The Tester's Triad: Bug, Product, User

Brian Marick, Testing Foundations
marick@testing.com
www.testing.com

I am not, by nature, a particularly gifted tester. But I am, in practice, a pretty good one. Why? I spend time observing gifted people. I figure out just what it is that they do that doesn't come naturally to me. Then I do those things.

This paper is about what I've observed. I claim that good testers pay attention to three things – bug, product structure, and the user – in ways that less good testers do not.

Background

There are many different kinds of testers, working in different situations. Most of my testers have been in situations like the following. Testers in similar situations will be best served by this paper, though I believe it has broader applicability.

1. They were "black box testers". That is, they executed their tests through the same interface that the end user uses.
2. They were typically testing a particular feature in a product, such as a mail program's Address Book feature. (Such a feature allows you to associate an abbreviation with an email address, so that you can send mail to "dawn" instead of having to remember d132@notarealaddress.com.)
3. They were usually brought into the project late, after the feature was mostly coded.
4. "Upstream" documentation like requirements documents, specifications, and design documents were often missing. When they existed, they were out of date. User documentation (user manuals, help pages, and the like) probably existed, but were often incomplete.

Everyone would probably agree that (3) and (4) cause problems. I think (1) and (2) do as well, for reasons I'll explain later.

The Tester's Triad compensates for these problems.

Using Bugs

Many bug tracking systems have an email notification feature. You can ask to be mailed copies of any bug reports filed against a particular area. In a medium-sized project, I would expect a good tester to get – and read – email for **all** the bugs filed against **any** part of the system. In a larger project, there are too many bugs to do that. But I'd still expect a good tester to read bug reports in areas other than her own.

A good tester reads bug reports.

Why? Let me illustrate with a story.

One day, I received email from a casual professional acquaintance. I was one of several people on the "To" line. The mail looked something like this:

> To: …, "Brian Marick" <marick@testing.com>, …
> Subject: Fwd: Check this out!
>
> Truer words were never spoken!
>
> Attachment: picture.jpg

Well. I had no idea what was going on here, but JPEG files can't contain viruses, so I opened it.

<You're expecting something embarrassing, right?>

It was a picture of the back of a man wearing a black T-shirt with white lettering. The message was: "I am a bomb technician. If you see me running, try to keep up."

I thought that was pretty funny. I also had no idea why she sent it to me. So I asked. Here's what I remember of her explanation.

She uses an email program that allows you to have several address books. She had one address book for professional acquaintances. She had another one for personal acquaintances, including various fun people to whom she might forward amusing mail.

As the sort of person who still uses "to whom" in sentences, I'm obviously not qualified to be in a "fun people" address book. But, as a testing person with whom she'd once corresponded, I was in the professional acquaintances address book, under the alias "brian".

Unfortunately, she also knows a fun person named Brian. His email address is in her fun people address book, also under the alias "brian". So when she composed her message:

> To: mark, sue, bill, **brian**, betty…

there was an ambiguity. Her mailer is notorious for resolving such ambiguities in ways predictable only to the programmer who coded the rules. In this case, it resolved it wrong.

No harm was done to my correspondent, but you hear stories of people being much more than mildly embarrassed by similar occurrences.

I think this is inarguably a user interface bug. I've just given you a bug report. What would I expect you, as a tester, to do with it?

We have a situation in which a "brian" appeared in two address books. Let's generalize: this bug was caused when a particular **name** appeared in two **places**. Suppose you were testing the Addressing feature of a mail program (the code that takes email addresses on the To, CC, and BCC lines and does the right thing with them). I would expect you to apply this generalized notion to your situation. Each Addressing line is a **place** in which a **name** can appear. I'd expect a test like this:

> To: marick@testing.com, billg@microsoft.com
> CC: marick@testing.com

My name appears in two places. (You'd have many other tests as well, of course.)

Testers read bug reports so they can make generalizations that they can use in other situations. There's a famous quote by Isaiah Berlin (after Archilochus) in his essay *The Hedgehog and the Fox* [Berlin53]:

> *The fox knows many things, but the hedgehog knows one big thing.*

What does **that** mean? It means that Berlin (and I) claim that it's often useful to divide people into two kinds. One kind seeks fundamental principles or rules that explain *everything*. ("All modern history is driven by the conflict between capital and labor." "From this set of axioms, one can work out all of mathematics".) Those people are hedgehogs.

