# What Is a Defect Anyway?

**Reevaluating our understanding of defects**

### A definition

What is a defect anyway? If you ask 10 people, you're bound to get ten different answers (or at least 9 different answers and a dumb look). For years, a war of words has been raging about how to define just what defects are, and just how important fixing them really is. Defects, or bugs, have been defined, categorized, classified, rated, and still we can't seem to agree. So why is getting some kind of consensus like trying to nail Jell-O to a tree? Because developing and testing products are, in most cases, distinctly different activities. You have different groups, each with different (often oppositional) charters, each asserting its own definition on a single process. Probably the most accurate and succinct definition I've heard came from… well, me.[1] I was giving a presentation on exploratory testing methods and someone in the rear of the audience (they're always in the rear of the audience) spouted out the "what's a bug" question. I paused, poised my head on my hand Rodin-style, pontificated for a moment, and then answered, "I'm not sure, but I know it when I see it."

There are surely more formalized definitions of what constitutes a defect than my semi-intuitive response above. Most projects define defects as operation or functionality that fails to meet the product functional specification. Where the functional specification document calls out the ability to do a specific task or operation with the product, a failure to do that task successfully constitutes a bug. This definition, however, seems a bit nebulous, allowing too much wiggle room if debates arise over whether something is a defect or not. The lines heard most often in these cases are "that's not a bug, that's a feature," "it's not in the spec[ification]," and "it's working as designed." My favorite rebuttal to the last one is that it's working as poorly designed. Unfortunately for simplicity's sake, none of these arguments actually resolves whether something should be classified as a genuine defect (which initiates the formal process of fixing an issue). Is there a way to solidify this definition so that everyone involved is working from the same set of rules? Probably not, but that shouldn't keep us from trying.

[1] Actually, I was paraphrasing (well out of context) Supreme Court Justice Potter Stewart, who, in Jacobellis v. Ohio, 378 U.S. 184, 197, deferred to his own intuition when proclaiming that he didn't know how to define obscenity.

Establishing a universal definition of what constitutes a bug requires a thorough understanding of the objectives of both your test and development organizations (this may actually explain why a universal definition is so hard to achieve). It might be surprising to some readers to think these two organizations could have different objectives, since the goal of producing a quality product, that meets the customers' needs, in a timely manner, seem self-evident. In practice, however, the two teams' objectives often deviate from this harmonious aim. This divergence can range from the slightest differences in what defines a quality product, to conflicting ideas of why products are tested in the first place, and can reach the powerful accusatory tones of "Development as an impediment to quality" and "QA as an impediment to market." At these extremes, the caustic environment created can rip through morale in both groups and cause irreparable damage to the program. Unfortunately, I've seen a number of organizations that not only allow this situation to develop, but actually promote it.

The basis for these oppositional views can be traced to the duties of each team in the traditional model. The developer creates a tangible product; one that the customer can see, use, and the most visible of these concrete measurements, buys. The development group spends its time creating and integrating defined features into a complete package. While it may seem otherwise at times, development is a finite process. There is an end. This closed system gives the developer specific objectives, identifiable checkpoints, and clarity about what constitutes 'the end.' Things that stand in the way of these checkpoints (and especially the end) often appear as roadblocks to success. QA organizations ferreting out issues—breaking the product, delay 'progress' through these checkpoints, so the testers (or their work) become barriers to success.

The tester lives in a different world. For QA, the objective is to minimize risks by improving product quality. This sounds lofty but the objective is nebulous at best since, in even the finest product, there is opportunity to increase the level of quality—anything can be improved upon. So the tester is caught in a never-ending product improvement process, reaching for the impossible goal of ensuring the product is perfect. When the project is declared "finished" and ships to the customer, the tester never really knows if all the bugs were found—in fact, the tester is confident there are still bugs that went undetected by the test process. This can be verified with a quick glance into the bug tracking database.) In other words, for the tester, there is no end, only the realization that an unfinished product is in the customer's hands.

Because every environment is different, there is no magic bullet to offer here, but understanding these divergent views goes a long way to defining just what a defect really is. What needs to be done now is to understand the significance of these bugs, and how we determine that significance. To do this, we need to look more closely at how to measure defects.

#### Ratings

Bugs tend to be classified by severity—allowing management to determine, within a limited development schedule, which ones should have resources assigned to fix. The scale tends to look something like the following—

| Severity | | |
|---|---|---|
| 0 | Critical issue | Issues involve data loss, data corruption, or whole system failure. Customer impact could be devastating. |
| 1 | Important issue | Major functionality is missing in key components of the product. Some kind of workaround usually exists. Customer impact would be significant. |
| 2 | Minor issue | Cosmetic problems or minor functionality issues. Customers may consider these annoyances, or possibly a sign of a sloppy development/test process, which reduces overall customer confidence in the product. |
| 3 | Change request | These include cosmetic issues, typographical errors, awkward menus, etc. These are generally considered low priority (though not always—read on). |

*Table 1. Typical bug severity scale*

Severity 0— These defects are generally considered 'show stoppers' in that the product cannot ship with this type of defect. These almost always trigger emergency-like responses from Development (though they should not always, as we'll see later).

