

A Full Life Cycle Defect Process Model That Supports Defect Tracking, Software Product Cycles, And Test Iterations

Jim Nindel-Edwards

jimne@ieee.org

Gerhard Steinke

Seattle Pacific University

gsteinke@spu.edu

ABSTRACT

There are a variety of models, methods and tools to help organizations manage defects found in the development of software. Defect tracking and processing must be integrated in the project life cycle and the testing process for software. This paper reviews a number of defect models and proposes the Full Life Cycle Defect Process model to manage defects that supports defect, project, and test processes. We describe the various states in our model and provide examples of various scenarios and paths through the model.

INTRODUCTION

Software development projects in major corporations are more frequently using a defect management process to decide what is to become of software defects, or bugs, found in the development cycle. Sometimes these processes are simply termed the “triage process” after the key activity in the larger process where management or project leads makes the decision on whether a defect is to be fixed or not. Yet, the defect process is larger than the triage process. Briefly stated the defect process starts with the discovery of a product defect and normally ends with the defect either corrected, or a decision made that the defect is not to be corrected at this time.

The defect process typically works within a larger process we will simply refer to as the project life cycle, which can take many forms ranging from a full waterfall project model to Agile or XP (Extreme Programming). The intersection of the defect process with the project life cycle is best seen with a defect that cannot be corrected in the current project cycle, but can, should, and will be corrected before the final software release.

Likewise the defect process also operates within another process, the test process for a software product (Humphrey, 1989). While typically closely matching the project process, the test process can and often does iterate more quickly than the project itself. For example, before an alpha release of a product the software application may go through several test passes with a final pre-alpha test pass just before release to alpha users for the product. The intersection of the defect process with the test process can be seen in two areas. First, there are defects that may be resolved as duplicate to other defects already recorded. And secondly, verifying that defects that have been corrected in a prior test pass and are still functioning as expected on subsequent test passes – a type of regression test.

As the defect process often is the most formal of the processes used in the development life cycle there are occasions where the defect process will be used to drive project, test, or both processes at the intersections points. Our intent is to draw clear focus on these intersection points and to propose a Full Life Cycle Defect (FLCD) process that supports defect, project, and test processes that complement all of the later activities.

TYPICAL INDUSTRY DEFECT MODELS

Many if not most companies that develop and maintain software applications have defect models in use (Black, 1999, Rahman, 2004). While there are wide variations in implementation, manual vs. automated, internal tools vs. external tools, they typically have many common features.

The simplest form for the defect model itself represents two states with three state changes: defect discovery (state change), repair or other resolution (state), and verification (state) of the resolution. Figure 1 represents this simple two state model.

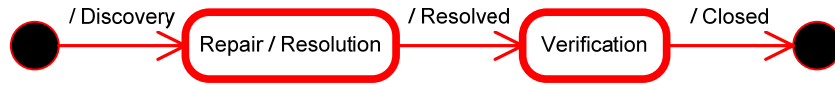


Figure 1: Two State Defect Model

In practice this model typically is expanded into a three state model typified in Figure 2 below. For example, the prevalent defect, or bug process as it is known locally at Microsoft (Kipman, 2006) has three basic states:

1. Submitted/Open
2. Resolved
3. Closed

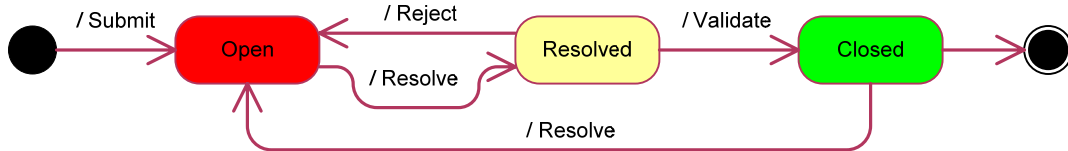


Figure 2: Three State Defect Model

The additional process state, that of “closed”, reflects that defects when resolved and verified are in fact “closed” as an activity, but may in fact be re-opened at a later time – a prelude to our discussion of postponed items. The simple three state model is the baseline for many, if not most of the reported defects for a software system.