The foxes use a large number of less fundamental rules in their day-to-day life. They're more likely to believe that effective knowledge comes from the accretion of lots of facts than from the seeking of first principles. My wife (a practicing veterinarian as well as a professor of Veterinary Medicine) seems to me a fox[1]. When diagnosing, she does not first go to first principles; instead, she works from a truly amazing memory of a multitude of diseases and symptoms. Among the sciences, biology is much more of a "foxy" science than nuclear physics.

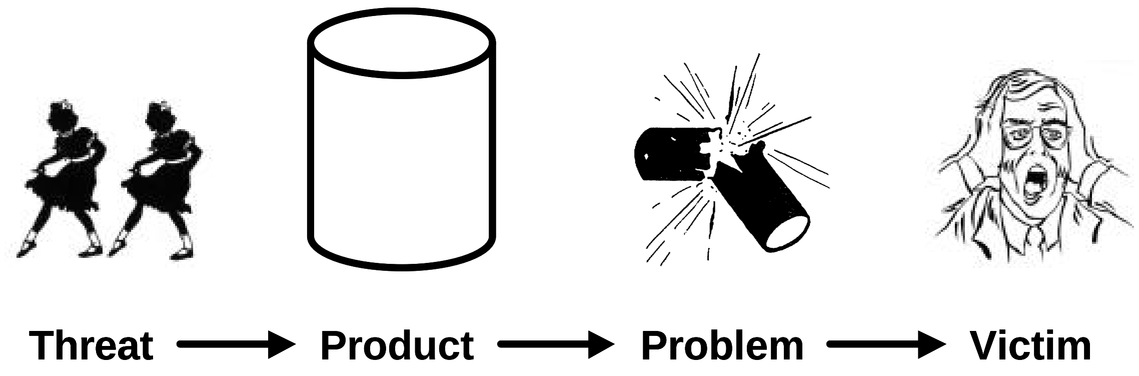
(See also Freeman Dyson's essay "Manchester and Athens" in his book *Infinite in All Directions* [Dyson88].)

I claim that good testers are foxes. I claim they do **not** apply a few general rules to all situations. Instead, they match a vast number of special rules to specific situations. They get those special rules from bug reports.

[1] In both senses of the word, for those whose slang comes from the 60's and 70's.

More about using bug reports

There's more to using bug reports than that. Let me explain in the context of James Bach's model of project risk. Here's a picture of his model:



What does it mean to say that a product has a bug? First, the product must be presented with some **Threat** – something in the input or environment that it cannot handle correctly. In this case, the threat is two cute but identical twins. (This corresponds to the common case of software being unable to handle duplicate inputs. That's not the threat we talked about in the previous section, but I couldn't find a good illustration for that one.) Next, the **Product** must perform its processing on that input or in the context of that environment. If it cannot handle the threat, it produces a **Problem** in the form of a bad result. But a problem means little or nothing unless there is some **Victim** who is damaged by it.

In the previous section, I concentrated on how testers learn about Threats from bug reports. They also learn about the Product, the Problem, and the Victim.

Product

By reading bug reports, you learn about "hot spots" in the code. For example, you might discover that the product as a whole has a lot of problems handling network timeouts and various other network glitches. That might inspire you to think of how network glitches could affect *your* feature.

Problem

In desktop applications, a window's title bar often contains useful information (program name, file being worked on, etc.) When there's more text than fits in the title bar, a product should do something sensible (not crash, truncate meaningfully, etc.). Many testers before you have discovered that their product isn't sensible. Perhaps your product has similar Problems. Perhaps you can think of ways to make your title bars overflow (such as asking for translations into a language that, on average, uses more characters per word than yours does).

Victim

Bug reports don't just go into the bug database and sit there. Someone reads them and decides what to do about them. Should they be fixed? Deferred? Marked as "not a bug"? In some companies, that someone is called the "bug triage team" or "bug council".

Ideally, the bug triage team would be composed of purely rational entities that weigh each bug on its merits. It is in fact composed of human beings, who are both busy and filled with idiosyncrasies.

It is your job to write bug reports that are **effective**, that convince the bug triage team to make the right decisions. It is not enough for you to be right; the triage team must be too. By tracking the progress of other people's bug reports, you learn how to deal with the triage team's idiosyncrasies, learn about how to make persuasive bug reports. (For more on this, see [Kaner93], [Black99], and [TestingCraft].)