Severity 1—While these defects are major in the context of final product quality, they do not impede the test process because some workaround to the problem exists.

Severity 2—These issues are generally not damaging to product operation, but instead, are more of an annoyance to the user. Severity 2 issues are sometimes ignored or forgotten, but are often the easiest to fix.

Severity 3—Issues that fall outside the product specification or the architectural documentation often fall into this category. They are, in the strictest engineering definition, not defects (but I make the argument later in this article that this definition is wrong).

While the actual scale may vary somewhat between organizations, the general breakdowns are the same—"It's really really broken," "It's broken but doesn't fail the entire product," "It's got some small glitches but is mostly alright," and "in your incredible amount of free time, could you change this one tiny thing?" Sometimes the last category is split into two, with the higher category covering the cosmetic issues and the lower one logging enhancement requests for future product revisions. So it goes, for now.

#### Prioritizing

Due to the inevitably short project schedule, decisions have to be made about which of these bugs gets fixed. They have to be prioritized. But in the scalar "severity only" model many organizations use, severity equals priority. Severity zero bugs are generally attended to before ones, twos, and threes—in that order.

For the defect that panics the system kernel and spills customer data on the floor, the priority is pretty high, so the severity rating is at the top (in the example scale above, Severity 0). This is almost always an unacceptable condition. Data integrity is of utmost importance, because if the data lies to the customer just once, the data can never be fully trusted again, and unreliable data is worthless. Companies have lived and died over data integrity. The oft-adapted phrase that best sums this up is "It's the data, stupid!"

When a user attempts to configure the product using a wizard or GUI, and it fails, requiring them to use a cryptic command line interface or hack through some kind of configuration file to complete the task, this falls into the category of severity one. The purpose of the GUI (or wizard) is to create a 'friendly' user interface (or at least a civil one[2]) that allows configuration, control, or use of the application. This is the part of the package that makes the product easy to use, which is why customers buy products in the first place—to simplify and expedite tasks. If the product is too difficult to operate, the task has not been simplified, so customers no longer have a reason to buy it.

For the context-sensitive help screen that is difficult for some readers to understand, this is a minor problem and warrants being labeled severity two. While the help screen should be helpful to the end user (that's why it's called the help screen), it cannot be all things to all users. The novice user may require a great deal of step-by-step assistance and is lost without it. The experts (or at least the impatient ones) may go mad if they have to read more than a few sentences to understand a topic. In this case, you must know your intended audience. What does the customer expect out of this feature? If it's not easy for your intended audience to use, you have a customer annoyance issue.

Where some customers have commented that your previous product should have included shortcuts or "hot keys" for common tasks, these issues fall into the change request category (designated in the table above as severity three). It is your customer who will use the product, so serious consideration should be made as to what kind of features the customer is looking for. In the traditional model, where testing comes late in the process, it is usually too late to ask your customers for input. The QA organization, however, may be able to communicate these types of issues by testing the product from a customer perspective. Understanding the customer perspective should be something your Quality group is trained to do. (Actually, if customers are asking for features like these, you may have more than severity 3 issues on your hands; but "architectural requirements" is outside the scope of this article).

[2] With as many poorly written GUIs as I've had to work with, I'm being generous on the "friendly" comment. However, to the 'defense' of those developers who make such imperceptive arguments, the acronym 'GUI,' for Graphical User Interface, never mentioned anything about being user-friendly. Besides, isn't the 80x25 character set actually 'graphical' when you get right down to it?

So, we have it, as the project schedule is getting tighter and tighter, project management begins culling through the issues to determine which ones get fixed first, claiming to have full intention to work through the entire list (but if we were going to fix everything on the list, I'm not sure why we'd have to prioritize the bugs in the first place). As the schedule locks its clammy hands around management's collective necks, issues defined with low priorities start being labeled "do not fix" or "deferred for future release" and (at least mentally) discarded from the stack. So now we're only fixing the really big problems and leaving the smaller ones for *later*.[3]

