumes that these relationships are always linear, although there is no proof of this and considerable evidence to the contrary. We take a different approach, building a systems model that does not assume a linear ratio but instead can be used to predict the direction (if not the magnitude) of different factors on the relationship between the number of testers and the number of developers. This allows the systematic examination of the effect of a number of different factors at the same time.

This heuristic method does require at least one baseline project in a similar domain – a project where the ratio of testers to developers was known, and the relative values of the influences as compared between the baseline project and the new project is known. It isn't necessary to know absolute values if you can say that something is "oh, maybe twice as big" or "about the same".

The Model

The effects that drive the number of testers needed on any particular project are quite complex. The diagram below shows the major effects that contribute to the need for the number of testers.



# of developers
tester efficiency
Tester time spent on other matters
amount of functionality
-
developer efficiency at removing defects before test
-
# of defects per function entering test
# of testers needed
developer efficiency at inserting defects
value of finding (and fixing) defects

Fig. 1: Diagram of Effects on the Number of Testers Needed

This is a diagram of effects[1], in which each node represents a potentially measurable quantity and the arrows represent effects. The clouds are quantities that could be measured, at least approximately, but have not been. An ellipse represents a quantity that has been measured. The arrows are labeled to indicate whether one node affects another positively or negatively. For example, if tester efficiency is better (higher), then the number of testers needed is lower. If the time testers spend on other matters is increased, then the number of testers needed is higher. If the developers are more efficient at inserting defects, then more testers are needed to find those defects. ("Efficiency at inserting defects" sounds bad, but it is sometimes a natural outcome of overall productivity improvements in development, as we shall see.)

But You Promised Me a Ratio…

The "Ideal" Tester-Developer Ratio

The concept of an ideal, unvarying tester-developer ratio is the same as believing that all the effects shown above are either

- Constant and never-changing for all projects in all situations
  or
- Magically cancelling each other out exactly

which would reduce the complexity of the previous effects diagram to Fig. 2. below. This should strain the credulity of any thinking person, including high-level managers.

[1] For a detailed description of effects diagrams, see G.M. Weinberg, *Quality Software Management, Vol. 1: Systems Thinking*. (New York: Dorset House Publishing, 1992).

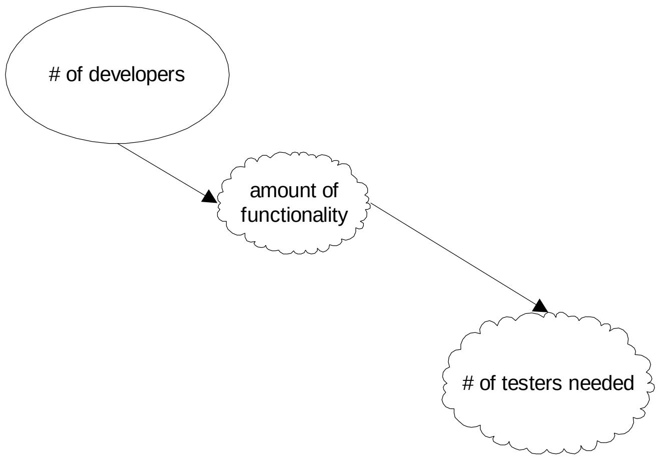


Fig. 2: The ideal tester-developer ratio ignores most effects

Furthermore, a ratio means a linear relationship, so the concept of an ideal ratio implies that the relationship between number of developers and amount of functionality produced is also linear – i.e. that Brooks' "mythical man-month"[2] is not mythical at all.

Comparing Ratios Between Businesses or Companies

It may occur to you to wonder whether the effects in the diagram can be accounted for by having several different ratios, to represent different sets of values of the various effects.

Unfortunately, it is not possible to know whether there is an ideal ratio for any particular business situation. The reported ratios have been measured in a huge variety of ways. The ratio varies dramatically depending on the definitions of "tester", "developer", "headcount", and so forth.[3] Unless you can find out in considerable detail how the other people's ratios were measured, the accuracy of any extrapolation will be seriously questionable.

However, the measurement systems don't appear to vary infinitely. For instance, no one reports ratios of 1 tester to 1000 developers, or 1000 testers to one developer. The combined margin of error in the measurement systems and the actual difference between various projects and businesses doesn't appear to exceed an order of magnitude. Therefore, published ratios can sometimes be used to defend oneself from utterly ridiculous proposals, such as a ratio of 1:30.

So is There Any Good Use for Ratios At All?

The authors believe that there is.

Ratios are useful to estimate the needed number of testers when all of these are true:

- the amount of planned functionality is **not** known or very vague
- the number of developers proposed is considerably more reliable than the functionality description
- data from past projects on number of developers and number of testers is available
- the staffing data is using known units (this almost always means that it is your own data from your own past experience)
- some relative information about the magnitude of the various effects is available.

[2] *The Mythical Man-Month, 1st edition;* Frederick P. Brooks; Addison-Wesley; 1975
[3] see "Managing the Proportion of Testers to (Other) Developers" by Kaner, Hendrickson, and Brock, Quality Week 2001 Proceedings for an in-depth discussion.

If the amount of planned functionality is known in detail, it is more accurate to estimate the number of testers by doing a work breakdown of the proposed testing tasks, and then estimating the number of testers needed to complete those tasks. (Note that both methods involve using your own estimate of tester efficiency in your organization).

Estimation in the Absence of Data

The authors have observed a variety of cases in which the number of developers is more reliably known than the functionality and other information on which a work breakdown would be based.