One way to make a convincing bug report is to tell a story. You must put into the imaginations of the bug triage team a plausible tale about the damage a bug would do to an important victim. Again, to do this well, you must understand what the bug triage team thinks is important.

Let me summarize how a tester uses a bug report with this table:

	Threat	**Product**	**Problem**	**Victim**
Bug	Threat types	Hot spots	Problem types	Convincing bug reports

The two other elements of the Tester's Triad will add more rows.

Using Product Structure

One place you'll find a good tester is sitting in a room while a developer draws circles and arrows on the whiteboard. Testers talk to developers to learn about product structure: what are the pieces? How do they fit together?

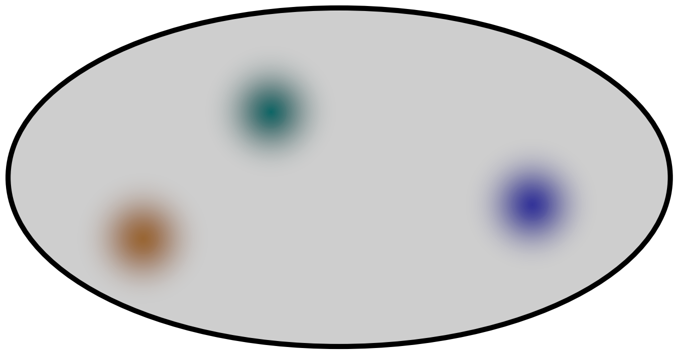
A good tester builds a mental model of product internals.

A good way to begin such a conversation is by asking for an explanation of what went wrong to cause the failure in bug 3434. (See also [Bach99].)

Why are testers talking to developers? Why aren't they reading design documentation?

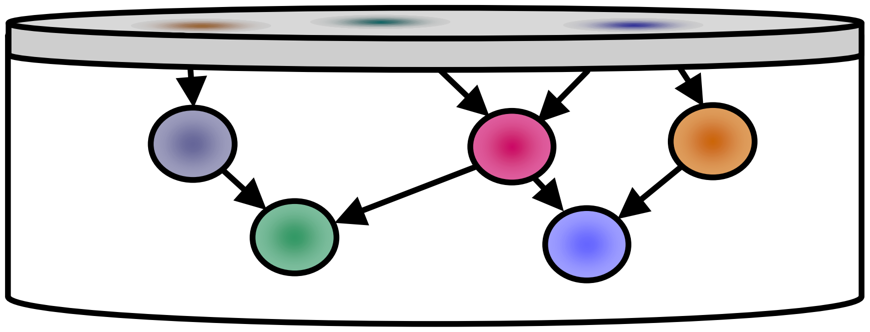
I'm not entirely sure, but I suppose that the reason is that design documents are static. They describe relationships between unmoving entities. People seem to learn better – and to better retain knowledge – when those static relationships are put to use in a dynamic description, when the explanation is of the form "First this happens, then this module passes this data to that module, and that module stashes it here for when…[2] Asking for an explanation of a bug report makes for a good example to trace through the system.

Why is an internal model useful? Here's the picture you as a black box tester see of the product:



The fuzzy blobs are features to be tested. (I'll explain why they're fuzzy in a moment.)

But there's a great deal happening beneath that surface you see. The whole product might look like this:



The surface features communicate with **subsystems** that perform parts of the task. Those subsystems communicate in turn with other subsystems. The subsystems are quite often shared among features.

This sharing explains why it's often difficult to tell whether a particular something the product does is part of your feature or some other feature (and thus whether testing it is your job or someone else's). The action is in fact done by something shared between the two features. (It's this fuzzy division of responsibility at the feature or external interface level that I meant to imply with the fuzzy blobs in the first picture.)

If you know about the internal structure, you can make better testing decisions:

[2] Is it an accident that we call good speakers "dynamic speakers"? If you were a speaker, would you want to be called staid? steady? See [Lakoff80].

1. Without knowledge of internals, testers often write redundant tests. If your feature and Betty's feature both use a math library, it makes no sense for both of you to test whether the `sqrt` function actually works correctly (that is, that the square root of 4 is 2). That testing should be done by only one of you (or by some "white box" tests specifically targeted at the subsystem). Both of you will have to test that your feature **uses** the `sqrt` function appropriately (for example, that neither of you try to take the square root of a negative number), but that's usually a much smaller set of tests.