The recently released Microsoft Team Systems, a workgroup addition to the Visual Studio 2005 development suite, uses a four state defect track (or “bug” workflow) for CMMI Process Improvement projects (Microsoft, 2005). (See figure 3).

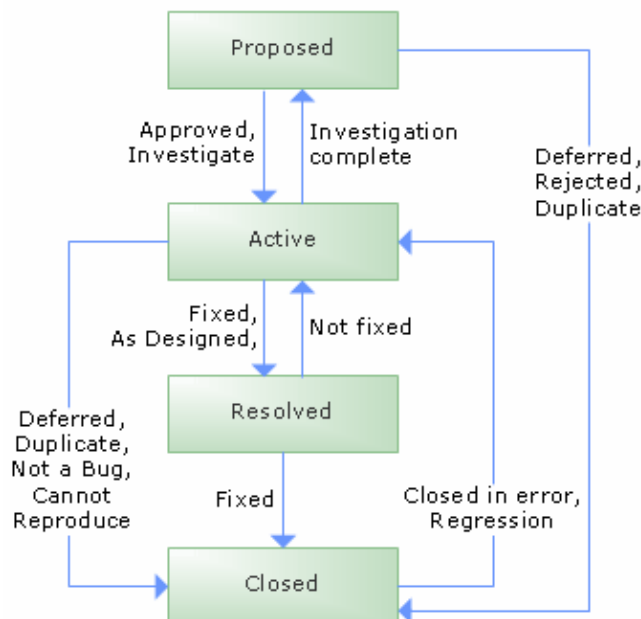


Figure 3: Microsoft CMMI Project Bug Work Item Flow

In this model you see the “open” state broken into “proposed” and “active” states with the “resolved” and “closed” states matching those of figure 2.

IBM/Rational markets a very robust process tracking and support tool, ClearQuest and its “out of the box” state model (IBM, 2005) appears below, figure 4. The IBM/Rational process contributes the “submitted” state where newly discovered defects are examined for the first time. In the Microsoft model this “triage” process occurs as activities in the “open” or “proposed” state. Other process models such as the IBM/Rational model are more formal in the treatment of defect triage. Through the use of various status fields they could include various sub-states, for example:

- Submitted (new) vs. Reopened
- Open/Active vs. Open/Postponed
- Close/Fixed vs. Close/Won’t fix

