Working Effectively With Developers

Brian Marick
Testing Foundations
marick@testing.com

Developers and testers often do not work effectively together. Either the developers can change, we can change, or both. I've never had much luck directly changing other people's behavior. I *have* had luck changing my own behavior to indirectly elicit improvements in the way developers act. And those changes feed back to me, producing a virtuous cycle in our relationship. This paper is about my approach to working effectively with developers.

I expect that you, my reader, are a tester and spend most of your time looking for bugs, talking to developers, and writing bug reports.

This paper describes three things I've learned to do better over the years:

1. Define my role in a way that makes a developer want me around. I help the developer get rid of bugs before anyone else sees them, which reduces total project cost.

2. Explain myself and my job such that a developer gives me the benefit of the doubt. I reduce the opportunity for snap judgements about me, and I take an unusual approach to convincing the developer that bug reports should not be seen as threats.

3. Write bug reports in a way that allows for little misinterpretation.

The three main sections are largely independent. The techniques do not depend on each other.

1. Picking an Effective Role

In this section, I first describe my preferred role and its implications for day-to-day work. I then describe the problems this role solves and, importantly, what new problems it might create.

I assume you've been told to test a particular developer's work, perhaps a new feature added to the product. You are probably handling more than one developer at once, but I cover the complications that causes only later in the section. I assume you'll start working during coding: after the developer writes the first line of code but before she's finished (except for fixing whatever bugs you and others find). If you're involved earlier, that's great, but I don't assume it. If you start after the code's done, this section will be of little use, except to help you argue that you should start earlier next time. It also doesn't apply if you're doing configuration testing, load testing, or other types of testing where you're not looking for bugs in a particular developer's work.

Who does the testing team serve? One common answer is "the customer". In this model, the testers serve to protect the customers from the developers. The customer is the helpless maiden tied to the train track with a runaway product bearing down on her. Only the testers are there to save her by stopping the train (from shipping - at least until the bugs are fixed).

I have a number of problems with this model, not the least of which is that it rarely works. Too often, the train runs over the maiden, leaving her intended saviors frustrated, demoralized, cynical, and ineffective [Marick97a]. For purposes of this paper, though, a specific problem is more relevant: casting people into the role of villains is not the best possible prelude to asking them for help. Make no mistake: you'll spend a lot of time seeking help from developers. You'll need explanations of the program, which is asking for a developer's most precious gift: time. You'll file bug reports and you'll trust the developer to handle them sensibly, not seeking every excuse to declare a bug a feature, to insist on more information, or otherwise waste your most precious resource: time.

Perhaps I exaggerate. Perhaps no developer *really* believes you think that she's a villain. More likely, she believes something worse: that you think she's too error-prone to trust. Most developers would rather be considered a villain than an incompetent. Whatever the nuances of reaction, the fact remains that this model gives developers and testers conflicting goals. The developers want to ship; the testers try to stop them. Cooperation from the developers will be at best dutiful, at worst resentful: never reaching the peak possible when testing is integrated into the team.

So: if not the customer, who does the testing team serve?

I claim it should be that person who must take all things into account and decide whether the product should be shipped. "All things" includes what competitors are doing, the risk that the programmers are so tired that further effort is likely to do more harm than good, the costs of breaking commitments, and what benefits might be lost to both the customers and the company were the product delayed. (See [Bach97] for a longer discussion.) I'll call that person the "project manager". The testing team's job is to provide an accurate assessment of one of the factors the project manager tracks - the product's bugginess, specifically from the perspective of what customers are likely to see and care about. (See [Marick98a] for an elaboration of this argument. See [Marick97b] for how the testing team can report bug data, as well as a few other types it makes sense for them to track.)

Let me emphasize that, in my scheme, testers still have the same core work product: bug reports. It's the use of those bug reports that differs. Rather than being used as evidence against the product, they're turned into information that allows the project manager to make better decisions.

Now, it's far from clear that this new role helps cooperation with the developer. Consider it from her point of view: the tester is always there, hovering around, intent for any sign of a problem, ready to pounce on it and carry it off to ~~daddy~~ the project manager. Nobody likes a tattletale.

How to counter this? One solution is a team structure called *egoless programming* [Weinberg71], in which programmers do not think of code as an extension of themselves. Their ego is not involved, so their own errors are less threatening. However, I can't allow this solution, because it would mean developers have to change their behavior, not me. The developers I work with are seldom egoless, and I have no choice but to live within that constraint.

My solution hinges on the distinction between *private bugs* and *public bugs*. A private bug is one that a programmer creates but fixes before placing the code into the public source base, where it can potentially harm her fellow programmers and, much later, customers. Once a bug is out there, protection of teammates dictates that it should be publicized, if known. (Note the important semantic shift: it's not publicized to punish the creator, but to protect her friends.)

As a tester, my message to the developer is that I'm here to help her keep her private bugs from becoming public. Presto! I am now an ally, someone devoted to making her look better. My job is to serve as her safety net, to concentrate especially on those bugs that she has trouble preventing or finding herself. I'm a tool, a way to expand the developer's mental reach. (Let's not forget that, even when we work closely with developers, they usually find more of their own bugs than we ever do.)

Let me be clear about what's happening here. Suppose my developer is ready with a chunk of new code and three bug fixes. I will begin testing immediately. The only thing that would cause me to defer testing would be if I were already doing the same thing for another developer.

When I find problems, I email bug reports to her. I normally do <u>not</u> put those bug reports into the public bug tracking database. I don't share them with other testers, other developers, or managers. They're private. I record and track them in my own email folders unless the developer asks that they be tracked publicly.