- **Long-term planning in a company that can project its headcount**. Future projects whose start is a year or more away are often represented simply by the number of developers and a project title, such as "Driver for the next cool printer".
- **An early estimate for Return-On-Investment calculation while a project is still in the idea stage**. The available data is a few paragraphs of project description and a number of developers, often stated as a range. In this case, reporting the estimate as a ratio is often the most useful prediction for the company – because they are trying to get a quick handle on whether this project would be worth doing at all, and your test cost definitely figures in.
- **Initial bid submissions**. The initial proposal stage for contract software in a fast turnaround business can be as short as a two-page estimate or description document. Due to market conditions, turnaround is critical. There are many more proposals being bid on than actually landed. In order to save time, a "solutions architect" often does the initial analysis and proposal without involvement from QA or engineering. The QA staff provides a "rule of thumb" for estimating test time by giving the solutions architect a ratio, plus a couple of items which are added to the estimated test time if needed, such as performance testing for high reliability situations or formal usability testing.

How To Estimate Using Ratios

Given that you have an estimation job that fits the above criteria….

1. Choose a baseline project or projects.
2. Collect data on its ratio of testers to developers.
3. Collect data on the various effects shown in Figure 1.
4. Make an initial estimate of the number of testers, based on the ratio in step #2.
5. For each effect bubble, compare the factors that are substantially different between the upcoming project and the baseline project. (An extensive list of factors is provided in the appendix to this paper.)
6. Combine the results of the various differences.
7. Adjust the number of testers accordingly, using your professional judgment.

If you have more baseline projects, you may wish to repeat the exercise and see if the second estimate is close to the first.

Your ability to estimate accurately will increase over time if you collect enough data to have several interesting points of comparison.

The Various Effects

The STMR group came up with a long list of factors that had been observed to affect the ratio of testers to developers on a particular project. The authors grouped the factors into the effect nodes shown in Figure 1.

Developer Efficiency at Inserting Defects

The more defects that are inserted, the longer it will take to find them all, write reports on them, talk to the developers about the details, and re-run the original tests when the fixes arrive in testing.

While it is tempting to say that defects are entirely due to sloppy development work, in truth there are quite a few factors that are completely outside of the developer's control. Some of these factors are multiplicative. For instance, products that are meant to run on a variety of operating systems or browsers are particularly vulnerable to this multiplying effect.   One line of code can spawn dozens of different failures.

Other factors are actually good for the project overall – for instance, code reuse.   Incorporating large amounts of reused software is a very easy way to increase the number of defects inserted per developer, because the developer now has many invisible and uncountable developers helping to insert defects, as well as additional functionality. It would be more accurate to say that the number of developers has increased, and thus the amount of functionality has increased, but we don't know any ways to count the number of developers for off-the-shelf software.

- **People** – Inexperienced people make more mistakes. Understaffing or rushing the developers will also cause errors. Some programmers have habits that reduce errors, such as routinely initializing all variables, and others have habits that tend to introduce errors, such as leaving error-handling code until the end.
- **Organization** – Anything that hinders communication among the developers or from sponsors will increase the number of defects inserted.
- **Product** - It's easier to make mistakes on some products – poorly documented legacy code, for example. Some products use a line of code in only one environment, whereas others run a line in dozens of environments, thus increasing the developer's efficiency at inserting defects through unintended consequences in one or more of the environments. On the other hand, it's difficult to make interesting mistakes on websites that are essentially online brochures.   Incorporation of others' code, including off-the-shelf software, increases the efficiency of the visible developers at inserting defects (which they themselves did not create).
- **Process** – Use of good software engineering practices such as determining user requirements and modular design will result in fewer defects inserted.

Developer Efficiency at Finding Defects before Entering Test

Developers find almost all of their own errors before the code ever reaches the test group. Even small increases in efficiency at finding defects will reduce the number entering test substantially.

- **People** – As before, experience, talent, education in relevant techniques, and motivation all help.
- **Organization** – The STMR group reported many cases in which the organization actually discouraged developers from finding their own errors by dictating that they must not spend time looking for them, or refusing to provide the appropriate equipment. For instance, if the product will run on multiple operating systems, then multiple computers are usually needed.
- **Product** – I can't think of any product factors that would make it easier to find defects.
- **Process** – Code inspections and unit testing increase the number of defects found by the developers. Frequent milestones also tend to encourage finding defects by encouraging focus on the current work, with more confidence that all the work will get done.

Tester Efficiency at Finding Defects

This factor inversely affects the number of testers – the more efficient the testers, the fewer of them are needed. This is primarily affected by the skill and experience of the testers, and by the tools available to them. Automation can increase tester efficiency, but it can also decrease efficiency if the testers end up enslaved to maintenance of marginally effective automation.

- **People** – Inexperienced testers take longer to set up for the tests, to find the defects, to isolate and characterize the defects, and to write the defect reports. Inexperienced testers often use an inefficient test strategy. They spend more time testing in places where the defects are not lurking, or test the same code multiple times, or order tests in such a way that the setup or execution is less efficient than it could be.
- **Organization** – Anything that hinders communication between the testers themselves or between developers and testers will slow down the finding of defects and communicating them to the developers.
- **Product** – Some types of products take longer to test – those with no user interface where a test harness must be written, for instance, or where a great deal of setup time is required.
- **Process** – Failure to communicate the correct behavior of the product to the testers will really slow down the defect finding (for instance, no specifications at all). Not using appropriate tools for the situation will

slow things down. Insistence on extraneous process (ill-conceived metrics programs, for instance) can also slow things down.