ClearQuest Deployment Kit
Defect Usage Model

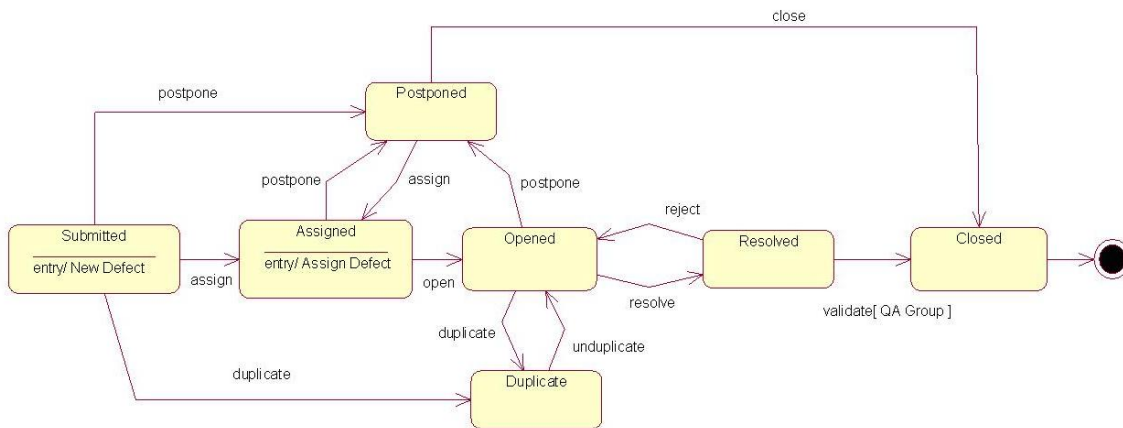


Figure 4: ClearQuest Defect Usage Model

The ClearQuest system is highly adaptable for a variety of processes and its “out the box” implementation is a useful starting point for any team that desires a high control level over the defect process whether they have an existing defect management process or are starting from no defined defect process.

These two process flows, Microsoft Team Foundation Server and IBM/Rational, are simply representative of typical process flows supported by defect tracking systems. There are many defect tracking systems available—over 100 packages are currently listed at the StickyMinds.com website (Sticky Minds, 2006). Many of the packages support workflow customization just as both the Microsoft and IBM/Rational tools support customized defect states, state transitions, etc. The underlying defect process described here is however tool agnostic as its implementation can be supported in a wide variety of available defect tracking tools.

PROCESS DEFINITIONS – DEFECT, PROJECT, TEST

We will start the discussion with some elaboration on the whole of the defect process and its interface with the project and test processes.

Defect Process

The defect process starts when a defect (Jacobson, Booch, Rumbaugh, 1999) is discovered and ends when the defect is resolved, hopefully repaired, for the most immediate release of the software application. The driver here is the discovery of the defect. The conclusion is the determination that the defect has been somehow resolved.

Project Process

The project process starts with the normal project initiation activities and ultimately ends with the release of the software application. Within this process there will typically be phases or iterations with each phase building on the prior phase. For the purposes of interfacing the project process with the defect process it is the project phases that are most important here. The driver therefore is the application phase start and phase end using whatever criteria is applicable for the particular application life cycle model for phase start and phase end.

This process starts as most projects do with a business need or market opportunity and typically end with a completed product ready for the user to install and put to productive use. Within a project process there typically will be many phases or iterations (De Man, Ebert, 2003) depending on the development methodology used by the product development team. Without delving too deeply into project methodology suffice it to say that project phases and/or iterations introduce the possibility that a defect can be postponed for later action. A simple example might be a reported defect of a spelling error in a report title may be validly postponed in a project where all report titles are subject to change and scheduled for comprehensive review later in the project.

Test Process

In Quality Assurance terms this is the Quality Control (or QC) process where software testers inspect the project deliverables to assure that the product performs to specifications and does not exhibit any undesirable side effects. The first part of the test process then is the independent verification that any corrected defect is in fact repaired in the appropriate new release of the software. After verification that a defect is not longer present the defect may be closed. If however the defect still is present the defect must be referred back for further consideration.

The test process typically starts when the development effort has produced something “testable” and ends with the application meeting its release criteria for the project phase. Within the project phase the test process may iterate more than once and ideally the phase iteration will be documented in the test plan. For example, in phase 2 of a project there could be a functional, security, and performance test pass each producing a number of relevant application defects that need to be reviewed and hopefully corrected.

The final intersection of all three processes, defect, project, and test, is whether a defect that has been identified as being corrected (typically closed) must be re-validated in subsequent project phases or iterations. One approach to this may be to require defects to be verified in subsequent project phases/iterations. Another approach might be to add to the product test plans specific test scenarios and/or test cases based on previous defects and include appropriate re-testing as part of future test plans, which are typically linked to the project plan.

A FULL LIFE CYCLE DEFECT MANAGEMENT PROCESS

A comprehensive defect model needs to account for both postponed and duplicate defects, as well as have a defect lifecycle perspective. We propose the model shown in figure 5, the Full Product Life Cycle Defect (FPLC) model, which is an extension of the “out of the box” IBM/Rational model with changes to include the test and project management interfaces.