The developer might not fix all the bugs before putting the code in the public source base. I might disagree with that decision, but I accept that the developer is better placed to make it. (This is precisely parallel to my opinion about the relation of the whole testing team to the project manager.) If she doesn't fix a bug, its bug report moves from my email folder to the bug tracking system. Since the bug is now public, the public (that is, the project manager and other developers) needs to know about it.

When I'm not looking for private bugs, I look for public bugs. That is, I finish testing code already in the public source base. (Perhaps it's there because I wasn't assigned to the task early enough. Or perhaps the programmer needed to make something partly working available for demo purposes or to unblock other programmers.) Bugs I find there are public bugs and reported through the public bug tracking system.

That's an oversimplified description, in order to make the point that I always err on the side of helping the programmer find private bugs. In reality, things are more complicated. Suppose that the code just finished is not very important, and that bugs in the code already made public could seriously hamper other programmers. Then I might concentrate

on public bugs. I'd almost certainly have the programmer's support in that, for reasons explained in the next major section. And, sometime before the end of the project, I need to make sure that all the code gets tested appropriately. Testing a steady stream of changes mustn't prevent that.

What problems are now smaller?

Why am I going to all this trouble? It sure sounds like I'm out to make friends. Friends are nice, but companies don't pay me to improve my social life. They pay me to use collegiality as a means to an end, a way to reduce problems. What problems?

In general, friction. I am much taken with John Daly's Rule[1], which has you always ask this question of any activity: "What bugs aren't I finding while I'm doing that?" Friction slows down movement. Typical developer/tester friction wastes time that could be better spent finding bugs.

But there are other, specific problems that this approach helps with.

Bugs found uneconomically late. A bug found immediately is almost always cheaper and faster to fix than the same bug found after the developer's had time to forget what she had in mind. Avoiding the overhead of a formal bug tracking database can also be a savings.

Developers who don't test. A common complaint about developers is that they "throw the latest build over the wall" without trying it at all. It breaks immediately, is therefore untestable, and so it gets thrown back. Negative accomplishment: wasted time.

A similar problem is a developer who claims a bug has been fixed, but the simplest test reveals that the bug is still there. I once watched a programmer fix the same six-line routine <u>four</u> <u>times</u>, introducing an obvious bug each time. I don't see how he could have possibly tried his fixes at all.

It might seem I'm making this problem even worse by shackling the tester to the programmer, having you test each change as soon as possible. Am I not giving her permission to test even less? No, because I've changed the perceived economics of the situation. Before, the programmer lost nothing if she saved ten minutes of her time by costing you two hours. Now she does: those two hours of wasted time mean that more private bugs will become public. She has more of an incentive to run a quick automated build verification test ("smoke test") that you might have built for her, and to try the new feature or bugfix before handing it to you. She has a reason to talk to you about what kind of division of labor makes sense.

"Please debug this for me." Developers often annoy testers by asking them to do a large part of the debugging. They may ask for a very specific test case that has a minimum number of steps. They may ask for extensive documentation of variant tests that they could try more efficiently themselves.

---

[1] As reported to me by Noel Nyman. Quoted by permission of John Daly himself.

Again, this problem is reduced because wasting your time costs the programmer something. Further, because the feedback loop is tighter - because you're more likely able to just show the developer what's wrong, as opposed to filing a bug report that has to be understandable in three weeks when the developer can look at it - the cost to you of debugging is reduced.

Not knowing what's new. Testers often complain that they don't know what's in new builds of the product, so they don't know what to test. Sometimes this omission is thoughtless: it's just too much bother to keep everything up to date and everyone informed. Sometimes it's mischievous: the developer *really* wanted to add that feature, couldn't get permission, but had heard Grace Murray Hopper give a speech that included her slogan "it's easier to get forgiveness than permission", so added it.[2]

It's harder to be thoughtless to someone you work with closely, especially someone who's helping you keep problems in that new feature private. It's also slightly harder to hide mischief from that person.

"Stop asking for all those stupid documents." Testers need to know what the product's supposed to do before they can tell if it does it. Programmers aren't thrilled about providing that documentation.

When working closely, more information can flow informally. That's perhaps not ideal, since any tester who comes after you will still be in the dark, but it may be the reasonable alternative to nothing[3]. As a tester, I've also made myself useful by writing the documentation myself (turning the informal into formal). (This is similar to Coplien's Mercenary Analyst pattern [Coplien95].)

"Please do all these things that are not your job." Testing teams seem to collect unpleasant or difficult jobs that often get in the way of actually doing testing. That's especially true when testing teams are labeled Quality Assurance, a term vague enough that anyone who doesn't want to maintain the configuration management system or create system builds (that is, anyone sane[4]) can argue that QA should do it instead. And individual testers helping developers can similarly end up sidetracked, for example, into writing some support code.

As you might guess from my experience as a technical writer, I'm not averse to testers helping out the team. But, once bugs become a service to the developer, John Daly's Rule becomes more compelling.

Untestable code. Testers are often presented with code that is difficult to exercise intentionally in ways that users might do accidentally. Or they might have trouble knowing they've done so. Or it might be hard to tell what the code has actually done.

---

[2] Rear Admiral Grace Murray Hopper, deceased, was there at the beginning of the computer era. She is credited with discovering the first computer bug, a moth trapped in a relay of the Mark I computer. (Some question the claim.)

[3] The customers will also be in the dark, but the consensus these days seems to be that they learn what a feature is supposed to do by trying it, not by reading documentation with the detail testers want.

[4] Okay, I exaggerate for effect. I was a "build czar" for over a year, and I was sane. At least at the beginning.