2. Suppose Betty finds a bug in the math library, but it could affect a user of her feature in only a minor way. If you know something of how your feature uses the math library, you might be able to find a corresponding, but much more serious, manifestation of the underlying problem. (Note that you need to read Betty's bug report to know to try.)

3. It's often the case that subsystems store data. That data is, in effect, shared among different features. A tester who knows what data is shared can often exploit that knowledge.

   Here's a fanciful (and implausible) example. (For a real but more technical example, see [Marick99b].) Suppose that you know that putting someone's email address in the To field of a mail header stores that address in the "most recent person" variable. And suppose that you know that, for some efficiency reason, putting an address in an address book does the same thing.

   That's a bug. A persuasive bug report might explain it with this story of how a Victim experiences a Problem:

   > *Joe is sending a love letter to his sweetie in another company. He wants to make sure it's just right, so he doesn't hit* `Send` *right away. He decides to wait a bit, then double-check it. So he pulls down new mail. One is from a major client, wanting to start a product evaluation. Since Joe knows he'll be corresponding with this person often, he adds her to his address book.*
   >
   > *Having read all his new mail, he then re-reads his love letter and sends it. Unfortunately, the program uses the* most recent person *address, so he sends a mash note to the major client.*
   >
   > *Now, Joe might be an irresponsible employee, but should his company lose business because of it?*

We can add internal knowledge to our table of why good testers do what they do:

	Threat	Product	Problem	Victim
Bug	Threat types	Hot spots	Problem types	Convincing bug reports
Product	Support code problems	Connections between subsystems		

Now, one objection to this idea is that testers don't have time to learn about the guts of the product. However, it's been my observation that the product model doesn't have to be very detailed. You can get a lot of benefit out of a rough, high-level model, one that a non-programmer can readily develop.

In fact, I've seen testers work effectively with models that I knew to be flat-out *wrong*. But even those wrong models provided value in the form of useful testing ideas.

As in many things testing, perfection isn't the goal – the goal is to be good enough. I am actively researching what "good enough" means, and what techniques one uses to get good enough knowledge [VisibleWorkings].

Using the User

A good tester's conversation is often in terms of the users: what their goals are, what tasks they perform, and how they go about those tasks. When first looking at a feature, a good tester will as a matter of course think about the user's reaction to that feature, how it will affect what the user does, and what mismatches it might have with the user's habits.

A good tester understands how users spend their days.

Why is this useful?

One reason has to do with a problem with the approach in the previous section. It's all very well to talk about exercising interactions between subsystems or between features. But *which* interactions between *which* subsystems? Even though you'll unaware of many interactions (remember, you have imperfect knowledge of the interior), there will still be too many possibilities to try them all.

If you can't exercise all interactions, it's best to exercise those that are most likely to cause problems to users (victims). To do that, you must understand the users. You must, for example, realize that a user may be performing several tasks at once. When the computer is crunching away at one of the tasks, she's likely to switch to another. Maybe she'll start up another copy of the same program. If that new copy changes some files being used by the first copy… blam!

These are exactly the sort of interactions that programmers overlook. Feature testing, which concentrates on one feature at a time, also overlooks them. But the total testing effort should not.

There should be task-based testing that’s guided by an understanding of the user. In addition to covering the interactions you’ve decided are most important, it will also unintentionally exercise important interactions you didn’t know about.

Another reason for understanding users is that it enables better bug reports. The product’s usability is important. But we as testers have been trained not to report usability problems. And yet we are probably the closest approximation to the expert user available to the project team. (Usability testing typically concentrates on novice users, not on people like us – people who use the product many hours a day for many days). As such, we can discover expert tasks that cannot be done, tasks that are needlessly difficult, tasks that are needlessly error-prone, and features that promote misunderstandings.

When we report these bugs, the likely reaction is “works as designed”. Our counter is, essentially, “So redesign it!” We have to express that message politely, of course, and we have to present it persuasively: by being a credible and knowledgeable representative of the user.

Getting there

The question is how one learns about the user. I don’t have a satisfying answer for that.

I have observed that good testers really *care* about the users. They have a protective attitude that is often not shared – at least not so strongly – by the rest of the product team. I think this leads to testers thinking of users as people, rather than abstractions. That engages empathy and imagination, and leads to better test cases.