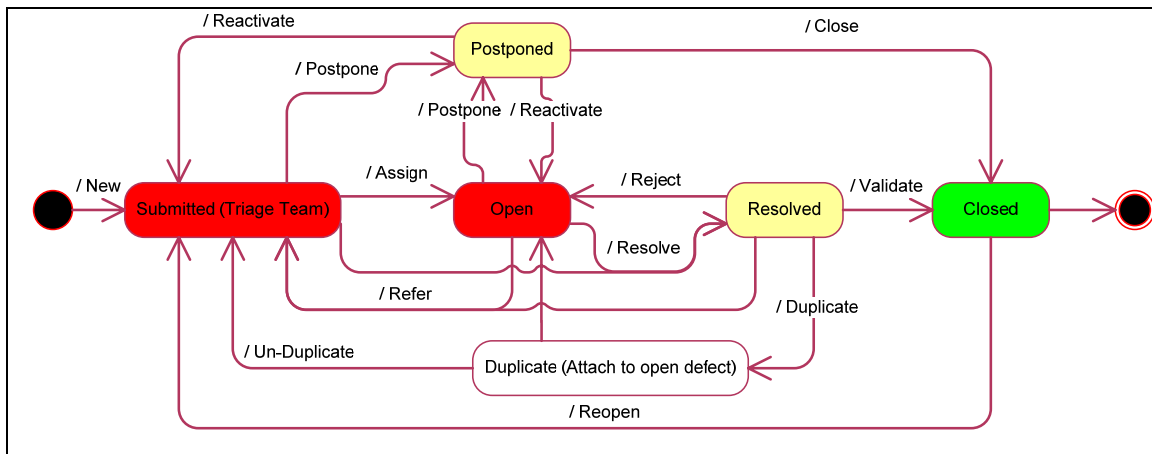


Figure 5: Full Product Life Cycle Defect Model

We now describe our Full Life Cycle defect model.

Submitted State

The submitted state is the initial state of all defects, a holding pattern waiting for the triage team (Black, 1999) to review, evaluate, prioritize, and assign defects for action. The submitted state therefore is somewhat context sensitive in that the triage team always needs to know where the defect came from. New defects are simply that, newly reported defects. Yet even these carry the possibility that these are duplicate defect reports – ones that the triage team has already considered from a different source.

Beyond the “new” defects the submitted state also considers defects that have either been reopened (typically postponed defects) or referred back for additional triage review. In practice this is clearly evident from the defect report from both seeing the prior state of the defect and review of the note log for the defect. Truly new defects will have no history. Defects that are re-opened, referred, unduplicated, etc. will have a history and review of both the defect history with the current entries in the defect log will determine the next step in the defect’s life cycle.

The essential activity in the submitted state is for the triage team – project managers and lead – to decide what defects will be worked on in the next time period (as determined by the frequency of the triage meetings), assign out for evaluation defects where the affect, scope, or work involved is not clear, and postpone for the short term or longer term defects perceived to be of lower priority or due to the size and nature of the repairs cannot fit into the current development cycle.

Disposition and state movement from the submitted state would be:

- To Open for defects that are to be repaired and released in the current product / test cycle or that need more research and recommendation for repairs.
- To Postponed for defects that are determined not to be fixed in the current product / test cycle but should be addressed before the final gold candidate builds.
- To Resolved (as “won’t fix”) when decided that a particular defect will not be fixed in the current product release cycle and must therefore be released to production with the defect included. Typically it will be up to the triage team in the next product release cycle to re-evaluate and decide which of the “Close”, “Won’t Fix”, “Postponed” items will be included in the next product release.

Open State

Normal entry into the Open state is from the Submitted state. Open simply means the defect has been approved for work and is being resolved. While in the Open state the only common sub-state is “open, but not being worked on” either due to the assigned resource not picking up the defect for work, or progress on the defect resolution is stalled due to a road block or other resource constraint. The best the defect process here can provide is sufficient status information (through notes in the defect log) reflecting the current status of a defect, or nature of the blockage if any.