Value of Defects Found – A Look At Risk

The cost of unfound defects varies tremendously across different businesses and types of products. It's reasonable to spend a lot more time and money to find a defect in a missile targeting system than one in the latest experimental Internet game. When the defects are not particularly valuable, it doesn't make much sense to do a lot of testing to find them.

The magnitude of this effect is directly related to the potential impact of errors. If a defect that slips through can cause serious damage or injury, or fixing the defects after release would be prohibitively expensive (e.g. firmware), then the testing effort must be greater and therefore the number of testers goes up. The increased testing effort consists both of looking more carefully and thoroughly for defects on the first time through any area, and increased regression testing.

The value of defects found considers only the impact of defects, whereas the probability of the presence of defects is represented as part of the "developer efficiency at inserting defects". Both are part of risk.

- **Product** – The end use of the product, as one would imagine, has a huge effect on the value of defects found. Medical products, missile control systems, air-traffic control – all require placing a high value on defects found.
- **Customer Expectations** – If the customers are more tolerant of defects in exchange for a lower price or an earlier delivery, it may be acceptable to have less testing and therefore fewer testers. This changes over the course of the adoption of a product, so it's not the same as the end use of the product. Currently, the users of many Internet products will accept lower quality because it's a hot new product, and some businesses take advantage of that by skipping testing[4].

Time spent on other matters

The STMR group reported a wide variety of jobs being performed by testers that were not strictly testing. This ranged from documenting product requirements, to writing installer code, to staffing the technical support help desk. This isn't necessarily a bad arrangement for a business, but it does affect the number of people labeled as "Tester" who will be needed to get a particular project done.

How to Use the Model and Ratios to Estimate Needed Testers

In this section, we will walk through some examples of using the model, using case studies derived from some actual projects. Note that if we had had data about the amount and type of functionality in the proposed project, we would have used a more traditional work-breakdown method. The names and some details have been changed to preserve anonymity.

Collecting Data for a Baseline

The first step involves collecting some data on past projects. The data will involve both hard data and soft data. Hard data includes the number of testers and the number of developers in whatever format you count – hours billed, Full-Time-Equivalents, engineer-months. Soft data can include a wide variety of information that is implicated in the effects discussed above. You may have hard data for some of these, or they may all be subjective.

We have two potential baseline projects: MergoApp and DinkyApp.

Baseline 1 – MergoApp

MergoApp was a fairly straightforward e-commerce website tying into a generic back end SQL database, with third party credit card tie-ins and a custom administration tool.

[4] You can learn more about "early adopters" in Geoffrey Moore's *Crossing the Chasm*, Harper Business, 1999.

Developer efficiency at inserting defects:

- **People:** The technical lead, project manager and developers were all quite experienced and quality conscious.
- **Organization:** The project manager was excellent at enabling good communication between team members. There was good communication between client and project manager.
- **Product:** The product was reasonably well documented and there were no legacy issues.
- **Process:** Good development processes were followed: with reasonable specifications and tools used when needed.

Developer efficiency at removing defects before test:

- **People:** The technical lead and developer had the necessary skills to remove most defects.
- **Organization:** The developer was expected to unit test.
- **Product:** No barriers to defect removal.
- **Process:** Supportive of early defect removal.

Tester efficiency at finding defects:

- **People:** The QA person was an expert tester with extensive programming and testing experience.
- **Organization:** The environment was conducive to good communication, and team members respected each other.
- **Product:** It was reasonably well documented and the user interface was conducive to testing.
- **Process:** A good interactive process was followed between test and development.

Value of defects found:

- **Product:** There were no injury issues to worry about. There was an average level of concern for accuracy, security, and privacy, and medium desire for reliability.
- **Business goals:** There was a low push for time-to-market and little value was placed on low rated bugs.

Time spent on other matters:

- Minimal.

The tester – developer ratio for this project was 1:4. Remember that this ratio makes sense only when compared to other ratios measured using the exact same measures. Since we aren't going to explain how we measured this ratio, it isn't a relevant baseline for your own projects.

Baseline 2 - DinkyApp

Now, we have DinkyApp. DinkyApp is a fairly simple and small e-commerce site. However, an explosion of factors caused quite a variation in the tester-developer ratio.

Developer efficiency at inserting defects:

- **People:** The technical lead was the only programmer on the project. His experience was relatively low compared to the programmers on MergoApp.
- **Organization:** Little attention was given to this tiny project from the project manager.
- **Product**: The product interfaced with unfamiliar third party software and hardware.
- **Process:** There was only a two-page product proposal, no specifications, and little or no up-front design.

Developer efficiency at removing defects before test:

- **People:** Compared to the programmers on MergoApp, the programmer had lower expectations of his work.
- **Organization:** Unit testing and reviews were not encouraged.
- **Product**: It was difficult to do good unit testing due to unfamiliarity with the third party software.
- **Process:** No real feedback on defect release into test.

Tester efficiency at finding defects:

- **People:** The QA person was less experienced, with only a couple of years in testing.
- **Organization:** Neither the developer nor the project manager communicated effectively with the tester. The tester had to pull every bit of information, including when code was ready for testing.
- **Product:** DinkyApp was simpler than MergoApp and thus was easier to test.
- **Process:** There was no specification, written or verbal.

Value of defects found:

- **Product**: There was a medium desire for reliability. Little value was placed on low rated bugs.
- **Business:** nothing relevant.

Time spent on other matters:

- The tester took up the slack in documenting much of what the product does in the test plan and deployment documents.

The tester-developer ratio for this project was 1:1.

Estimating the Number of Testers

Case Study

Now we are asked to make a quick-and-dirty estimate of the number of testers required for DataApp. DataApp is a database implementation to replace an existing spreadsheet solution. We know the estimated number of developers.

