RBCS

TIME TESTED.
TESTING IMPROVED.
www.RBCS-US.com

The Defect Life Cycle and the Software Development Life Cycle

Rex Black, President, RBCS, Inc.

Mistakes lead to the introduction of defects (also called bugs). As I personally am aware, like all human beings I can make mistakes at any point in time, no matter what I might be working on. So it is on projects, where the business analyst can put a defect into a requirements specification, a tester can put a defect into a test case, a programmer can put a defect into the code, a technical writer can put a defect into the user guide, and so forth. Any work product can and often will have defects because any worker can and will make mistakes!

Now, I like to drink good wine, and I've collected bottles from around the world on my travels. Some I've stashed away for years, waiting for the right time to drink them, which always comes sooner or later. Some of these bottles have gotten a lot more valuable over the years. Odd as this will sound, in just a few ways, bugs are like wine. They definitely get more expensive with age, taking more effort and thus incurring more cost the longer they are in the system. Also, sooner or later most bugs need to be fixed. However, it's definitely not a good idea to leave bugs lying around—or perhaps crawling around—in a cellar, and they're certainly not appetizing!

Because defects can be introduced anywhere in the life cycle, in any work product, and because the cost of removing them increases the longer they are around, we should try to remove them throughout the life cycle, as close as possible to the point of introduction. When all defects are removed in the same phase in which they were introduced, this is referred to as perfect phase containment. Perfect phase containment minimizes the cost of quality for a system with a given level of quality. Of course, cost of quality can also be reduced through defect prevention, which can be promoted through process improvements.

So, by using static techniques such as reviews and static analysis on work products during the process of creating those work products, we can contain many—in fact, with the right processes, most—of the defects to their exact phase of introduction. This minimizes the damage those defects can do, especially since defects are found directly during static testing, thus avoiding the costly and slow debugging process.

Of course, debugging defects revealed as failures during dynamic testing is more expensive than fixing defects found in static testing, but it's still much cheaper than debugging defects found in production. Removing bugs found in dynamic testing requires a solid process for investigating failures, usually via a defined bug management process.

Let's review the chain of events associated with dynamic test failures a bit more closely. The author of a work product introduces a defect. These defects may be introduced in the code itself, or they can be introduced in requirements specifications, design specifications, user stories, use cases, or other precursor work products, but they eventually end up as defects in the code. (If defects in precursor work products were removed before they ended up in the code, they were perforce removed by static testing because only the code can be subjected to dynamic testing.)

Now, a defect in code is a somewhat passive and shy thing. It is passive in the sense that it will not display symptoms unless someone executes the code in which it exists. It is shy in the sense that it can be seen only through the symptoms of its presence, and often those symptoms can be seen only by executing the code with particular types of inputs and preconditions. Those symptoms take the form of an anomaly, a case where the actual results don't match the expected results. Sometimes, the anomaly is sufficiently subtle—or the tester sufficiently inattentive or the test poorly defined—that the tester misses it, which is referred to as a false negative in ISTQB terminology.

When the tester observes the anomaly, the defect management process can be said to begin. After some efforts to avoid a false positive (again, more on that shortly), having been satisfied that the anomaly is truly a failure— that is, not a false positive but rather an anomaly due to a defect—the tester files a defect report. The defect life cycle begins now.

Before we go further though, here's a riddle for you: When is a bona fide test failure not the result of a defect? Give up? How about now? The answer is, When the failure is the result of an automated unit test used in test-driven development. In test-driven development, the tests are written before the code and are basically executable design specifications, in that the code is written and refined until the tests pass. Usually the process is iterative, in that a few tests are written to start with, then some code is written and revised until the tests pass, and then more tests are written and the cycle resumes. Because the failure of the tests is by design—in other words, it's inherent in the process—the failure is not the result of a defect because a defect is introduced by mistake rather than by the nature of the process. Since these test-driven development failures are not defects, they should not be reported, though there's little risk that they would ever be, given developers' hesitancy to report any defects found in unit testing of any sort.

Defect Workflow

Once surfaced as a failure in dynamic testing, each defect goes through a life cycle from discovery to some sort of ultimate resolution. Without a well-defined workflow, though, it's quite possible for the life cycle of some defects to suffer from unnecessary delays or even to get lost and never actually fixed. Even with a well-defined workflow, there is potentially a lot of overhead associated with managing all the defect reports on a project, so most mature organizations use a defect management tool to automate the defect workflow. Let's look at how to make this work.