One typical automated method of detecting stalled defect progress is to report to the project team any defect that has not received any updates or state changes in 3 days.

The other typical entry into Open is a defect that having been resolved is determined not to have been successfully resolved. The Reject action results in the return to the Open state with additional information provided to help explain why the defect resolution was found to be lacking.

Exit from Open typically is to the Resolved state, not that the defect is actual fixed here – a common misunderstanding of defect processes. A whole range of possible “resolutions” are possible but the commonly used resolutions are:

- Resolve - - actually fixed. Defect status fields here should reflect when the fix is actually available for test and/or other review. For example if the fix is in a particular build of a software package, its release number should be identified. If the fix is a documentation update, then the planned document version and release date should be noted.
- Won't fix – The defect either will not, or cannot be repaired at this time. Exit here could be to QA/Test (Resolved) or back to triage (Refer) depending on the reason for a “won't fix” resolution. If the defect is simply an erroneous defect report the “won't fix” is simply that – there is nothing to fix. If the fix to the defect is too significant to be considered at this time a referral back to submitted (potentially postponed) typically is warranted.
- Duplicate – Reflects that the developer believes this is a duplicate of another defect report (and identifies the linkage to the master defect) and indicates that this particular defect will be resolved with the repair of the master defect.
- Postponed – That the defect repair is to be postponed to a future time (see the below discussion of defect postponement).
- Unable to Reproduce – The developer is unable to resolve the problem either because they cannot reproduce the reported problem or in some instances, because the defect has already been repaired. In this event no repair of the defect is either possible or required as the situation where the defect was first reported may not be itself reproducible (Beniaminy, Joseph, 2002).

Resolved State

The Resolved state is the pathway to defect closure where the QA or test team will compare the reported defect with the resolution, test to verify the appropriateness of the repair and/or assess the impact of the non-repair resolution. In some cases the defect repair can cause other problems potentially opening additional defects. The straightforward path through the Resolved state is an appropriate fix to a reported defect that tests “OK” in which case the defect is normally closed. But there are a variety of other possibilities also.

In the event that a defect has been closed as “won't fix” the QA/tester assigned to the problem may choose to escalate the issue back to the triage team (return to Submit state) with some discussion of the impact of the “won't fix” resolution to the product, or users of the product. Resolutions of cannot reproduce (“No repro”) typically call for additional work by the QA/Test team, etc. Looking at the various resolutions discussed above, the initial state of the activities in the Resolved state would appear as follows:

- Fixed – Verify the repair in the appropriate build and either close the defect or flag it for additional testing at a later time (further discussion below).
- Won't fix:
 1. If the reason for the “Won't Fix” reflect that the defect itself reported an erroneous problem the defect itself may be closed without further action. While not particularly common a defect report may in fact reflect a misunderstanding of the application and can in fact be in error.
 2. If the reason for the “Won't fix” reflects a developer decision that the error is not likely to occur, or impact on the larger system is minimal, it becomes incumbent on the QA/test analyst to describe why the defect is worthy of attention and repair. This is assuming the QA role here – representing the user of the application – is required to make the persuasive argument for actually repairing the application. Disposition of the “won't fix” defect could include:
 - Reject – back to the Open state with additional information on why the defect should be repaired.

- Refer – back to the Submitted state for management and lead review of both the problem, priority, severity, and argument for repair of the defect
- Closure of the defect, if in fact the defect is so minor as to not warrant further project attention.

Failure to argue for some actual repair of a defect can be cause to recommend postponement for the short term, or longer term, a “won’t fix” defect if it really should be repaired at some time in the future. Depending on the accepted process flow in an organization the QA/test team may have the responsibility to make such a recommendation or, more commonly, it is referred back to the triage team (Submitted state) with the recommendation that a defect be postponed rather than Closed/Won’t fix.