1. *Choose a baseline project or projects.*
   Our two previous projects had dramatically different ratios, so we need to choose one to be the baseline in this case. We decide to choose MergoApp – so all subsequent comparisons will be made with MergoApp.
2. *Collect data on the baseline's ratio of testers to developers.*
   That was 1:4.
3. *Collect data on the various effects shown in Figure 1.*
   We can use the information given in the section labeled "Baseline 1" above.
4. *Make an initial estimate of the number of testers, based on the ratio in step #2.*
   The number of developers on this project is 8, so the initial number of testers is 2. This is the preliminary value for the "# of testers" bubble on the effects diagram.
5. *For each effect bubble, compare the factors that are substantially different between the upcoming project and the baseline project.*
   This is where it gets interesting. For each effect cloud, compare the current project with the baseline project. If the comparison suggests that the effect will be more powerful, put a "+" in the bubble. If it suggests that the effect will be less powerful, put a "-". You will have to use your own judgment as to whether factors should be counted equally or not.

   Developer efficiency at inserting defects:

   - **People:** The developer is quality conscious with considerable experience. SAME as MergoApp.
   - **Organization:** Adequate coverage of project management duties and reasonable communication between client and team is expected, based on past performance of the assigned staff. SAME as MergoApp.
   - **Product**: The staff acts like they understand the project. SAME as MergoApp.
   - **Process**: There are reasonable specifications and good process will be followed. SAME as MergoApp.

   It appears that the number of defects inserted per developer will probably be pretty similar for MergoApp and DataApp.

   Developer efficiency at removing defects before test:

   - **People:** The developer is skilled at removing defects. SAME as MergoApp.
   - **Organization:** Some level of unit testing is expected. SAME as MergoApp.
   - **Product**: No barriers to defect removal. SAME as MergoApp.
   - **Process**: Again, process expectations appear much the SAME as MergoApp.

Tester efficiency at finding defects:

- **People:** The QA person is less experienced than MergoApp's tester, with only a few years in testing. (Same tester as in DinkyApp.) LOWER than MergoApp – put a "-" in the "Defects found per tester" cloud.
- **Organization:** We anticipate good communication and respect between team members. SAME as MergoApp.
- **Product:** A test harness and some SQL programming will be required in order to test. Also a "-" – this is going to take extra time.
- **Process:** The test process is quite similar to that of MergoApp. SAME.

The "defects found per tester" cloud gets two "-", suggesting that we will need extra testers.

Value of defects found:

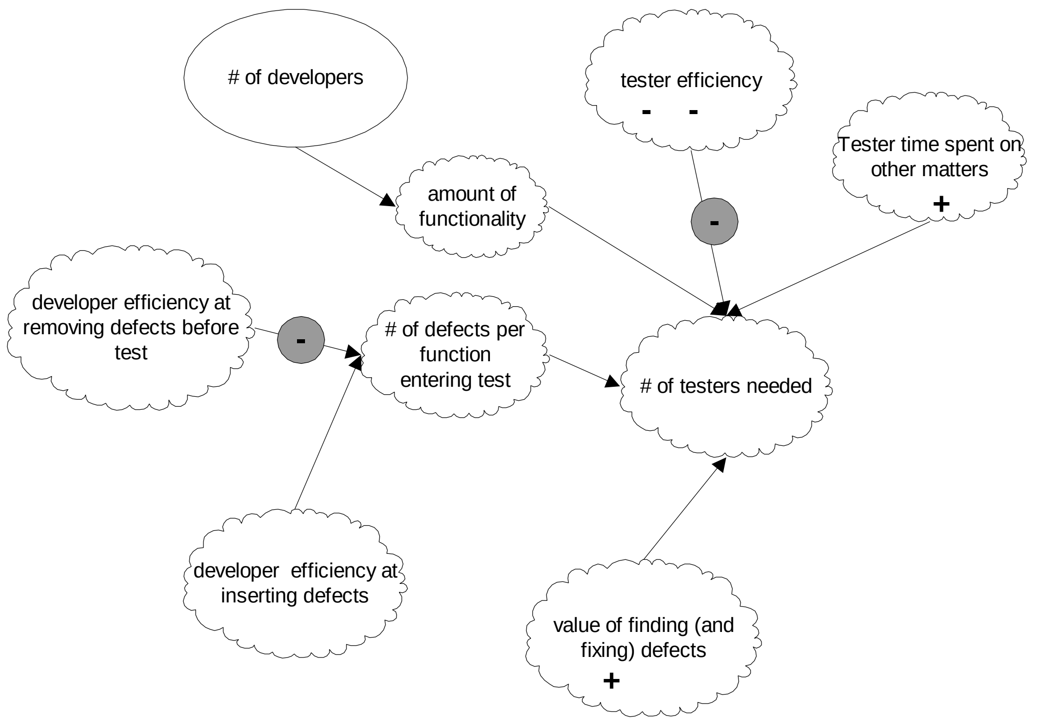
- **Product**: There is a high desire for reliability, some value on low rated bugs. A little HIGHER than MergoApp.
- **Business**: Nothing relevant.

The value placed on defects found appears to be a little higher for DataApp than for MergoApp, so we put a "+" in the cloud for "value of defects found".

Time spent on other matters:

- Tester is to participate in installation at client site. HIGHER than MergoApp.

