The Bug Life Cycle

Lisa Anderson
Consultant

Brenda Francis
PowerQuest Corp.

Bugs are everywhere! How do you keep track of them all and still make sure the bugs that need fixing get fixed, the fixed bugs really are fixed, and the little bugs that do not make a difference do not crowd the schedule? Read on to discover how the bug life cycle works and how a database, along with a little organization, will make all the difference in the world.

There are a lot of theories presented at testing seminars. There are a lot of *why test* classes, and a lot of classes on specific techniques, but nothing on a couple of practices that can improve the testing process in a company. We are talking specifically about setting up a defect tracking system and enforcing policies and procedures to resolve those defects. Setting up these two things, more than anything else, can put a company on the road to organizing its testing and quality assurance effort. To fill that gap, we have come up with the *Bug Life Cycle*, as shown in Figure 1.

While we cannot claim it as our own, it is what we have learned over the years as testers. Many of you will find it familiar. Anyone who can figure out that the software is not working properly can report a bug. The more people who critique a product, the better it will be. However, here is a short list of people who are expected to report bugs:

- Testers/Quality Assurance Personnel.
- Developers.
- Technical Support.
- Beta Sites.
- End Users.
- Sales and Marketing Staff (especially when interacting with customers).

When do you report a bug? When you find it! Waiting means that you might forget to write it down altogether, or important details about the bug can be forgotten. Writing it now also gives you a *scratch pad* to make notes on as you do more investigation and work on the bug.

Writing the bug when you find it makes that information instantly available to everyone. You do not have to run around the building telling everyone about the bug; it is in the database. Additionally, the information about the bug does not change or get forgotten with every telling of the story.

The easiest way to keep track of defect reports is in a database. Keeping track on paper works, but paper can get lost or destroyed; a database is more reliable and can be backed up on a regular basis.

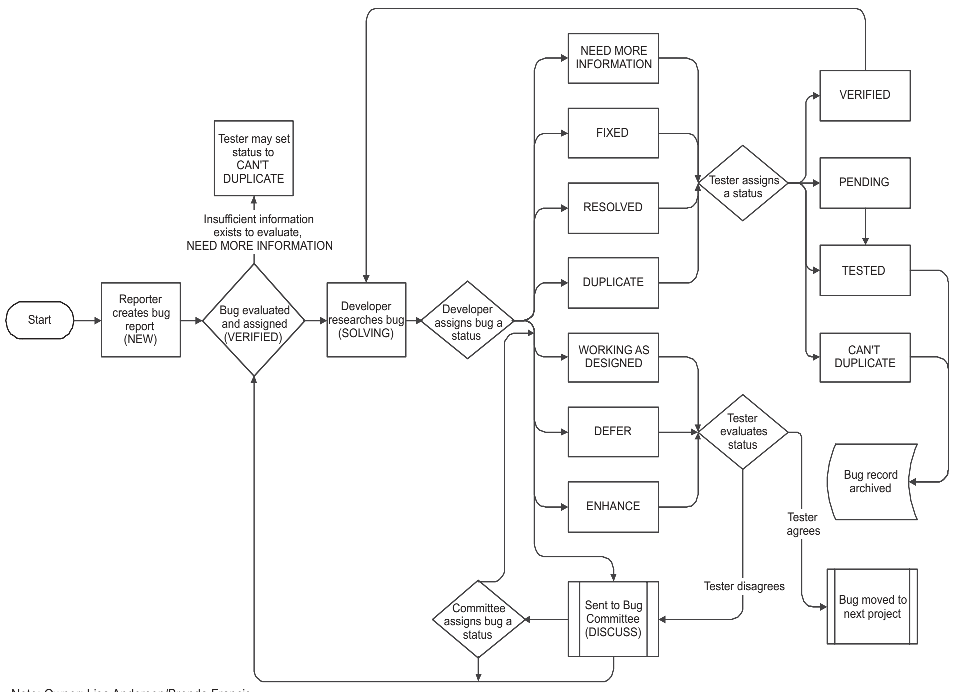
You can purchase many commercially available defect-tracking databases, or you can build your own. It is up to you. We have always built our own with something small like Microsoft Access or SQL Server. It was cheaper to build and maintain it on site than to purchase it. You will have to run the numbers for your situation when you make that decision.

The rule of thumb is one (and only one) defect per report (or record) when writing a bug report. If more than one defect is put into a report, the tendency is to deal with the first problem and forget the rest. Remember that defects are not always fixed at the same time. With one defect per report, as the defects get fixed, they will be tested individually instead of in a group where the chance that a defect is overlooked or forgotten is greater.