Let me compare testing to a related field, software marketing and product design. An important book in that field is Geoffrey Moore’s *Crossing the Chasm* [Moore91]. In it, he says:

> *Neither the names nor the descriptions of markets evoke any memorable images – they do not elicit the cooperation of one’s intuitive faculties. We need to work with something that gives more clues about how to proceed. We need something that feels a lot more like real people. However, since we do not have real live customers as yet, we are just going to have to make them up. Then, once we have their images in mind, we can let them guide us to developing a truly responsive approach to their needs. (pp. 94-95)*

He describes a process of target-customer characterization that includes naming the person, sketching their history and goals, and so forth. That seems a little silly, a little far afield from our technical task of finding bugs. But finding bugs is a creative task, in part, and creativity comes on its own terms.

Glendower: I can call spirits from the vasty deep.
Hotspur: Why, so can I, or so can any man;
But will they come when you do call for them?
Shakespeare, *Henry IV, Part 1 (III, I, 53)*

Target customer characterizations are closely related to "use cases" and "user stories". See [Beck99] and [Pols].

Testing that is entirely analytical and painstaking and planned will miss bugs. It is difficult to keep the user in mind while you're doing something quite different from what a user does (writing an automated test script; stepping through test case checklists written months ago). For that reason, many good testers practice some form of **exploratory testing**, which has more of an emphasis on an interweaving of test design and creation – just the kind of thing users do, except they're "doing their job" instead of "creating tests". For some writings on exploratory testing see [Exploratory].

I believe the act of exploratory testing will increase your understanding of the user.

This final table summarizes how keeping the user in mind further addresses Bach's four risk areas:

	Threat	**Product**	**Problem**	**Victim**
Bug	Threat types	Hot spots	Problem types	Convincing bug reports
Product	Support code problems	Connections between subsystems		
User	User actions	Picking subsystem traversals		Reactions to program actions; impact of bugs

Reprise: Testing Problems

In the introduction, I listed four problems. Two were unexceptional: being brought in late and not having upstream or high-level documents. Two I expect to be controversial: being a black box tester, and feature testing.

I list these as problems because it seems to me that I observe good testers pretending to be the first, and to perform the second, while actually doing something more. They make rough models of the internals despite supposedly working from the interface a user sees.[3] While supposedly doing feature testing, they add "irrelevant" steps to tests in order to exercise interactions. They lean toward exercising their features in the context of user tasks.

[3] I don't deny that users also make models of the internal workings of products they use, but they do it differently. For example, they don't interview developers.

That's good. It grates against the rather tired, rather rote way we look at testing. Our terminology and approach haven't changed much in the last 20-some years, and I think that's a problem [Marick99]. If the best testers are doing something beyond the received wisdom, we need to fix the received wisdom.

References

[Bach99]

James Bach, "Risk-based Testing", *Software Testing and Quality Engineering Magazine*, Vol. 1, No. 6, November/December 1999.
http://www.stqemagazine.com/featured.asp?id=7

[Beck99]

Kent Beck, *Extreme Programming Explained: Embrace Change*, 1999.

[Berlin53]

Sir Isaiah Berlin, *The Hedgehog and the Fox: an Essay on Tolstoy's View of History*, 1953.

[Black99]

Rex Black, *Managing the Testing Process*, 1999.

[Dyson88]

Freeman Dyson, *Infinite in All Directions*, 1988.

[Exploratory]

Various authors writing on exploratory testing. http://www.testingcraft.com/exploratory.html

[Kaner93]

Cem Kaner, Jack Falk, and Hung Quoc Nguyen, *Testing Computer Software*, 2nd edition, 1993.

[Lakoff80]

George Lakoff and Mark Johnson, *Metaphors We Live By*, 1980.

[Marick99a]

Brian Marick, " New Models for Test Development" *Proceedings of International Quality Week*, May, 1999.
http://www.testing.com/writings/new-models.pdf.

[Marick99b]

Brian Marick, "Interactions and Improvisational Testing",
http://www.testingcraft.com/exploiting-interactions.html

[Moore91]

Geoffrey A. Moore, *Crossing the Chasm*, 1991.

[Pols]

Pols Consulting, Ltd. The Use Case Zone, http://www.pols.co.uk/usecasezone/

[TestingCraft]

Various authors on writing good bug reports.
http://www.testingcraft.com/techniques.html#bugs

[VisibleWorkings]

Brian Marick's site for research on an adequate understanding of programs.
http://www.visibleworkings.com/