Now that we have a bug classification defined, an even bigger point of contention between Development and QA organizations usually revolves around applying the scale to an actual defect. The tendency I have witnessed is that testers are prone to overrate the severity of bugs while developers often underrate them. If you think about it, this is actually what you might expect, given the element of human nature. Though we'd like for all organizations to apply the same judgments against the same scale, Quality Assurance engineers and managers often feel the need to increase the severity of an issue so as to attract attention to it in hopes of seeing it fixed. If they believe that more attention is being focused on severity 2 bugs than those of severity 3 (which is, in all likelihood true, as discussed earlier), here is a way to attract additional attention to an issue. Fudge on the severity. Make that severity 3 issue that's really bothering you, a severity 2 so the development team will look at it. For the developer, buried in bugs, working fixes at one workstation and commenting on specifications for the next product revision on another machine (which is also on an unrealistically short schedule), high profile bugs require more attention than lower-profile ones. So exists a way for the developer to minimize some of the attention a bug receives—fudge on the severity. Make the severity 2 bug a severity 3. Since it's been decided that you're not fixing severity 3 issues, this becomes one less to be concerned with. In a nutshell, both QA and Development teams often seek to set their own agendas by manipulating the severity scale. Bad idea. Why? Aside from the obvious fact that the misclassifications violate the process, at every project meeting for months on end, representatives from QA and Development will spend valuable time trying to sell each other on why the severity should be what it is—QA arguing up the severities and Development arguing them down. Instead, stick to the scale, and hash out the fix precedence another way.

#### A more complex proposal

What better way to clarify the severity scale than to make it more complex. No, really. So I suggest an increase in the complexity of issue classification by stating not only a severity, but by setting a priority as well. While many companies interchange these terms with careless abandon, I'll try to clarify why these are very different things, and why we need to use these terms carefully. As we discussed earlier, severity represents the

[3] Many years ago I learned that the real definition of the word "later" is 'ultimately never.' Nothing in all my years has significantly altered my concept of that word. More on this *later*.

functional characteristic of the bug—how it impacts program operation. The more broken the program is because of a defect, the higher the severity warranted. In the earlier program examples, a kernel panic is unarguably more severe than a flawed wizard or GUI. But what if the whole purpose of releasing this new product version was to introduce a graphical interface? Would that affect the GUI bug's importance? Not according to the definitions we established above (and that most organizations use). This is still a severity 1 bug, which will probably be prioritized behind the severity 0 issues. Changing it to severity zero is just fudging, which we already established is a bad idea. Since this functionality, however, is actually critical to this product release, we could leave the severity at 1 where it belongs, but also describe the issue as *high priority*.

In another, more blatant example, suppose a program's install routine splashes the company logo and product name on screen as a backdrop to the installation process. This is actually common practice for GUI-based installations as it gives the company an opportunity to tell the user about important features, pitch other products, and otherwise distract the user from wondering why the installation is taking so long. In other words, they actually want the customer to watch these screens. For this example, our product is called "SAN Manage Pro" (I've always loved the exoti-names those marketing folks come up with, and this is my contribution to the mystique). As the install begins, a flashy spinning company logo appears on the screen (a spinning logo is always good), and then comes the product name in huge, bold letters—SAN Mange Pro, presenting itself to the user for the rest of the installation. It takes somewhere between 3 seconds and forever before the problem sinks in. If you missed it, go back and re-read what is emblazoned across the screen, because we've got ourselves a typo. That rates nothing more than a 3 on our severity scale. But wait, we don't usually get around to fixing severity 3 issues. So our product goes out the door proclaiming itself as mange? Do you know how many customers are going to see that? And if the rest of the product is as "quality assured" as the splash screen, some customers may think we meant for it to say mange[4] in the first place. If we follow the severity-only model I suppose it's all right to ship it that way (unless somebody fudges again and calls this a severity two). But if we add the priority scale, we get a severity 3 bug that is high priority. The first scale represents only the functional impact. It is the second scale that helps us to define the importance of the issue. Vector them together and you have your fix precedence.

[4] This example, and the related discussion are only semi-fictitious. The product did go out the door proclaiming itself as a skin disease, and customers did notice. In this instance, the bug had been properly entered as severity 3, but this organization had no priority rating system so, like many severity 3 bugs, it was ignored. Ouch!

In our first example, where the system panics and corrupts customer data, the severity rating is zero and the priority rating is high (because it's the data, stupid!). For the GUI problems in the second case, the severity is one. The priority depends on the purpose of the GUI. In our clarification of the example we decided the GUI was an important new feature of this product release, so here the priority is high. In the case of the severity two help menu problem, if you know that your product is intended for the novice user, who will rely heavily on easily accessible documentation, this is also a high priority issue. For the typographical error, originally rated a severity 3 and stored away for all eternity as a de facto 'do not fix' (again, because we never seem to get around to fixing severity 3 issues) we added a high priority to it and changed it to a 'must fix' issue, which is appropriate for this bug.

You may have already noticed that each of our examples is now classified as high priority. In any real project there will no doubt be more than 3 issues, and there will be issues that are indeed high priority, and many that are lower. If you find yourself categorizing every issue as high priority, you should consider reevaluating your definition of what constitutes a bug, and of who your customer is. After doing this, if your issues are all still classified as high priority, your program management is going to have to make some tough decisions—some of which may impact the program's schedule and/or budget.