First, you need a clearly defined set of states for defect reports from discovery to resolution, along with allowed transitions between those states. This is the defect workflow. Many defect management tools include such a workflow for the defect reports by default, but it's a good idea to use this workflow as a starting point rather than accepting it without question. It may be that the "out of the box" workflow is fine for your organization, but that's not always the case. Any good defect management tool will allow you to customize the workflow to fit your organization's needs. We'll look at an example of a workflow with a more detailed discussion of the states in the workflow later.

Part of understanding the proper workflow for your organization is to understand the roles involved in the workflow. A fairly typical workflow involves the following states and roles:

1. The defect finder, often a tester, reports the defect. The underlying report is in an initial state, which may also be called a new or open state.
2. The defect report is triaged by a cross-functional team, often including representatives of the test team, the development team, project management, and other business and technical stakeholders. Triage involves deciding whether the defect report actually describes a failure at all; if not, the defect report is put into a canceled state or placed into a closed state with the reason that the report was invalid. For actual failures, triage also involves deciding whether to fix the defect at all and, if so, when to fix it. If the defect is not to be fixed in this release, the defect report may be placed in a deferred state, either deferred to a subsequent release or deferred indefinitely. If the defect is not to be fixed at all, then the defect may be placed in an accepted state as a permanent limitation.
3. If the defect is to be fixed, it is placed in an assigned state for a particular fixer, often a developer. In some cases, the fixer needs more information or disagrees about whether the behavior is incorrect. In the former situation, the defect report is placed in a returned (or clarification) state and routed back to the defect finder in step 1; in the latter situation, the defect report is placed in a rejected state and returned to the triage team in step 2. In other cases, the fixer has no problem with the report but simply cannot reproduce the failure and may return the defect report to the triage team. The triage team may place the defect report in an irreproducible state, which might involve no further action on the report until the failure is observed again. Alternatively, the triage team may reassign the defect report to the fixer, returning to the start of this step, or they may reassign the defect report to the finder to gather further information, returning to the start of step 1.
4. If there is no problem (either with the defect report and the failure itself) detected in step 3, the developer fixes the underlying defect. Once the developer believes the defect is fixed, the defect fix is routed to configuration management for inclusion in a test release. The defect report is usually placed in a build state.

5. Once the test release is installed in the test environment, someone—often but not always the original finder of the defect—is assigned to verify the repair of the defect described in the report, which is in a confirmation test (or simply test) state.
6. If the confirmation test passes, the defect report is placed in a closed state. If a new defect is discovered during the confirmation test, a new report is originated and the cycle for that report begins at step 1.
7. If the confirmation test fails, the defect report is placed in a reopened state and returned to step 3. If a new defect is also discovered during the confirmation test, a new report is originated for that defect and the cycle for that report begins at step 1.

Note that, in each state, the report has an owner who is responsible for taking action to move that report into an appropriate next state, unless the report is in a terminal state. The terminal states mentioned in the syllabus are closed, canceled, irreproducible, or deferred. I added the terminal state of accepted in the preceding list because that is also a possibility.

Notice that the finder, who is often a tester during dynamic testing, owns the defect report at the beginning and the end of the workflow. Let's look more closely at what the tester must do during these states.

Tester Defect Report States

In the workflow outlined previously, the finder of the defect—who I'll refer to as the tester for convenience and familiarity for now—had certain states in which actions are—or may be—required.

In the initial state, the tester needs to collect information about the failure and the underlying defect. There are limits as to how much information is possible and practical to gather, but in general, the more information collected in the report, the easier it is for the fixer to reproduce the failure and repair the underlying defect.

In the returned state, the fixer either has asserted that no problem exists or wants the tester to provide more information. If possible, the tester must either substantiate their assertion of a problem or gather additional information about the problem. The test manager should run regular reports to check the number of defect reports that enter this state because it's a potential source of inefficiency and delay in the entire defect workflow. My usual rule of thumb is that no more than 5 percent of defect reports should be returned; if the number is higher than 5 percent, then a problem exists that the test manager should address.