When developers value you and your bugs, they're more apt to add testability support code. Often a little goes a long way ([Marick95], chapter 13).

Lack of developer introspection about bugs. When I work closely with a developer, they learn my approach to detecting bugs. They become more able to prevent the bugs I would otherwise detect. In this way, I may "move upstream", helping the developer with design or planning rather than only doing after-the-fact testing. I don't push this on developers; I wait for them to ask.

What problems might I create?

Every solution brings with it a potential for new problems.

You have to start earlier

If you begin working with the developer after all the code has been made public and all that's left are bug fixes, you won't get much benefit.

Greater sensitivity to tester/programmer ratio

Consider the differences between this approach and "over the wall" testing.

- It is more interrupt-driven. To maintain a good relationship, you need to respond quickly when a programmer needs help. In conventional testing, you have more freedom to queue testing tasks.

- It is more dependent on structural knowledge. In both types of testing, you <u>must</u> be able to view a feature as its user might. In conventional testing, some understanding of the structure or architecture of the feature or whole product is useful; it enhances your user-centered tests and sometimes allows you to omit unneeded tests. But, when working closely with a developer, such understanding is more than useful: it's essential for effective communication and - just as important - for giving the developer the feeling of effective communication.

  I want to be careful here. I do not believe that the degree of structural knowledge you need requires you to be a programmer. I have seen nonprogrammers do this perfectly well. I've known one good programmer who explicitly values nonprogrammers precisely because explaining structure to them requires him to clarify his thoughts.

What are the implications of these differences?

Suppose the tester/programmer ratio is 1 to 10, and you're trying to work closely with each of them.

- Requests for help will come at relatively unpredictable intervals. You'll have to queue some people. But that reduces your value to them, so they're less likely to ask for help next time.

- You won't be able to keep up with the structure of each programmer's subsystem. You'll need explanations of things the programmer may think you should have already

learned. You may need to have something explained again, which the programmer may find infuriating. (From her perspective, she just explained it to you. From yours, seven other tasks intervened, driving the explanation out of your mind.) Or, worse, you may remember something that's no longer true, leading to misdirected tests or spurious bug reports.

In such a situation, testing will devolve into "over the wall testing", because that just works better under such constraints. To avoid bad feelings, you might as well start it out that way.

Suppose the ratio is more like 1 to 3. That's more tractable, but it's still likely that you'll disappoint the developer by not being able to test as much as she'd like (meaning too many private bugs are missed). You need to handle this carefully. Be sure not to over-promise, to raise unmet expectations. I've done that. It's bad, worse than not trying to work closely. Be especially careful to identify risky areas of the developer's code, or risky types of bugs, and to explicitly agree that those are what you'll concentrate on.

Because I'm sensitive to how easily developers can pin their hopes on testers, I would prefer to be assigned to work closely with one developer and handle the other two in a more conventional way.

You don't learn from secrets

Suppose testers and developers do a wonderful job, and few bugs are ever found in public source. How does anyone learn from those paltry few public bug reports? Noel Nyman, who reviewed a draft of this paper, wrote: "When I started in each [new] test group, a primary source of information was the bug history. It's an invaluable insight into the product and what regression tests might be needed. No one describing the products has ever given me 10% of the information I found in the old bugs." Other reviewers pointed out that, while developers might learn to avoid their own characteristic bugs, they won't benefit from seeing what kind of bugs other developers make. Process improvement groups would have a similar problem.

I have no compelling answer to this problem, other than that it's a price I'm willing to pay. Public bugs are still found and logged. In some sense, they're the best ones to learn from, since they escaped both the developer and tester, but it would be better also to know about private bugs. A company with a successful process improvement effort (I've never worked in one) might be more egoless, a place where developers are more willing to log private bugs for the public good. (Two reviewers, Johanna Rothman and Jeanine Brown, have seen developers do this.) The arguments I make in the next major section might help toward that end. I've never been fortunate enough to work at a company that does inspections well, but such companies might already be used to sharing private bugs, at least among developers.

Build control

I've applied this approach to relatively monolithic and self-contained products: I get an executable from the developer and run it on my machine. If the product is broken up into

many pieces (shared libraries, configuration files, etc.) and the developer doesn't build and deliver the whole thing, or if there's no practical way to test the developer's piece in isolation and <u>know</u> that it's really isolated from whatever else happens to be on your machine (including especially the simultaneous work of other developers) - well, you have a configuration control nightmare that might swamp any benefits of close work. Best to retreat to getting full builds at periodic intervals.

Setting the wrong expectations

Developers may expect great benefits at no cost. Set expectations carefully.

- Your job is to provide early information about bugs. That means the overall task - that of creating code that the project manager deems of sufficient quality - will go faster [Maguire94]. However, if the developer fixes bugs as you find them, the first part of the task - getting the first demonstrable version of the code into the source base - may go slower. The developer might find that annoying or inappropriate.

- Although you will be careful to conserve her time, you'll need information. You'll have to talk to her, or send her email. Both will take up her time, inevitably on days when she feels she just hasn't got it. Promise to work with her to minimize disruption. For example, in order to use time when they wouldn't be working anyway, I've interviewed developers at lunch, breakfast, and dinner, and while driving them to the airport or to pick up their car from the repair shop. But some disruption is inevitable.

- Make sure to emphasize that your relationship will unavoidably cause some friction. You *will* annoyingly discover a serious bug the day before she thought she was done, making her wonder why you couldn't have found the important things first. It is *much* better that she complain, even inappropriately, than "go dark" (disappear as a way of avoiding an unpleasant problem or person). Going dark won't work, anyway, because you'll be persistent. Politely persistent, but persistent. (I've said things like "I'm going to help you even when you can't stand it any more.")