A good bug report includes the following items:

- Put the *reporter's name* on the bug. If there are questions, you need to know who originated the report.
- Specify the *build* or *version number* of the code being worked on. Is this the shipping version or a build done in-house for testing and development? Some bugs may only occur in the shipping version; if this is the case, the version number is a crucial piece of information.
- Specify the *feature* or *specification* or *part of the code*. This facilitates getting the bug to the right developer.
- Include a *brief description* of what the problem is. For example, *fatal error when printing landscape* is a good description; it is short and to the point.
- List *details*, including how to duplicate the bug and any other relevant data or clues about the bug. Start with how the computer and software are set up. List each and every step (do not leave anything out). Sometimes a minor detail can make all the difference in duplicating or not duplicating a bug. For example, using the keyboard versus using the mouse may produce very different results when duplicating a bug.
- If the status is not *new* by default, change it to *new*. This is a flag to the bug verifier that a new bug has been created and needs to be verified and assigned.

Figure 1: *The Bug Life Cycle*


Start
Reporter creates bug report (NEW)
Bug evaluated and assigned (VERIFIED)
Insufficient information exists to evaluate, NEED MORE INFORMATION
Tester may set status to CAN'T DUPLICATE
Developer researches bug (SOLVING)
Developer assigns bug a status
NEED MORE INFORMATION
FIXED
RESOLVED
DUPLICATE
WORKING AS DESIGNED
DEFER
ENHANCE
Tester assigns a status
VERIFIED
PENDING
TESTED
CAN'T DUPLICATE
Bug record archived
Tester evaluates status
Tester agrees
Tester disagrees
Bug moved to next project
Sent to Bug Committee (DISCUSS)
Committee assigns bug a status

Note: Owner: Lisa Anderson/Brenda Francis

Rating	Value
Blue Screen/Hang	1
Loss Without a Workaround	2
Loss With a Workaround	3
Inconvenient	4
Enhancement	5

Table 1: *Bug Severity*

Rating	Value
Always	1
Usually	2
Sometimes	3
Rarely	4
Never	5

Table 2: *Bug Likelihood*

Things to Remember

Keep the text of the bug impersonal. Bug reports will be read by a variety of people, including those outside the department and possibly the company. Please do not insult people's ancestors, their employer, the state where they live, or make any other impulsive or insensitive comment. Be careful with humorous remarks; one person's humor is another person's insult. Keep the writing professional.

Be as specific as possible in describing the current state of the bug along with the steps to get into that state. Do not make assumptions that the reader of the bug will be in the same frame of mind as you are. Please do not make people guess where you are or how you got into that situation. Not everyone is thinking along the same lines as you are.

Prioritizing Bugs

While it is important to know how many bugs are in a product, it is even more useful to know how many are severe, ship-stopping bugs compared to the number of inconvenient bugs. To aid in assessing the state of the product and to rank bug fixes, bugs are prioritized. The easiest way to prioritize bugs is to assign each bug a severity rating and a likelihood rating. The bug reporter does this assignment when the bug is created. Each severity and likelihood category has an associated value. The bug's priority is calculated by multiplying the value of the severity and likelihood ratings.

The severity tells the reader how bad the problem is. Or in other words, it tells what the results of the bug are. Table 1 shows a common list for judging the severity of bugs. Sometimes there is disagreement about how bad a bug is.

To determine how likely it is for a bug to occur, put yourself in the *average* user's place. While the tester may encounter this bug every day with every build, if the user is not likely to see it, how bad can the bug be? Table 2 shows a rating of bug likelihood.

Severity x Likelihood = Priority

To compute the priority of a bug, multiply the numeric value given to the severity and the likelihood. Do the math by hand or let your defect tracker do it for you. The trick is to remember that the lower the number, the more severe the bug is. The highest rating is a 25 (5 x 5), the lowest is 1 (1 x 1). The bug with a 1 rating should be fixed first, while the bug with a 25 rating may never get fixed.

This system is just the beginning. A more sophisticated or advanced way of prioritizing bugs would be to weigh the features and add a development risk value. Each feature adds a different value to the product. Some features are more important than others are. In order to weigh the features, consider each feature's contribution to the product, and weigh it accordingly on a scale of one to five.