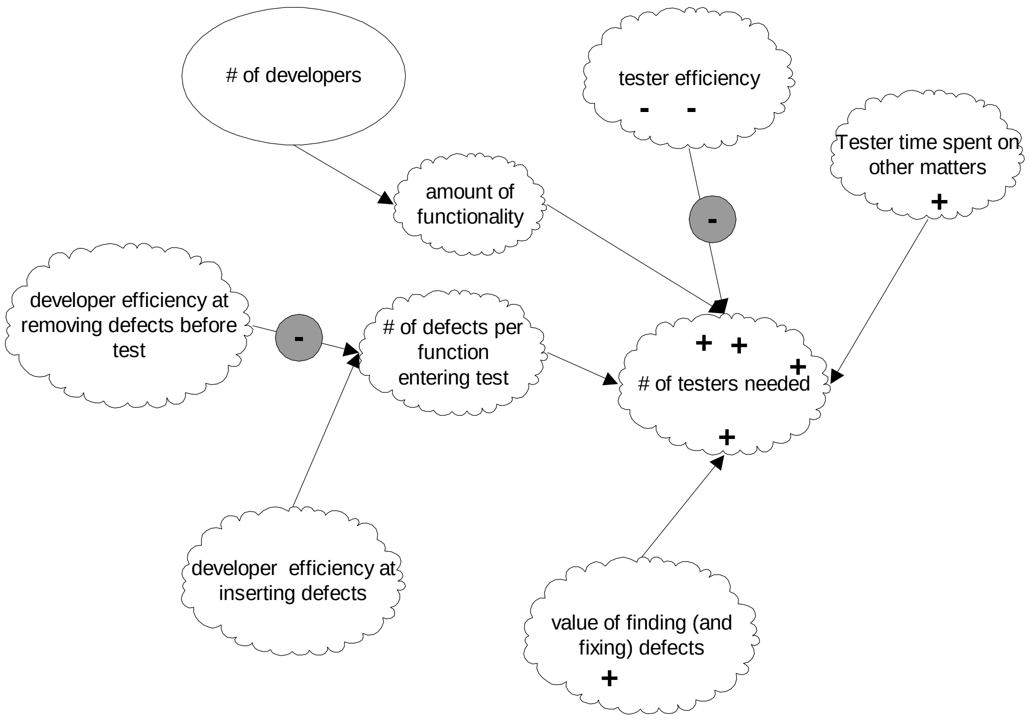
The "Time spent on other matters" cloud ends up with a "+", since this effect will be stronger than it was on MergoApp.



# of developers
tester efficiency
- -
Tester time spent on other matters
+
amount of functionality
-
developer efficiency at removing defects before test
-
# of defects per function entering test
# of testers needed
developer efficiency at inserting defects
value of finding (and fixing) defects
+

Fig. 3: A worksheet for comparing DataApp with MergoApp.

6. *Combine the results of the various differences*. For each bubble, follow the arrow to the next bubble and put the same number of "+" and "-" in it. Don't forget to change sign if the arrow represents an inverse relationship!
When you are all finished, the "# of testers needed" cloud should have a bunch of "+" signs in it, and a bunch of "-" signs, as shown in Figure 4.

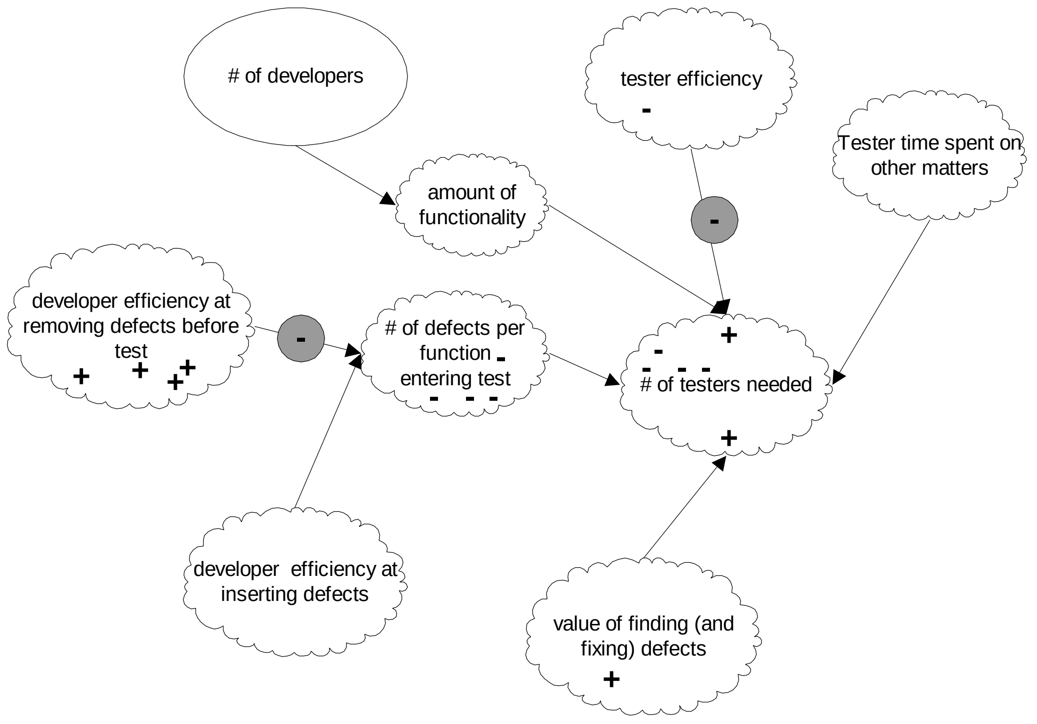


# of developers
tester efficiency
- -
Tester time spent on other matters
+
amount of functionality
-
developer efficiency at removing defects before test
-
# of defects per function entering test
+ +
+
# of testers needed
+
developer efficiency at inserting defects
value of finding (and fixing) defects
+

Fig. 4: The completed worksheet for comparing DataApp with MergoApp.