In the confirmation test state, the tester should repeat the steps to reproduce the failure from the defect report. In some cases, the test strategy will call for a complete repeat of the test or tests that originally identified the failure; while this is the most risk-averse approach, it is more expensive than just repeating the steps to reproduce the failure. As noted, the confirmation test may pass or fail, which results in different actions, and in addition a new failure might be observed as part of that confirmation test.

Whenever defect reports are owned by testers, the test manager should make sure that prompt and proper actions occur. It's a common problem for test managers to lose track of which reports their testers must act upon. Such problems can cause delays and inefficiencies in the defect workflow and in some cases reputational damage for the test team.

Invalid and Duplicate Defect Reports

As much as the test team might try to avoid it, some defect reports are invalid in that the anomaly does not result from a defect. Such invalid defect reports are referred to as false positives in the ISTQB nomenclature. A false positive can occur when the test environment is set up incorrectly. It can also occur when the test data is wrong or just improperly loaded. It can occur when there are problems with the test steps, the test inputs, or the test's expected results or when the automated test script is wrong. It can also occur when the tester is mistaken in their expectations of the proper results or behavior.

Test teams should also try to avoid filing duplicate defect reports. Duplicate defect reports exist when two or more reports describe behavior that is due to a single underlying defect. This can happen because multiple testers detected a related failure at the same time; either the testers might have not communicated about the failure or the symptoms might have been different enough that they thought different defects were in play. It can also happen because the number of active defect reports becomes so large that it is impossible for testers to keep track of what has already been reported. When duplicate reports are detected, the best practice is to keep the better of the reports open as the main report and to close the other report or reports as duplicates (while retaining a link to the active report to help searches). Duplicate reports should not be canceled or closed as invalid because the problem described is real.

As I said earlier, we should try to avoid invalid and duplicate defect reports. Inefficiency is invariably associated with such reports, due to the extra effort associated with managing them through their workflow from discovery to resolution. In large numbers, the associated effort can be significant. However, the test manager's dilemma is that pressure put on testers to eliminate such reports will result in hesitancy by testers to report defects at all. This leads to an increase in the number of real defects that are not reported at all, which drags down the test team's defect detection effectiveness. Since defect detection is a central objective for most test teams, such a problem is worse than the usually minor inefficiency caused by a small percentage of invalid and duplicate defect reports. As a general rule, I feel that as long as the number of invalid and duplicate defect reports is no more than 5 percent in each category, the situation is acceptable.

Cross-Functional Defect Management

The bug triage team, also called the defect management committee, is a cross-functional team that includes representatives from various business and technical stakeholder groups, such as development, project management, product management, and other stakeholders. The test manager might be the moderator of the meeting, or the moderator might be a project manager or sponsor. I have seen many variations in terms of leadership and composition of these teams.

The triage team should meet regularly during periods of active testing, when defect reports are being entered into the defect management tool. This might include periods when reviews are happening, if defect reports from those reviews are being entered into the defect management tool, but my experience is that more frequently defects identified in reviews are managed by the review team rather than by a defect triage committee.

As I mentioned earlier when I outlined the defect management process, the triage committee decides whether the defect report describes a failure. If the defect report doesn't, the report is canceled or closed as invalid. If the defect report does describe a failure, the triage committee decides whether to fix the defect or to defer it. There are benefits of fixing defects, of course, such as increased quality, but there are also costs, because resources are always limited, and there are risks associated with changing code to fix a bug. If the costs and risks outweigh the benefits temporarily, then repair of the defect should be deferred until later in the project, until the next project, or until some yet-to-be-determined project in the future. If the costs and risks outweigh the benefits permanently, then the defect may be deferred in perpetuity, accepted as a permanent limitation on the product.

If the defect is to be fixed, then the triage committee must consider priority. Many other project tasks are going on during test execution, including fixing defects that were previously discovered. While you as the test manager might want every bug your team finds fixed, and fixed immediately, that is not possible on most projects. Some bugs will be deferred, and there will be some delay associated with the repair of all but the most critical bugs. It behooves you to be realistic in your approach to and participation in the bug triage committee. Remember that the best possible project outcome must be achieved within the constraints of the project, including limits on the number of bugs that can be fixed. You—and perhaps other testers—should participate constructively, give good (not strident or angry) advice, and provide the information the bug triage team needs to make smart decisions.