Development risk encompasses a number of things. How risky is it to fix a specific piece of code? How will this fix affect the rest of the code base? If it is a minor fix but affects most of the files in the code base by forcing a recompile, then it is a serious fix. This kind of fix could force regression testing that could add significant time to the schedule. Many features may depend on this base feature; this would increase the development risk. If the fix is to a help file that does not affect any other files, then it is a minor one and may be of acceptable risk. This seems like a lot of questions, but the answers can help you assign the proper development risk to each feature.

Using these algorithms may cause bug priorities to cluster around certain values. If you notice this is occurring, you can adjust the algorithm accordingly using a *fudge factor* but that is beyond the scope of this article.

A listing of these bugs ordered by rating means the most important ones will be at the top of the list and should be dealt with first. Sorting bugs this way lets management know whether the product is ready to ship or not. Use whatever criteria you select such as, *all bugs with a priority of 10 or less must be fixed*. If the number of these bugs is zero, the product can ship. If there are any severe bugs, then bug fixing must continue.

Other Useful Information

- Who is the bug *assigned to*? Who is going to be responsible for fixing the bug?
- What *platform* was the bug found on (B, Windows, Linux, etc.)? Is the bug specific to one platform or does it occur on all platforms?
- What *product* was the bug found in? This is important if your company is doing multiple products.
- What *company* would be concerned about this bug? If your company is working with multiple companies, this is a good way to track that information.
- Whatever else you want or need to keep track of. Some of these fields can also have value to marketing and sales. It is a useful way to track information about companies and clients.

Now We Have a Bug

At this point, it may be helpful to have access to the bug life-cycle chart and refer to it during the following discussion. Some paths that a bug may take can be confusing; the chart helps simplify the process.

The first step after the bug is created is *verification*. A bug verifier searches the database for all bugs with a *New* status. He duplicates the bug by following the steps listed in the *details* section of the bug. If the bug is reproduced and has all the proper information, the *Assigned To* field is changed to the developer who will be fixing the bug, and the status is changed to *Verified*. If the bug is not written clearly, is missing steps, or cannot be reproduced, it will be sent back to the bug reporter for additional work.

The Assigned To field contains the name of the person responsible for that area of the code. It is important to note that from this point forward, the developer's name stays on the bug. Why? There are usually more developers than there are testers. Developers look at bugs from a standpoint of *what is assigned to me?*. Testers have multiple features to test, which means that testers look at bugs from a standpoint of *what needs to be tested?* Because of the different way testers and developers work, developers sort bugs by the Assigned To field and testers sort bugs by the Status field. Leaving the developer's name on the bug also makes it easier to send the bug back to the developer for more work. The tester simply changes the Status field to Verified, and then automatically goes back to the developer.

The first thing the developer does is give the bug a *Solving* status indicating that he has seen the bug and is aware that it is his responsibility to resolve it. The developer works on the bug and, based on his conclusions, assigns a status to the bug indicating what the next step should be.

Remember, the developer does *not* change the Assigned To field. His name stays on the bug in case the bug has to go back to him; it will make it back to his list. This procedure ensures that bugs do not fall between the cracks. The following paragraphs list statuses that a developer can assign to a bug.

The *Fixed* status indicates that a change was made to the code and will be available in the next build. Testers search the database on a daily basis looking for all Fixed-status bugs. Then the bug reporter or tester assigned to the feature retests the bug, duplicating the original circumstances. If the bug passes, it gets a *Tested* status. If the bug does not pass the test, it is given a Verified status and sent back to the developer with information about the test performed (for example, the build that was used to test the fix). Notice here that since the bug's Assigned To field has retained the developer's name, it is an easy process for the tester to send the bug back by simply changing the status to Verified.

The *Duplicate* status bug is the same as a previously reported bug. Sometimes only the developer or someone looking at the code can tell that the bug is a duplicate; it is not always obvious from the surface. A note referencing the previous bug number is placed on the duplicate bug. A note is also placed on the original bug indicating that a duplicate bug exists. When the original bug is fixed and tested, the duplicate bug will be tested also. If the bug really is a duplicate, when the original bug is fixed the duplicate bug will be fixed as well. If this is the case, both bugs get a Tested status.

If the duplicate is still a bug – while the original bug is working properly – the duplicate bug does not keep its Duplicate status. It gets a Verified status and is sent back to the developer. This is a *fail-safe* built into the bug life cycle. It is a check and balance that prevents legitimate bugs from being swept under the carpet. However, here is a note of warning: Writing lots of duplicate bugs can give a tester a bad reputation. It pays to set time aside daily to read all the new bugs written the previous day to avoid re-reporting bugs.