7. *Adjust the number of testers accordingly, using your professional judgment.*
The number of "+" and "-" in the "# of testers needed" cloud suggests the necessary adjustments to the number of testers estimated in step #4. You will have to decide for yourself whether the effects are of comparable weights or very different weights. In this case, the number of testers clearly needs to be higher than 2.

If you have more than one baseline project, it may be useful to run through this exercise once for each project. We repeated the exercise using DinkyApp as the baseline project. The comparison worksheet is shown in Figure 5.



# of developers
tester efficiency
-
Tester time spent on other matters
amount of functionality
-
developer efficiency at removing defects before test
+ + + +
-
# of defects per function entering test
- - - -
+
- - - -
# of testers needed
+
developer efficiency at inserting defects
value of finding (and fixing) defects
+

Fig. 5: DataApp compared with DinkyApp

The ratio of developers to testers on DinkyApp was 1:1, so the starting value for number of testers would be 8. The effects diagram suggests that the actual number of testers needed is probably less than 8.

This brackets the estimate for DataApp with a floor of 2 and a ceiling of 8. The margin appears considerable in both cases, so neither 3 nor 7 seems like a reasonable value. That leaves 4, 5, and 6 as possible numbers of testers (assuming that they have to be whole numbers, which usually isn't the case).

The margin of error in the estimate appears to be around 25% to 50%. If you cannot afford this margin of error, then we don't recommend this method.

Incidentally, the actual tester-developer ratio for DataApp was about 4:8.

Summary of Case Studies

As we used this model with this particular case study data, one thing that stood out was how much the experience of the people involved affected the factors and thus the ratio. A more experienced developer or tester is much more likely to follow good process, to be quality conscious, to insist on some documentation or write it herself, and to communicate in a positive way with her entire team.

Other Uses for the Model – Explanations to Management

We found the model and its effects diagram to be useful in explaining a number of things to management. An effective presentation in slide format can be done by starting with the entire effects diagram, and then circling or marking the effects that have changed or that you propose to change.

How Can We Reduce the Number of Testers?

Test managers are often asked to reduce the ratio of testers to developers. This model can be used to demonstrate that development activities and organizational activities play a large role in the ratio, which is often overlooked by test managers. For instance, a large organization with many teams and several layers of integration may need more testers proportionally, because the developers have more opportunity to make mistakes based on miscommunication with each other.

The technology used in the product itself can also affect the number of testers in ways that management may find hard to understand without a pictorial representation. The replacement of in-house code with an off-the-shelf software application will reduce the number of developers, but the number of testers will not go down proportionally because the off-the-shelf software is not itself perfect. The effects diagram can be used to demonstrate that there are, in effect, invisible and uncountable developers who have inserted defects into your system via the off-the-shelf software.

More Testers, Worse Code?

We found the model useful in analyzing or explaining some situations that defied the ordinary logic of the tester-developer ratio.

For instance, some STMR members observed instances in which increasing the number of testers (thus raising the ratio) resulted in more defects inserted and yet more need for testers. In one instance[5], the increased number of testers caused the developers to start skipping parts of unit test and generally take less responsibility for the quality of their code, in the belief that the increased number of testers would pick up the errors. This reduced the developer efficiency at removing defects before test. Using the model to analyze this situation suggests a positive feedback loop, which could be dangerous.

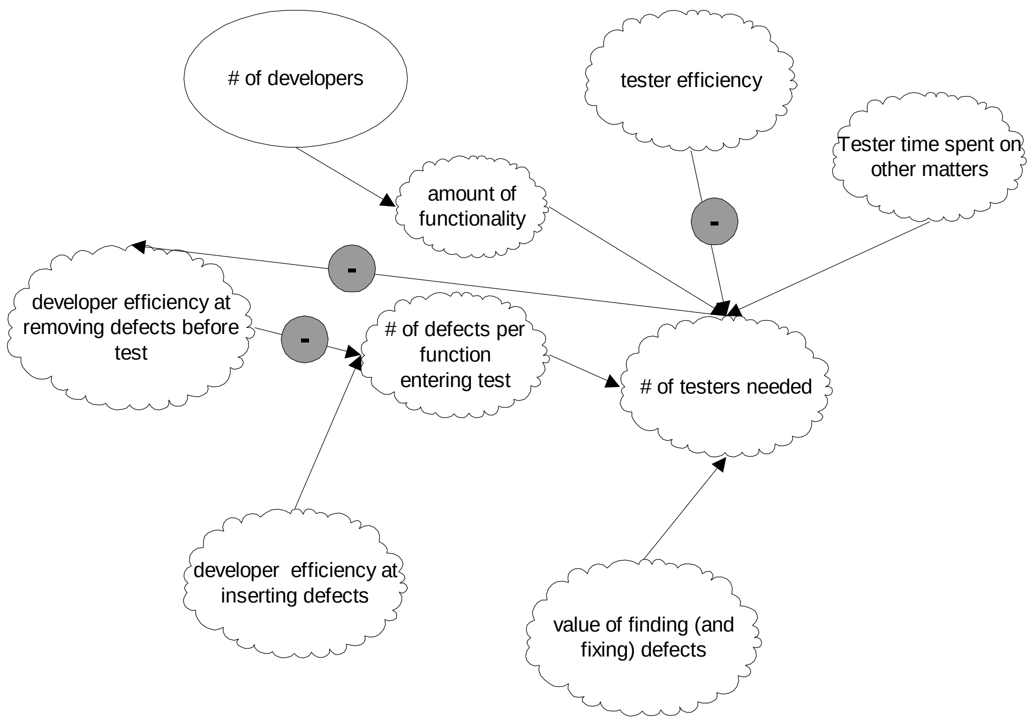

# of developers
tester efficiency
Tester time spent on other matters
amount of functionality
-
-
developer efficiency at removing defects before test
-
# of defects per function entering test
# of testers needed
developer efficiency at inserting defects
value of finding (and fixing) defects