Collaboration can fail

A close relationship is an investment of time. Sometimes investments don't pay off:

- Your programmer is so stubborn that you could scream - except that maybe no one would hear you over the sound of *her* screams about *your* stubbornness.

- The programmer seems deliberately obstructive or misleading. (She may be trying to conserve her time by directing you using fair means or foul. But sometimes she's just incorrigibly careless, or trying to hide incompetence, or whatever.)

- You've raised expectations that will be unmet. The programmer is unhappy with what you're doing.

My personal tendency is to pour more effort into a bad investment, to throw good time after bad. That's wrong. Sometimes you need to admit that a plan has failed and switch to a plan that will work better. In this case, the backup plan is over-the-wall testing: pulling

back, working at a distance, more formally, concentrating on the public source code at the significant builds.

Because it's easy for me to get trapped in a role, I find it important to explicitly assess the status of the relationship, especially early on. So I track how much time I'm spending communicating with the developer. At some point, I ask myself to be hard-nosed about whether that time is worthwhile. What is the character of the communication?

- Is it about answering specific questions and explaining specific points? Or is it about generalities? (When I'm stuck in a rut with a developer, I find myself writing long emails about the fundamentals of software testing as a way of explaining my decisions.)

- Do I find myself carefully crafting email to avoid any possibility of misunderstanding, or preparing myself carefully before talking to the developer? Those are preemptive measures against inefficient conversation, but they take much time themselves.

- Am I repeating myself? Is the developer?

- Some inefficiency is inevitable at first. Is it decreasing? Fast enough?

And I ask what I am gaining from this.

- Does what I'm learning about the code help me test? What good ideas have I gotten?

- What bugs have I found earlier than I would have?

- Am I spending less time writing, discussing, and revisiting bug reports?

- What's the chance that my effectiveness will improve?

If my frustration has been building, I ask these questions after a weekend away from work. Then I balance the costs and benefits. If I doubt I'll achieve the needed results, I may try a new communication approach for a short time, then re-evaluate. Or I may simply pull back. If there's one particular testing task on which we're stuck, I'll pull back only from that.

I do this with no recriminations. If there's blame to be had, I'm happy to take it on myself, for two reasons. First, enough time has already been spent not finding bugs: why spend more? Second, I am convinced that, were I good enough, I could work effectively with almost any developer.

Although there's no reason for fanfare about the change in plan, do make sure to communicate it clearly to the developer, your manager, and the developer's manager. They have a right to know, and not telling them would make any problems worse.

How are you a star?

A reviewer wrote: "From my perspective, your proposal is the opportunity for me to make the [developers] I work with look good and me look like I do nothing [...] Every test manager I've ever personally met has mouthed the mantra that bug count is not a valid

criterion for measuring a tester's effectiveness. Every one of them I've actually worked with has, at some point, used bug count to single out, reward, or promote testers."

This has not been a problem for me. For whatever reason, I've not been aware of managers using bug counts (except in a vague general-impression sort of way) to evaluate me. If your managers do that, my approach is a problem. Having you sacrifice your job at the developer's feet is a bit much. Sampling might work: suppose you work with four developers, one closely and three in the conventional way. Your manager might reasonably assume that your bug counts for the three were representative of your overall work. (This invites you to under-report the time spent working on their code, but any such simple measure invites "gaming the numbers".) Testimonials from developers, especially in the form of requests for your services, can work wonders.

Personality and experience

Some testers strongly prefer to work collaboratively; others are loners. This approach will reduce your interaction with the rest of the testing team. Your manager will have less visibility into what you're doing, which is bad if you get stuck without realizing it. That makes this approach riskier with inexperienced testers.

Some testers like to make a plan and follow it. Others are more tolerant of interruptions. This approach favors the latter. In addition to being tolerant, you must be well-organized enough that you still finish all your tasks.

The Stockholm syndrome

The "Stockholm Syndrome" refers to the tendency of captives to bond with their captors and adopt their point of view. For example, a hostage might come to view the police as the enemy.[5]

It's more than a little over-dramatic to see testers as the hostages of programmers. But by cooperating closely with developers, you may adopt their point of view and lose the customer's, causing you to miss bugs. In particular, you're likely to miss those bugs where the product does exactly what it was intended to do - but what it was intended to do is not what the customer would want. Usability problems are an example of this type of bug. So would be a scientific calculator that provided base-2 logarithms (useful mainly to programmers) rather than the natural logarithms most of its customers expect, or an online banking system that gave new users passwords the same as their user names. (A random password would be much better since many people never change their password.)

So we have a tradeoff between the risks and value of closer work. In most projects, most of the time, I believe the value outweighs the risk. "Most" does not mean "all". In safety-critical projects, for example, the added expense of truly independent testing will pay for itself if it prevents testers from absorbing the programmers' assumptions about possible failure modes.

---

[5] <http://www.vswap.com/confuser/stockhol.htm> is a nice short description that will probably be gone by the time you read this.

The risk of infection can be reduced by making explicit attempts to model the customer and other users. The following books taught me ways to keep the customer in mind even while talking to the developer: [Moore91], a marketing book that describes a process of target-customer characterization; [Cusumano95], which describes activity-based planning; [Gause89], a more mainstream software engineering book on discovering requirements; and [Hohmann97], which advocates describing user activities in the future perfect tense. (Except for [Gause89], the books cover a great deal of other material less relevant to testing.)