The goal of defect management is to effectively and efficiently manage the known quality problems on a project up to the point of product release. No single element of defect management by itself can achieve this goal, but all elements—good communication, good defect management tools, and a proper defect workflow, guided by a professional, thorough, and committed defect management committee—must work together to reach it.

Example: IEEE 1044 Process

We've looked at the defect workflow and the participants in it. However, we haven't looked at the main force behind what moves defect reports through the workflow or what is happening to the participants as defect reports move through the workflow. The answer is that each owner, or the triage team collectively, learns more about the defect and what it means. This knowledge drives the decisions that are made. To be effectively utilized, the knowledge gained must be captured. Let's look at an example of how that knowledge can be captured throughout the defect workflow, using IEEE 1044 as an example.

Table 01 shows how the gathering of classifications and data for defect reports works throughout the defect life cycle within the IEEE 1044 standard. In each step—and indeed, embedded in each state—are three information capture activities:

- Recording
- Classifying
- Identifying impact

	Activities		
Step	*Record…*	*Classify…*	*Identify impact…*
1. Recognition	Include supporting data	Based on important attributes	Based on perceived impact
2. Investigation	Update and add supporting data	Update and add classification on important attributes	Update based on investigation
3. Action	Add data based on action taken	Add data based on the action taken	Update based on action
4. Disposition	Add data based on disposition	Based on disposition	Update based on disposition

Table 01 Example: IEEE 1044 process

During the recognition step, we will record supporting data. We will classify based on important attributes that we have observed. We will identify impact based on perceived impact, which might differ from the final impact assessment.

During the investigation step, we will update and record more supporting data. We will update and add classification information on importance based on attributes uncovered during the investigation. We will update the impact based on investigation too.

During the action step, we will record new supporting data based on the action taken. We will also add classification data based on the action taken. We will update the impact based on the action too.

Finally, during the disposition step, we will record final data based on the disposition. The classifications will be adjusted and finalized based on the disposition. The final impact assessment will be captured.

Notice that I've been talking about data and classifications. The IEEE 1044 standard includes mandatory and optional supporting data and classifications for each activity in each step. When I say "mandatory supporting data and classifications," I mean mandatory for IEEE standards compliance.

Each of these data items and classifications is associated with a step or activity. The IEEE has assigned a two-character code in the standard: RR (recognition), IV (investigation), AC (action), IM (impact identification), and DP (disposition).

Example: IEEE 1044 Life Cycle

In Figure 1, you see a diagram that shows the IEEE 1044 incident management life cycle, including a mapping from IEEE 1044 that shows how typical incident report states in an incident tracking system would fit into this life cycle.