- Duplicate – If the defect is resolved as a duplicate there is a three step process that should be followed:
 1. Verify that the defect *in fact* is a duplicate of the identified master defect.
 2. Decide whether for testing purposes the repaired defect (that is in fact a duplicate of another defect) needs to be independently tested.
 3. Verify the repaired resolution of the defect in accordance to the resolution of the associated master defect.

If the defect is not in fact a duplicate the defect should simply be sent back to the development team for additional action along with the explanation of why the Duplicate resolution is not in fact correct. The final determination of a defect correctly resolved as a duplicate depends entirely on what it might take to verify the resolution. The trivial example (mentioned above) of a misspelling of a report title may be appropriate flagged as duplicate. The probability that one needs to verify the correct report title on all reports is so small that it does not warrant individual report by report verification. If the problem is more significant the technical resolution may in fact still be duplicate, but the requirement for individual defect by defect verification may cause the movement to the defect state to be skipped. Even though flagged as duplicate for both process and metrics the requirement for individual verification can, and in some cases should, require that the defects be individually resolved (and closed) rather than be closed en masse with the master defect.

Duplicate State

Defects that arrive in the duplicate state are fairly benign. Having passed through several examinations before arriving in the duplicate state these defects simply are a group of defects each linked with an associated “master defect” waiting for the master defect to be resolved by the development team. Normal defect process would allow for mass closure of replication defects on successful validation of the master defect’s resolution.

The other exit from the Duplicate state – a rare occurrence hopefully – would be to re-activate a duplicate in the event that additional information is discovered / provided that a particular defect report is not in fact a duplicate. In this scenario the defect should simply be re-activated and returned to the Submitted state for triage to assign the priority for the defect. Having returned to the triage state a re-activated duplicate defect simply follows the regular defect process as described above.

Postponed State

The postponed state is likely one of the more problematic areas in defect management. Placing any defect in a postponed state is a tacit admission that it should be repaired, but at a later time. Every postponed defect is a negative report on the larger product quality.

The recommendation here is to use the postponed state strictly for those defects that will be, or should be, repaired within a product cycle (i.e. postpone integration defects to the alpha release, alpha reported problem to the Beta release, etc.) In this way the number of defects that appear in the Duplicate state helps product managers and leads to understand just how many defects must be repaired to meet product release quality standards.

Defects that do not have a significant impact are often given the Closed (moved to the closed state) with resolution of “Postponed”. These defects therefore will not be counted against the product quality for the current release, but

do reflect on the larger product quality (inter release quality). When the next release cycle starts up (i.e. V3 after a V2 release) the product manager can look at the inventory of closed as postponed defects and decide which should be re-opened (Davis, 2003) and included in the next release cycle. The long term goal here being to reduce on a release by release cycle both the postponed defect within the release (integration, alpha, beta) and between release cycles (V2 problems postponed to V3). When both postponed counts diminish the overall process and product quality can clearly be shown to be improving.

Another advantage of using both the “Postponed” and “Closed with resolution of Postponed” in a defect management process is metrics that show both within and between release cycle quality measures. Use of one or the other methods of postponing defects would tend to either prevent or make much more difficult this dual look at product and process quality.

Closed State

The last state of a defect is simply the Closed state and for the vast preponderance of defects this is simply good enough. The bulk of project metrics for projects and releases will come from the closed defects and it is both valuable and important that these be properly characterized when closed. Since it is typically the QA/Test team that performs the actual closure of a defect, the final check that the defect is in fact correctly identified (when discovered, how discovered, how resolved, duplicate, postponed, etc) clearly is the responsibility of the QA/Test team and will be reflected in long term organization and process metrics.

The ancillary discussion here is the treatment of the “Closed, postponed” defect largely covered above. In the event a defect is closed as postponed and at later review is determined that it will never be reactivated (perhaps the associated feature set of the application has been retired) the defect should be re-closed with a different resolution (i.e. won’t fix) so as to properly reflect the long term product quality. A postponed defect that is irrelevant should not be considered a reduction in overall product quality.