Some fraction of the testing team should concentrate on whole-product testing, rather than on testing features in isolation. Such testing is often organized around work flows (use-cases, scenarios, tasks), and is best done by domain experts (for example, accountants who test an accounting package by doing the things accountants do). These independent testers find bugs that feature testers infected by developer assumptions will miss (as well as feature interaction bugs and usability problems).

2. Explaining Yourself and Your Job

Whatever your role, you have to present yourself correctly, so that the judgement the programmer makes of you is the judgement you want.

This section does not assume you're following the approach described in the previous section. It applies to all situations where you must explain testing to a developer.

A good first impression

Testers are often hurt by programmers' snap judgements. The most common one is that testers want to be programmers, just aren't good enough to make it. So your first task is to avoid being classified too soon. Here are three ways.

Don't <u>ever</u> exaggerate your programming or technical skills

I was once in a meeting where a manager boasted "we did object-oriented programming in 1971; we just didn't call it that". Although he was correct in a narrow sense, he lost his audience that moment and never regained it.

The message he intended to send was "I'm technical, like you. It's safe to trust me for technical guidance." The message the programmers received was "I think I know more than I do. I will be resistant to the education I need. I will either waste your time in painful training or by making you do stupid things."

Testers who exaggerate their programming skills send the same message.

There's even a good case for downplaying your skills. Many programmers expect it. More than four hundred years later, they echo Castiglione's description of the ideal Renaissance courtier: "Therefore little speaking, much doing, and not praising [one's] own self in commendable deeds, dissembling them..." [Castiglione28]. Anything that looks like bragging is a warning sign.

Expertise in an odd field is useful

But failing to brag isn't enough to establish credibility. One surprising technique is transference. Suppose that you happen to be an expert on Greek oared ships of the 5th century BC. Programmers respect that sort of thing, because they value expertise in general. Expertise in weird subjects, ones that have little to do with programming, is valued more than expertise in allied fields. This respect for your "offline" expertise will transfer into respect for your testing ability. I don't know the reason, but it's worth noting that many really good programmers have surprising expertise in a completely unrelated field.[6]

But, again, be careful. I have a degree in English Literature, which gives me the background to scatter mildly entertaining and relevant quotes in papers. But it would be dangerous and wrong for me to claim expertise. I have a bachelor's degree, which is little more than a certificate of trainability. If I tried to pass it off as expertise, I'd run a substantial risk of running into someone who'd see right through me.

Place yourself outside the status hierarchy

My English degree doesn't make me an expert, but there's another way in which it's useful. When I'm in a Ph.D.-heavy situation, the fact that I have only a bachelor's and master's degree in computer science can be a negative. If things come down to ritual displays of credentials, my rank is clear - and it's lower than I'd like. But if I don't mention my computer degrees and instead describe myself as an English major, suddenly I can't be pigeonholed. I also acquire a slight cachet of being an interdisciplinarian, which specialists have a guilty feeling is a good thing. (Because of the widespread prejudice against English majors, this only works after I've displayed some technical competence.)

A tester can similarly defer classification. If, for example, you have green hair and play in a rock band most nights, it's harder for a programmer to put you in the "tester as wannabe programmer" pigeonhole.

Explaining your job

However, I wouldn't necessarily recommend dying your hair green. That's indirect. At some point, you must directly convince the programmer of your value. In this section, I'll present an approach based on the folk psychology of the programmer. *Folk psychology* is the way ordinary people explain why people do certain things. It's based on the notion that people have beliefs and desires, and that they act on those beliefs to achieve those desires. It differs from *real psychology*, which aims (in part) to discover what "beliefs" and "desires" are (if anything), and by what mechanism they are created and lead to actions. It differs from *pop psychology* in that it's not used to sell magazines or the products advertised in magazines.

---

[6] One technical field is useful for transference. There are testers who know everything about the hardware, who can figure out what's wrong with the printer, or why an ethernet card isn't working, or can configure any modem from memory. They're the ones developers go to when they can't figure out what's wrong with their machine. This is an expertise closely allied to programming, but one that confers real testing status.

For many programmers, a dominant desire is the respect of their fellow programmers. They believe that the way to achieve that is by having these characteristics:

- An ability to create concise and powerful abstractions. This is similar to the notion of "elegance" in mathematics and physics. Circumstances might prevent a good programmer from producing elegant code, but the ability and desire must be present.

- The ability to master a great deal of relevant technical detail. See, for example, the section titled "A Portrait of J. Random Hacker" in [Raymond96], in which Eric Raymond describes quickly absorbing a manual on a typesetting language in order to write a book. (This, in conjunction with the previous, makes programmers seem an odd combination of generalist and specialist.)

- Strong problem-solving ability, which consists of applying the consequences of those abstractions to the technical detail.

- High productivity, measured by amount of code produced. Any bits that aren't shipped (e.g., design documents) don't count nearly as much as shipped bits.

A legendary programmer would be one who was presented a large and messy problem, where simply understanding the problem required the mastery of a great deal of detail, boiled the problem down to its essential core, eliminated ambiguity, devised some simple operations that would allow the complexity to be isolated and tamed, demonstrated that all the detail could be handled by appropriate combinations of those operations, and produced the working system in a week.

Testing is a threat to the programmer's self-image when it strikes at those characteristics. It may demonstrate that the abstractions are not sound, the detail hasn't been mastered, or the problem hasn't been solved. It's not surprising, then, that programmers tend to divert testing away from those fundamentals and see it as a way of systematically exploring the code they've written, looking for coding errors. Coding errors are not a big deal. A programmer who is careless about coding errors might lose a little prestige, but could still be considered great. A programmer who never makes a coding error but produces clumsy abstractions never will be.