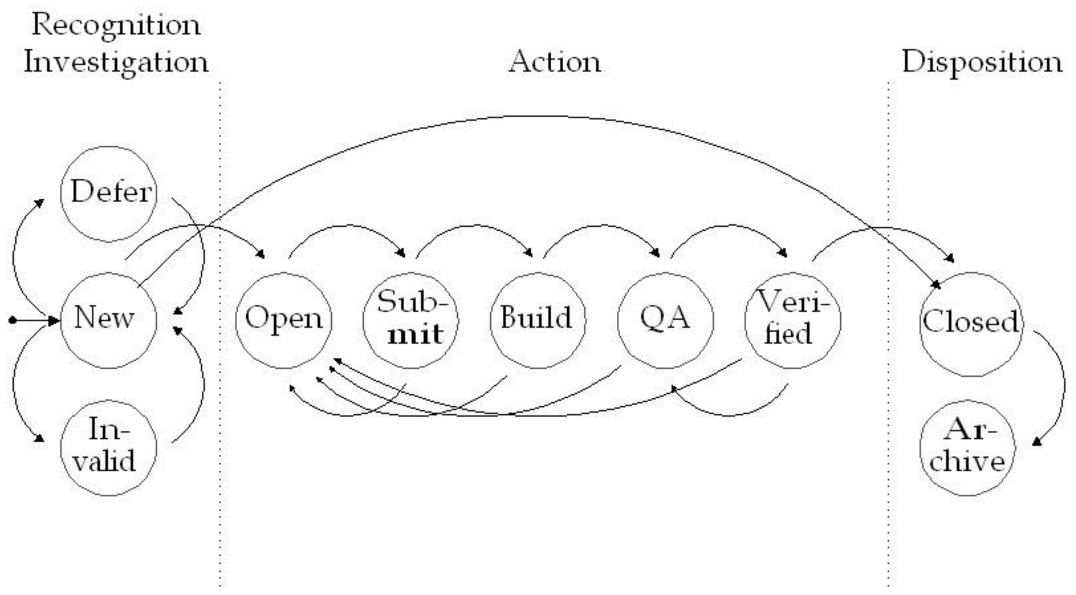

Figure 1 IEEE 1044 incident management life cycle

We assume that all incidents will follow some sequence of states in their life cycle, from initial recognition to ultimate disposition. Not all incidents will travel through the exact same sequence of states, as you can see from the figure. The IEEE 1044 defect life cycle consists of four steps:

Step 1: Recognition. Recognition occurs when we observe an anomaly, that observation being an incident. If the cause of the anomaly is a defect, then this incident is a failure. This can occur in any phase of the software life cycle.

Step 2: Investigation. After recognition, investigation of the incident occurs. Investigation can reveal related issues. Investigation can propose solutions. One solution is to conclude that the incident does not arise from an actual defect; for example, it might be a problem in the test data.

Step 3: Action. The results of the investigation trigger the action step. We might decide to resolve the defect. We might want to take action to prevent future similar defects. If the defect is resolved, regression testing and confirmation testing must occur. Any tests that were blocked by the defect can now progress.

Step 4: Disposition. With action concluded, the incident moves to the disposition step. Here we are principally interested in capturing further information and moving the incident into a terminal state.

Example: Defect Life Cycle

In Figure 1, you see the life cycle used on a defect tracking system we implemented for the IVR system of systems project. We created this defect tracking application for the client, using Microsoft Access. This was before widely available freeware defect tracking systems like Bugzilla became available, so we felt this was a cost-effective option.

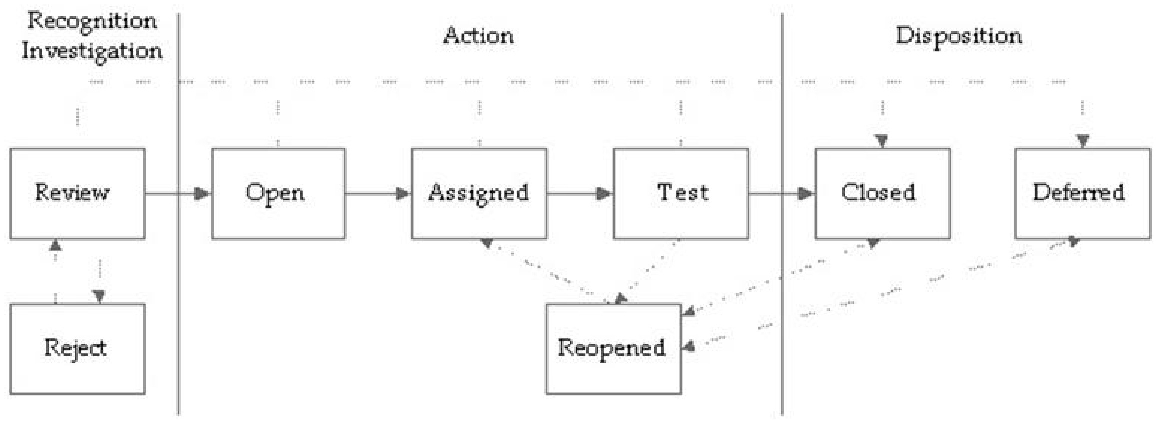

Figure 1 Example: Defect life cycle

If you refer back to Figure 1, you can see that the states and step mappings are quite similar. There are a few differences.

For one thing, we considered deferred a final disposition. We did not have an invalid state, but we would close a report as invalid if we decided to permanently reject it. That is also a final disposition.

In addition, we used a single combined assigned state rather than separate submit and build states because we didn't track build time separate from fix time. We used a single combined test state rather than separate QA and verified states because my test team was empowered to determine if the problem was fixed.

Finally, we did not have an archive state because we assumed the reports would remain indefinitely in their final state.