A note about typos and grammatical errors: many times *topographical* errors and *sentences that are not been worded very good* go unnoticed. When this happens, these errors can be an embarrassing blemish on the product and can give the customer a sense (rightly or not) that the entire product is sub-par. Customers say, "If they can't even spell correctly, I wonder how many functional problems the product has." I've heard that general sentiment in one form or another more times than I can count.

Here is one more example where priority clarifies issue significance, this time slanted in the opposite direction. Like in earlier examples, a defect in the product panics the kernel, causing total system failure and customer data loss. We decided earlier that this was undoubtedly a severity zero issue because the system is *really really* broken, which, in the classic scalar 'severity only' model first described in this chapter, Development management scramble to apply resources to resolve this issue quickly. Even in the more flexible 'severity/priority' model, this issue, on first inspection appears to warrant a high priority as well. After analyzing the circumstances surrounding this defect, however, you determine that there are several flaws with the system configuration. First, a number of important operating system files are found to be corrupt. Second, the system configuration is non-standard—one which might be seen in less than 1% of customer configurations. And finally, in this stress test on the product, a load was placed on the machine that was well in excess of what would normally occur in the field (say 400-500% more). This issue still warrants being rated severity zero; the product undoubtedly has a defect since it clearly cannot tolerate this kind of configuration. But if we consider

the circumstances—excessive system load, a highly unusual configuration, and questionable operating system integrity, we can easily dismiss the urgency of this issue. Before doing so, however, it is critical that you verify the circumstances behind the issue. If the problem can be reproduced without all or some of these exceptional circumstances, you should not reduce the priority, because those circumstances may not be relevant. If you're not sure, it is better to err on the side of caution and keep the priority high. It is much easier to lower a priority after further investigation than it is to raise it—since, once the priority has been lowered, you probably won't be doing as much (or any) investigating of the issue.

After confirming the circumstances, in our example, we could describe this issue's priority as low. It's still a significant issue (severity 0) but due to the likelihood of it happening to the customer, we can set it aside until the more critical issues are resolved. Keep in mind that this is a tricky game and issues like these have been known to come back to haunt companies. You should understand the risks (the topic of risk assessment is well beyond the scope of this article) before assigning priorities.

#### QA is always right

Just ask any QA person and he or she can confirm this assertion is true. Substitute "Development" into this declaration and you'll get Developers to agree with it. (It turns out that the Marketing folks concur with "Marketing is always right," as well—but they're not.[5]) Really though, when developing a product, the Quality Assurance organization is best suited to understand the consequence of a defect. They have the training, the expertise, and the customer focus required to appreciate just what a defect means to the customer—they are in touch with the customer experience. (If your QA organization does not have this type of training and expertise, you should get these quickly! The success of your product depends on it.) This is not to say that developers are clueless about the customer perspective, but that their product focus generally doesn't allow for this kind of comprehension. So, the task of determining defect priority should fall largely on the Quality Assurance organization. They can apply their expertise of the product, from a customer focus, to assign priorities. From there, however, other experts need to be involved.

[5] This is just an unabashedly tacky jab at those hard working Marketing people. So is this.

Developers are experts too, and their expertise lies in an understanding of the product's architecture—how everything is designed to behave. So, when an issue arises, the developer clearly has a better understanding of why the product is behaving the way it does. Because of this understanding, Development carries the primary responsibility of analyzing possible solutions associated with a defect, and of risks related to these solutions. The Quality Assurance team may have valuable input here, and should be part of the discussion, but that value comes largely from the perspective of 'customer advocate.' Development should identify options, assess the risks associated with each, and present this information to Program Management for their consideration. Program Management should own full responsibility for making decisions based on the options presented. It is the Program Manager's role in the process to consider risks to product quality, and to the schedule.

| Component | Control |
|---|---|
| Defect Severity | Defined by process, assigned by QA, modified by QA or Development (after understanding the nature of the defect) |
| Priority | Assigned by QA |
| Risk Assessment | Analyzed by Development, Decided by Program Management |
| Fix Precedence | Prioritized by Development considering Program Management decisions |
| | |

*Table 2. Defect procedural responsibilities*

Let everyone be their own experts, define and adhere to definitions, and allow the processes to work.

---

**Abstract & Author Information**

Excerpted from a book-in-progress, this article explores the concept of defects, provides some background on competing agendas, and provides a model for developing a consistent strategy for classifying and addressing defects.

Kenneth Hass has been involved in development, test and quality assurance for 15 years. His efforts have included computer hardware, software, and consumer electronics products. His current focus is on improving products by integrating the QA process through the entire product lifecycle. He can be reached at labboypro@hotmail.com.