Fig. 6: Effects diagram of a feedback loop between number of testers and developer efficiency at removing defects before test

[5] "Better Testing, Worse Quality?" by Elisabeth Hendrickson, presented at SM/ASM 2001, San Diego, CA. Available on her website at QualityTree.com.

Fewer Testers, Better Code?!

Another interesting situation was the surprisingly low ratio of testers to developers on a few good, solid products. The STMR group collectively reported a handful of instances in which the tester to developer ratio was 1:10 or less and the product was successfully released with appropriate quality levels.

This counter-intuitive result appears to result from the effects of the development staff placing a high value on finding and fixing nearly all defects, and the sensible observation that they wouldn't have to fix what they didn't insert in the first place. The developers adopted careful practices (usually by choice) that resulted in either never inserting the defects in the first place or finding the defects before the product entered test.

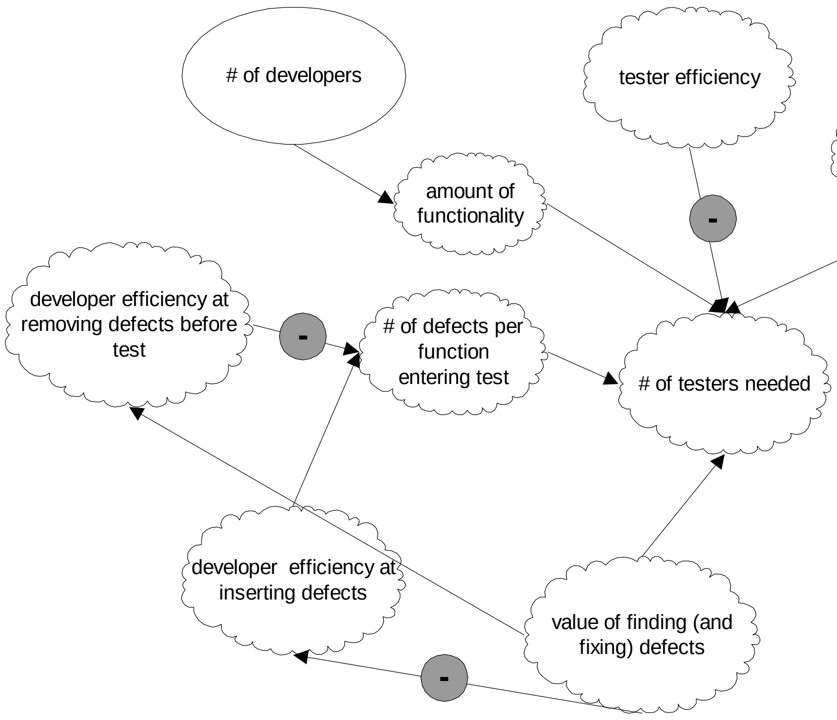
The high value placed on the defects may be a result of an unusually high potential impact of errors (such as in life-critical software), but it also occurred in other circumstances where possibly the developers themselves were deciding to place a high value on finding the defects.

There were some common factors observed in these situations:

- The developers fairly carefully followed the usual recommended practices – requirements, design, coding, and unit testing.
- In some instances, the product was intentionally constrained in order to reduce errors – for instance, was marketed to run on only one operating system even though theoretically capable of handling a family of operating systems. Products that run on a multiplicity of platforms allow developers to insert many more defects per hour of work.
- The developers were unequivocally responsible for the quality of the release. The testers were hired by the developers, and the developers themselves were expected to understand testing – in some cases, were the primary test planners.

To a testing manager, this scenario may be counter-intuitive. The lowest tester-developer ratios were achieved on the products with highest expectations for reliability, and the test group was not only unimpressive in terms of skills, but in at least one case no one was identified as a tester! Obviously some of the developers were indeed performing testing, but estimates of developer time spent on testing still suggested that the tester-developer ratio was far lower than usual. The key factor here may be the perceived value of defects found, which influenced the processes followed in development (see Figure 7).

It's not clear that this ratio actually minimized the number of people needed per function point delivered (and therefore cost), although current software engineering wisdom would certainly say it does.



# of developers
tester efficiency
Tester time spent on other matters
amount of functionality
-
developer efficiency at removing defects before test
-
# of defects per function entering test
# of testers needed
developer efficiency at inserting defects
value of finding (and fixing) defects
-

Fig. 7: Effects diagram of the results of developers placing a high value on defect-free software

Summary

The ratio of testers to developers on past projects in a well-known domain can be used in conjunction with an analysis of effects on the relative number of testers vs. developers to roughly predict the number of testers needed for future projects. This method is most useful when information on the functionality and features of the proposed project is not available, or when a quick estimate is desired and a large margin of error is acceptable.

The effects diagram can also be used to graphically demonstrate the probable effects of changes in people, organization, product, or process factors when making presentations to management.

Appendix – The List of Potential Factors

This list is derived from work done at STMR 3. You can use these lists to consider the differences that might affect the tester-developer ratio on a given project as compared to a baseline project.