Resolved means that the problem has been resolved but no code has changed. For example, bugs can be resolved by getting new device drivers or third-party software. Resolved bugs are tested to make sure that the problem really has been resolved with the new situation. If the problem no longer occurs, the bug gets a Tested status. If the Resolved bug still occurs, it is sent back to the developer with a Verified status.

Need More Information indicates that the bug verifier or developer does not have enough information to duplicate or fix the bug; for example, the steps to duplicate the bug may be unclear or incomplete. The developer changes the status to Need More Information and includes a question or comment to the reporter of the bug. This status is a flag to the bug reporter to supply the necessary information or a demonstration of the problem. After updating the bug information in the *Notes* field, the status is put back to Verified so the developer can continue working on the bug. If the bug reporter can no longer duplicate the bug, it is given a *Can't Duplicate* status along with a note indicating the circumstances.

It is important to note that the only person who can put Can't Duplicate on a bug is the person who reported it (or the person testing it). The developer *cannot* use this status; he must put Need More Information on it to give the bug reporter a chance to work on the bug.

This is another example of a *fail-safe* built into the database. It is vital at this stage that the bug be given a second chance. The developer should never give a bug a Can't Duplicate status. The bug reporter needs an opportunity to clarify or add information to the bug or to retire it.

The developer may want to protest the bug: Should it be included in this version of the product, or perhaps not be fixed at all? The status *Discuss* is used to send the bug to the Bug Committee (test manager, development lead, and/or development manager) for further discussion. The developer should be sure to include comments about why the bug is being protested or needs further discussion.

If the developer has examined the bug, the product requirements, and the design documents, and determined that the bug is not a bug, it is *Working as Designed*. In other words, what the product or code is doing is intentional as per the design. Or as someone more aptly pointed out it is *working as coded!*

This bug can go several directions after being assigned this status. If the tester agrees, the status remains and the bug is finished. The bug may be sent to documentation for inclusion in the help files and manual. If the tester disagrees, the bug can be appealed by putting a Discuss status on it to send the bug to the Bug Committee. The tester should include in the notes a reason why, although the developer has given it a *Terminal* status, it should be changed now. The bug may also be sent back to the design committee so that the design can be improved.

Working as Designed is a dangerous status. It is an easy way to hide annoying bugs. It is up to the bug reporter to make sure the bug does not get forgotten. Product managers may also review lists of bugs recently assigned Working as Designed.

The *Enhance* status means that while the suggested change is a great idea, because of technical reasons, time constraints, or other factors, it will not be considered until the next version of the product. This status can be appealed by changing the status to Discuss and adding a note specifying why it should be fixed now.

Defer is almost the same status as Enhancement. This status implies that the cost of fixing the bug is too great given the benefits it could produce. If the fix is a one-liner to one file that does not influence other files, it might be okay to fix the bug. On the other hand, if the fix will cause the rebuild of many files that would force product retesting and there is no time to test the fix before shipping the product, then the fix would be unacceptable and the bug would get a Defer status. To appeal the status, send it back through the process again by putting a Discuss status on it with a note stating why it should be fixed now.

You may see the *Not to be Fixed* status although we do not recommend making this status available for use. There may be extenuating circumstances where a bug will not be fixed because of technology, time constraints, a risk of destabilizing the code, or other factors. A better status to use is Enhance. To appeal the status, send it back through the process again by putting a Discuss status on it with a note saying why it should be fixed now.

This is similar to the Working as Designed status in that its use can be dangerous. Be on the watch for this one. Sometimes developers call this status the *you can't make me* status.

The Tested status is used only by testers on Fixed, Resolved, and Duplicate bugs. This status is an *end-of-the-road* status indicating that the bug has been verified as Fixed; the bug has now reached the end of its life cycle.

The *Pending* status is used only by testers on Fixed bugs when a bug cannot be tested immediately. The tester may be waiting on hardware, device drivers, a build, or additional information necessary to test the bug. When the necessary items have been obtained, the bug status is changed back to Fixed and is tested. It is critical that the bug is tested just as thoroughly as any other bug fix; make sure testing is not skipped.

The Can't Duplicate status is used only by the bug reporter; developers and managers cannot use this status. If a bug is not reproducible by the assigned developer or bug verifier, the bug reporter needs a chance to clarify or add to the bug. There may be a hardware setup or situation, or a particular way of producing a bug that is peculiar to a specific computer or bug reporter and he needs a chance to explain what the circumstances are. Limiting the use of this status to bug reporters prevents bugs from slipping between the cracks and not getting fixed.