Now, to a tester, looking for coding errors is only a small part of the job. Conceptual errors, such as inadequate abstractions or mistaken assumptions, are more important. They cost more to fix.

This mismatch of expectations can lead to intense conflict. A programmer may see you as doing the wrong job, using the wrong approach. Discovering and explaining the right approach now becomes a problem to solve. And programmers are relentless problem solvers. (As my wife, a veterinarian who'd never known programmers before meeting my friends, said when we were courting: "You know, your friends have really strong opinions... about everything.")

Avoid this problem by explaining your job to the programmer. There are several parts to this explanation.

1. Explaining the difference between your job and her job.

2. Demonstrating that you possess a skill the programmer is not likely to develop by sheer concentrated reasoning from first principles.

3. Showing how your exercising of that skill helps the programmer realize her natural abilities to their fullest, much more than it casts doubt on them. In an important way, you compensate for the intractability and uncontrollability of the world.

Let me begin with a specific example. It's one that I thought of while writing this paper, so I've only tried it on one developer, but I think it's the best way I've yet found to describe my job. Here's a puzzle:

> *That man's father is my father's son,*
> *But brother and sister have I none.*

This is puzzling because the first line describes this situation:
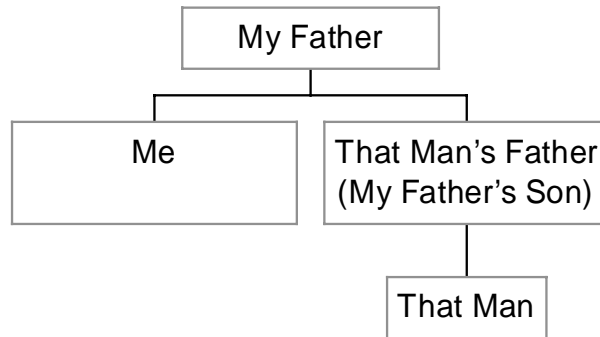
But the picture is contradicted by the second line, since my father doesn't have another son.

The puzzle is easily solved if you realize that the verse only *implies* that "that man's father" is someone different from "me". If they're the same person, you can draw this picture, which does not violate the second line:
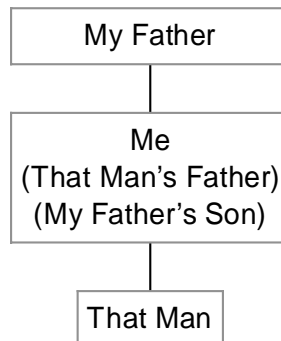
Now, I'd expect any programmer to quickly solve this puzzle - they're problem solvers, after all. But the key point is that *someone had to create the puzzle before someone else could solve it.* And problem creation is a different skill than problem solving. In this particular case, the puzzle creator wanted to confuse two situations and could do it after

realizing that fathers don't refer to their own son as "that man" or themselves as "that man's father". The skill lies in knowing how to mislead the puzzle solver.

This example is quite relevant to software testing, since one thing testers do is to wonder whether two different names could ever refer to the same thing. ("What if the input data file is the same as the output log file? Will the code overwrite its input before it reads it?")

Now, a programmer could argue that programmers can (and do) ponder such special cases themselves. They learn them by fixing bugs. However, testers make better problem-posers than developers do:

- It's the only thing they do. They see more bugs. They think more about bugs. They spend more time practicing problem creation.

- Just as puzzle creators know puzzle solvers, testers study programmers in order to gain insight into the kinds of things programmers might overlook. It's hard for programmers to think about the kinds of things they don't think about.[7]

- Testers also study users, in particular how to know roughly what a user knows, making it possible to do roughly what real users will do. This is difficult for programmers to do. Again, they may not have as much time. Also, they know too much about their solution to put themselves in the place of the user.[8]

So what a tester does is help the programmer master the relevant technical detail by presenting specific details (in the form of test cases) that otherwise would not come to her attention. Unfortunately, you often present this detail too late (after the code is written), so it reveals problems in the abstractions or their use. But that's an unfortunate side-effect of putting testers on projects too late, and of the unfortunate notion that testing is all about running tests, rather than about designing them. If the programmer had had the detail earlier, the problems wouldn't have happened.

Programmers help testers by describing how they'd test the program. That's useful for two reasons. First, it helps the tester understand where the programmer's blind spots might be. Second, it illuminates the tester's blind spots - testers overlook things too, of course.

That's the intellectual case. A tester helps the programmer realize her true ability by providing detail to master. That's an easy argument for a programmer to accept - at least provisionally. But there will be snags in practice, and they're most likely to be related to bug reports. That's the topic of section 3.

---

[7] To study programmers well, you need to become a good observer. I recommend [Weinberg85] and [Weinberg93]. And learn from what others have observed. See [Gabriel96], [Hohmann97], [Lammers86], [Levy84], [Turkel84], [Weinberg71] and [Weizenbaum76].

[8] In the section on the Stockholm syndrome, I mentioned books that would help you learn more about your particular product's users: [Moore91], [Cusumano95], [Gause89], and [Hohmann97]. I haven't read as much about the general user as I should have, but I can recommend [Cooper 95], [Norman90], and [Nielsen93].

Other oddities about yourself that you should explain

[Pettichord98] contains a comparison of programmer and tester characteristics. I want to emphasize three here, because I've seen them lead to friction.