Developer Efficiency at Removing Defects before Test

These factors *increase* the number of defects that get by the developers.

People factors:

- Inexperience with unit testing or review techniques.
- Over-confidence.

Organizational factors:

- Upper management belief that testers can discover all types of errors more efficiently than the original developers and therefore developers should not unit test. This is also known as "development already did enough work."

Product Factors:

- Code or design that is difficult to understand and thus hard to test.

Process factors:

- Failing to unit test.
- Failing to use reviews and inspections appropriately.

Developer Efficiency at Inserting Defects

These factors *increase* the number of defects inserted by the developers, which *increases* the tester to developer ratio.

People factors:

- Inexperience with software engineering principles, leading to poor design.
- Inexperience with the technologies or business domain of a particular project.
- Belief that working fast is better for the schedule than working carefully.
- Burnout.

Organizational factors:

- Clear communication between developers is hindered by:
    - Geographic separation between developers.
    - Failure to pass on requirements for product to developers.
    - Layers of management or organization between different developers, including outsourcing part of the development.
- Arguing or hostility between teams.
- A culture that does not hold developers responsible for quality of their product.
- Disfranchised engineers with imposed schedules.
- No accepted measure of success or "doneness" except "code complete".
- Death spiral: too many testers can cause developers to get sloppy.
- Little interest in finding patterns of mistakes (defect prevention).
- Many teams, resulting in many layers of integration, which increases the integration problems

Product factors:

- Complexity of product and range of technologies included.
- Incorporation of reused software, third-party custom-written software, or off-the-shelf software.
- Size. There tends to be a nonlinear relationship between lines of code and number of defects.

- Poorly designed code inherited from others – highly coupled, poorly modularized.
- Legacy code, which is not fully understood by the programmers.
- A single design fault can be multiplied into numerous failures when the product:
    - is translated or localized,
    - will run on a wide variety of platforms, or
    - will interoperate with a wide variety of other products.

Process factors:

- Use of powerful programming tools which allow each programmer to generate more code
- Insufficient configuration and build tools or process.
- Poor planning, resulting in developers in a hurry.

Defects Found per Tester

Factors that *decrease* the number of defects found per tester, and thus *increase* the ratio of testers to developers.

People factors:

- Inexperienced or poorly trained testers.
- Poor attitude or morale.
- Testers do not have the specific knowledge for this particular job. For instance: exploratory testing, familiarity with user domain, performance testing, combinatorial methods, or knowledge of a range of appropriate patterns for test design.
- Developers do not know how to write code to be easily testable.

Organizational factors:

- Barriers to communication with development staff.
- Inappropriate use of outsourcing of test, as in duplication of work between outsourcing and in-house test, not anticipating overhead in coordinating testing, using insufficiently skilled outsource test house.
- Culture of mistrust – time is spent gathering data to protect oneself later.
- Disagreement on role of testing – time is spent on arguments regarding who is responsible for what.
- Expectation of superfluous status reports, metrics, and so forth that do not help make testing more efficient nor assist the programmers.
- Insufficient time allowed for test group to develop the appropriate test tooling and collective knowledge and processes to be optimally efficient.
- Teams are constantly reshuffled, leading to loss of knowledge and barriers to communication.
- Lack of adequate equipment, work space, or ergonomics for the test team.
- Testers handle multiple projects simultaneously, leading to loss of time on task switching. (Some sources estimate a 5-10% loss for each additional project).
- Automation group is disconnected from the needs of the organization and burns resources without corresponding return on investment.

Process factors:

- Code is submitted to test before it works at all, wasting test time on installing broken code.
- Regulatory environment requires a lot of extra documentation. (e.g. ISO 9001, FDA approval)
- Test plan and tests will be sold along with the product, and therefore must look nice.
- Testing will be taken over by totally unrelated group, requiring much greater communication time.
- Test group has not acquired, built, or learned to use appropriate tools, such as test automation.
- Regression testing is repeated unnecessarily just to clock hours (often due to misuse of metrics)
- Test group does not store their tests and reuse them when possible.
- Testing does not start until code complete, which cuts amount of time to find defects, not efficiency of tester.
- Production of useless documentation of any kind, including metrics that don't guide testing effectively.
- Lack of metrics that would guide testing more effectively.
- Lack of test planning to use time most effectively.
- Defect tracking process or tools not understood by development staff.

Product factors:

- Needs a lot of automation to execute any test. (e.g. no user interface, like firmware.)
- No test interfaces built in, or insufficient or unclear error logging.

Value of Defects Found

These factors *increase* the value of defects found, which *increases* the ratio of testers to developers.

Product:

- Expensive to fix problems after release. (e.g. firmware.)
- Regulatory requirements are stringent.
- Will be changed frequently, thus maintainability is important.
- Has high need for reliability or safety.
- Will have long support life.

Customer expectations:

- Has competition giving users the choice to use the product or not.
- Is expected to be a professional-looking product, as opposed to having only internal users.
- Is for actual use, rather than being a demonstration or sample.
- Has customers who are unusually particular - adopters, resistant, cultural differences.
- Customers would rather have a good product later than a buggy product now.
- Competing products are less buggy than ours.

Tester Time Spent on Other Matters:

If testers spend time on these tasks, the ratio of testers to developers *increases.*

- Inspecting requirements
- Inspecting code
- Writing requirements
- Preparing material for customer support
- Participating in rollout of product
- Writing product documentation
- Taking support calls
- Handling configuration management and builds
- Writing installers
- Making site visits
- Running clinical testing
- Supervising beta programs
- Testifying or preparing for court cases involving your products
- Researching customer use models