It is important to note that before shipping a product, all active bugs must be addressed; that is, all bugs with a Fixed, Need More Information, Resolved, or Pending status must be taken care of. You should also set a criteria based on bug priority; for example, the number of active bugs rated five or less must be zero. These criteria are excellent benchmarks for judging the readiness of a product.

What Happens After Shipping?

All bugs with a Tested or Can't Duplicate status are archived. This means that the records are either removed and placed in an archive database, or are flagged to be hidden from the current database view. Never delete any bug records; it may be necessary to do some historical research in the bug file (*What did we ship when?* or *Why did we ship with this bug?*).

Bugs with Enhance and Defer status are moved to the New bug file or retained in the current bug file. The statuses of these bugs are then changed back to Verified.

This methodology not only shortens the list of bugs to deal with, but it also moves bugs that were not considered necessary for the current product to ship into consideration for the next version of the product.

Reports

The data in the bug file are not very useful until sorted and presented in an organized fashion; they then become information. For example, sorting by developer, the information becomes a *to-do* list sorted by rating. Sorting by status lets the reader know how many bugs are submitted or in progress; sorting by feature asks, "How many open bugs are there for a particular feature?" "What feature needs more work?" and "What feature is stable?" Sorting by product is useful when more than one product is being worked on simultaneously.

Be aware that there are certain metrics or reports that should not be used. If you use these reports you will destroy the credibility of your bug file and it will be reduced to a *laundry list* for developers. One of these reports is "*How many bugs did a tester report?*" and the other is "*How many bugs did a developer fix*?" Neither one of these has any useful purpose except to beat up people uselessly [1].

A defect database that has all these fields built into it and has a good query language is able to sort defect data and turn it into useful information. Setting up customized queries should not be too difficult for the average database administrator.

Conclusion

The challenges of following a bug life cycle are far outweighed by the benefits derived. A well planned and closely managed defect database not only tracks current defects against any number of builds and/or products, it also provides a virtual paper trail for the overall progress of a product as it is coded, tested, and released. If sufficient time is provided for building a defect tracker that works for your company, it is more likely you will release a less buggy product, or at least a product where most of the big ones have *not* gotten away.◆

Reference

1. Kaner, Cem, et. al. Testing Computer Software. 2nd ed. New York: International Thomson Computer Press, 1993.

Additional Reading

1. Beizer, Boris. Software Testing Techniques. 2nd ed. New York: International Thomson Computer Press, 1990.
2. Hetzel, Bill. The Complete Guide to Software Testing. 2nd ed. New York: John Wiley & Sons, Inc., 1988.
3. Jones, Capers. Software Quality: Analysis and Guidelines for Success. New York: International Thomson Computer Press, 1997.
4. Kit, Edward. Software Testing in the Real World. New York: Addison-Wesley, 1995.
5. Mirrer, Barry. "Organize Your Problem Tracking System: Cleaning Up Your Bug Database Can Be as Easy as Organizing Your Sock Drawer." Software Testing Quality Engineering Sept./Oct. 2000: 34-39.
6. Myers, Glenford J. The Art of Software Testing. New York: John Wiley & Sons, Inc., 1979.
7. Patton, Ron. Software Testing. Indianapolis: Sams, 2000.
8. Institute of Electrical and Electronics Engineers, Inc. IEEE Standard for Software Test Documentation 829-1998. New York: Institute of Electrical and Electronics Engineers, Inc., 1998.

About the Authors

Lisa Anderson has been a tester and quality assurance engineer with WordPerfect Corp., Novell, Inc., Corel, Inc., and PowerQuest Corp. Anderson has also been a director of Quality Assurance (QA) for a small startup company and was a QA manager at PowerQuest Corp. She has been in the software QA field since 1991; has attended STAR East 1999, 2000, and 2001; participated in Software Testing Manager Roundtable 4 and 5; and is the sponsor of Mountainwest Enterprise Testing Roundtable 2003. Anderson has bachelor's degrees in education and computer information systems and is currently working on a master's degree in computer information systems.

Phone: (801) 319-5840
E-mail: lisaan@lisaan.com

Brenda Francis is a software quality engineer at PowerQuest Corp. She worked formerly for Novell and WordPerfect in problem resolution teams and has been in the software quality assurance field since 1997. She has a bachelor's degree in international relations and a master's degree in American history.

PowerQuest Corp.
P.O. Box 1911
Orem, UT 84059-1911
Phone: (801) 437-8900
E-mail: brenda.francis@ powerquest.com