Testers tolerate tedious tasks; programmers automate them. When programmers turn to testing, they often concentrate exclusively on automated tests. (Sometimes they spend so much time working on automation support that they somehow don't quite ever get to writing any actual tests...) When programmers see testers running tests manually, especially the same test over again, an opinion that testers are less competent is reinforced. Explicitly counter this by pointing out that the decision to automate is an economic one, and that the tradeoffs are possibly not what they think. Space does not permit me to develop the argument here; see [Marick98b]. Suffice it to say that the tradeoff is <u>not</u> between the time to automate a test and the time to run it *N* times manually.

Testers get up to speed quickly; programmers gain a thorough understanding. Everyone values a quick study, but programmers may not realize how often testers are presented with a product and almost no time to learn it. That places a premium on people who are able to "pick the low-hanging fruit": find out what they need to know, find the important bugs, and move on. Because programmers value thoroughness, such testers may appear shallow rather than effective, unless they explain themselves.

Testers believe ignorance is important; programmers that expertise is important. Testers know that ignorance of the structure of the product and of "the right way to use it" allows them to see what the programmer misses and the user sees. More expertise than the user has is a dangerous thing. Asking naive questions can produce surprising answers. Programmers need to be told that naiveté is a deliberate policy, not a shortcoming.

3. Reporting Bugs

This section describes how bug reports can be used to help build an effective relationship. It doesn't require you to use the techniques of the previous two sections. For a discussion with broader scope, I recommend [Kaner93] and [Kaner98].

Bug reporting is a part of an evolving relationship between you and a developer. You do two things with a bug report:

1. provide information about the state of the programmer's code. This is the familiar purpose of bug reports.

2. provide information about you and the degree to which the programmer can rely on you.

Initially, the developer should be suspicious. Some programmers have experience with testers who waste time with bug reports that are not useful. The rest have heard horror stories. You must convince the programmer that you are a useful person to have around. There are three parts to doing that.

Don't waste her time.

The bug report should not make her guess at information.

- If at all possible, include an explicit sequence of steps that will make the bug reproducible. <u>Try</u> the sequence of steps, as written, before reporting the bug. Make sure you start from the clearly described starting state (for example, make sure you exit the program and restart it).

- Say what you expected to happen, what did happen, and what was incorrect about it.

- Most programmers prefer to get bug reports that reproduce the problem in the minimal number of steps. Some do not, because it doesn't really save them time. (For example, the instructions on how to report a bug in the GNU C compiler explicitly told you not to bother with producing a minimal example.)

- Alternate sequences that also fail can help the programmer discover the underlying cause faster, but the sequences shouldn't be trivial variations of each other. Those waste the programmer's time deciding that there's nothing new. (It will probably take some time for you to learn the feature well enough to decide what variations aren't trivial.)

- Descriptions of scenarios that surprisingly <u>do</u> work can also help.

- If the problem doesn't happen on her machine, quickly take it upon yourself to discover the differences between your machines. Check future bug reports for configuration dependence. Learn how the product is sensitive to configurations. (This is, of course, useful in finding configuration bugs. Early in testing, it's equally useful in helping smooth the developer's path.)

Keep yourself out of it.

- Ruthlessly scrub your bug reports of any hint of personal criticism. You may need to let some bug reports sit overnight until you can read them with a fresh eye. You may find that asking questions is a less threatening way of conveying information than making statements. (And don't forget to ask genuine questions: there's a lot you have to learn.) Ask yourself how you'd feel if it turns out that what you thought was a bug is correct behavior. If you'd feel like a fool, there's a fair chance that part of the reason is your wording of the bug report.

- Don't try to solve the problem you report. It's unlikely you know enough to solve it well. Attempting to do so is likely to be counterproductive.

Demonstrate your value with important bugs.

You and the programmer are likely to have differing opinions about what's important. You might believe, as I do, that usability problems are critically important. The programmer might not. How do you resolve this conflict? You might barrage her with usability bugs, hoping to persuade her with volume. Or, you might first barrage her with bugs <u>she</u> believes are important, bugs so serious that she heaves a sigh of relief that they

were caught. Then, having established credibility as someone whose judgement is sound, you expand the scope of "important" to include usability bugs.

As you might guess, I favor the latter approach. I have in the past held back less important bugs, waiting to report them until after I've gained the programmer's respect. That's not something I do lightly, especially with the bugs that the programmer would believe are real bugs, just low priority. What if I'm hit by a bus? What if the programmer is transferred to another task and is now unable to fix minor bugs she might have otherwise fixed? But sometimes I've judged that it's the most effective thing to do.

Here are some other techniques to help keep the focus on important bugs.

- Explain why the bug would be important to a customer. If your scenario could be accused of being unrealistic, devise a more realistic scenario that also demonstrates the bug. If your programmer says that no real user would ever do that, first carefully consider if she's right. She may be. If she's not, marshal good arguments in favor of your scenario. What do the customer service people say?

- Many bug tracking systems have both a severity and a priority field. The severity field describes the consequences of the bug. The priority field describes how soon a bug should be fixed. Usually, a severe bug is also high priority, *but not always*. There may be good reasons not to fix it. As a tester, you are probably not privy to all those reasons, nor should you have the authority to schedule a programmer's time. Therefore, leave the priority field blank when reporting the bug. By doing this, you are avoiding defining "important" as "what's important to me, the tester".[9]

  If the bug tracking system has only a severity field, make it clear - in words and actions - that the value you give it does not represent priority. And try to get a priority field added.

- When you find a bug, seek further consequences. For example, I once tested some code that changed the name by which a user identified herself to the product. For aesthetic reasons, the name could not begin or end with a blank. (Since blanks are not visible, it's difficult for other users to know what the real name is.) I discovered that the code did not enforce that rule. Suppose I'd filed a bug report titled "Can create name that's hard for other users to type". It would have rightly been given low priority. However, I had recently discovered that the username was part of the name of a file on the user's disk. I also knew, from reading past bug reports and keeping up with developer email, that the product had a hard time with certain files with blanks in them. That inspired me to try various uses of blank-terminated names, uses that would interact with the disk file. I also tried names containing the DOS directory separator character '\'. I was able to find much more serious bugs. I don't remember the exact details, but the bug report title could have been something like "All session data lost (names with backslash or trailing blanks)". That's a bit more compelling.

---

[9] Like any other employee, if you strongly disagree with the decision to defer a bug, you should consider fighting it. But pick those battles carefully, and let the decision be made before you inject your opinion.

As you gain rapport with a developer, you'll be able to spend considerably less time on private (email) bug reports. Most of the suggestions given above also make bug reports more useful to their other possible consumers: other testers, project management, people concerned with development process or metrics. So you should continue to take nearly as much care with bugs logged in the public bug tracking system.

Summary

So. Where do we stand after all these pages?

Testers often complain that the developer is king. Because of that, testers don't get the attention, respect, and resources they want.

But what if you accept this situation, then run with the metaphor?

My perspective is that the developer is often a benevolent and useful monarch, just sadly and inevitably incomplete in her knowledge of the user and the nature of her own fallibility. The effective counselor seeks to complete the monarch, to make her wiser in the eyes of the world, not become monarch himself.

Tyrants, of course, must be overthrown.

Acknowledgements

Bibliography

[Bach97]
James Bach, "Good Enough Software: Beyond the Buzzword", IEEE Computer (Software Realities column), August 1997.
[Castiglione28]
Baldassere Castiglione, *The Book of the Courtier*, 1528.
[Cooper95]
Alan Cooper, *About Face: The Essentials of User Interface Design*, IDG Books, 1995.

[Coplien95]
James Coplien, "A Development Process Generative Pattern Language", in *Pattern Languages of Program Design, Volume 1*, James Coplien and Douglas Schmidt, eds., Addison-Wesley, 1995. <http://www.bell-labs.com/people/cope/Patterns/Process/section24.html>

[Cusumano95]
Michael A. Cusumano and Richard W. Selby, *Microsoft Secrets*, The Free Press, 1995. Reviewed at <http://www.rational.com/connection/books/reviews/>

[Gabriel96]
Richard P. Gabriel, *Patterns of Software,* Oxford University Press, 1996.

[Gause89]
Donald C. Gause and Gerald M. Weinberg, *Exploring Requirements*, Dorset House, 1989. Reviewed at <http://www.rational.com/connection/books/reviews/>

[Hohmann97]
Luke Hohmann, *Journey of the Software Professional*, Prentice Hall PTR, 1997.

[Kaner93]
C. Kaner, J. Falk, and H.Q. Nguyen, *Testing Computer Software (2/e),* Van Nostrand Reinhold, 1993. Reviewed at <http://www.rational.com/connection/books/reviews/>

[Kaner98]
Cem Kaner, course notes for *Black Box Software Testing*, Summer, 1998. <http://www.kaner.com>. The discussion of bug analysis and reporting contains some excellent material not present in [Kaner93].

[Lammers86]
Susan Lammers, *Programmers at Work,* Microsoft Press, 1986.

[Levy84]
Steven Levy, *Hackers*, Anchor/Doubleday, 1984.

[Maguire94]
Steve Maguire, *Debugging the Development Process*, Microsoft Press, 1994. Reviewed at <http://www.rational.com/connection/books/reviews/>

[Marick95]
Brian Marick, *The Craft of Software Testing*, Prentice Hall, 1995.

[Marick97a]
Brian Marick, "Classic Testing Mistakes," Software Testing, Analysis, and Review (STAR) conference, Software Quality Engineering, May 1997. <http://www.stlabs.com/marick/classic.htm>

[Marick97b]
Brian Marick, "The Test Manager at the Project Status Meeting," International Quality Week conference, Software Research, May 1997. <http://www.stlabs.com/marick/status.htm>

[Marick98a]
Brian Marick, "The Testing Team's Motto". <http://www.stlabs.com/marick/purpose-of-testing.htm>

[Marick98b]
Brian Marick, "When Should a Test Be Automated?" International Quality Week conference, Software Research, May 1998.

[Moore91]
Geoffrey Moore, *Crossing the Chasm*, HarperBusiness, 1991. Reviewed at <http://www.rational.com/connection/books/reviews/>

[Norman90]
Donald A. Norman, *The Design of Everyday Things*, Doubleday, 1990.

[Nielsen93]
Jakob Nielsen, *Usability Engineering*, AP Professional, 1993.

[Pettichord98]
Bret Pettichord, "Developers Take On a Full System Test," Software Testing, Analysis, and Review (STAR) conference, Software Quality Engineering, May 1998.

[Raymond96]
Eric S. Raymond (compiler), *The New Hacker's Dictionary (3/e)*, MIT Press, 1996.
<http://www.logophilia.com/jargon/jargon.html>

[Turkel84]
Sherry Turkle, *The Second Self: Computers and the Human Spirit*, Simon & Schuster, 1984.

[Weinberg71]
Gerald M. Weinberg, *The Psychology of Computer Programming*, Van Nostrand Reinhold, 1971.

[Weinberg85]
Gerald M. Weinberg, *The Secrets of Consulting*, Dorset House, 1985.

[Weinberg93]
Gerald M. Weinberg, *Quality Software Management, Volume 2: First-Order Measurement*, Dorset House, 1993.

[Weizenbaum76]
Joseph Weizenbaum, *Computer Power and Human Reason*, W.H. Freeman, 1976.