TYPICAL DEFECT SCENARIOS

Since there are a large variety of common defect paths a walkthrough of some of the common paths will aid in the understanding of how the defect model actually works, and how our defect process can help to facilitate rapid defect resolution while giving good visibility of defects to the project and test management teams. Common defects paths are described in this section.

Normal Defect/Repair Process

The normal “happy path” defect flow (defect scenario #1) would be described in the following steps:

Step	Activity	State Transition	From State	State/To State
1	Initial discovery of the defect. Recording of the reproduction steps, effect on the product/user, assignment of initial severity	New	(new)/Submitted	
2	Triage review of the defect, assignment of work priority and passed to the development team for action	Assign	Submitted/Open	
3	Defect repaired, packaged into next available build. Assign to test team for verification	Resolve	Open/Resolved	
4	Installation of applicable release, verification that the defect has indeed been corrected. Record results and close defect	Validate	Resolved/Closed	

Ideally all defects would follow this most direct of paths but the model must allow for at least three re-work loops: 1) defect could not be reproduced, 2) defect was not corrected, and 3) defect cannot be easily fixed in scope of the current project phase.

Irreproducible Defects

In the case of a defect that could not be reproduced (Black, 1999) the resolution then in step 3 would be “**Not Reproducible**” (or the shorter “**No Repr**”) and the defect would be passed onto test for verification. Several explanations for this are possible but a simple one might be that the defect reported in fact had already been repaired though some other development activity. Another possibility could be that the reported problem in fact was not reproducible, perhaps a defective defect report. In either case the typical verification steps here would be for the assigned tester to verify that the reported problem in fact does not exist in the current product release, in this case verifying that the *resolution* of not reproducible is in fact correct.

Defects That Must Be Postponed

Let us consider the scenario where the defect either cannot be repaired at this time in the project life cycle, or has been determined to be of sufficiently low priority as to not warrant correction. In this case we will assert that postponement of a defect repair is the responsibility of the project leadership team and typically will occur when the triage process occurs (step 2 above), as in the following scenario:

Step	Activity	State Transition	From State	State/To
1	Initial discovery of the defect. Recording of the reproduction steps, effect on the product/user, assignment of initial severity	Submit	(New)/Submitted	
2	Triage review of the defect, determination is made that the defect is not to be addressed at this time.	Postpone	Submitted/Postponed	

Three variations on a postponed defect may be considered at this time, all reflecting various paths through the defect correction process and calling on additional resources in the project team to help better understand the nature of the defect and possibility of repair, as follows:

- a) Request addition information / options from development team

Step	Activity	State Transition	From State	State/To
1	Initial discovery of the defect. Recording of the reproduction steps, effect on the product/user, assignment of initial severity	Submit	(New)/Submitted	
2	Triage review of the defect, assignment of defect <i>with request to evaluate defect repair and make recommendation for future correction</i>	<i>Assign, Refer</i>	Submitted/Open	Open/Submitted
3	<i>Review of repair recommendation. Determination to correct in a future project phase / iteration</i>	<i>Postpone</i>	Open/Submitted	Submitted/Postponed

- b) Rejection of fix by test, recommendation to postpone from development

Step	Activity	State Transition	From State	State/To
1	Initial discovery of the defect. Recording of the reproduction steps, effect on the product/user, assignment of initial severity	Submit	(New)/Submitted	
2	Triage review of the defect, assignment of work priority and passed to the development team for action	Assign	Submitted/Open	
3	<i>After development review of defect the repair estimate is significant and other, better, options should be pursued (perhaps a design change is in order). Refer back to the Triage team for re-consideration of priority</i>	<i>Refer</i>	Open/Submitted	
3	Review of repair recommendation. Determination to correct in a future project phase / iteration	Postpone	Submitted/Postponed	

- c) Recommendation from test that proposed fix either does not fully address the base defect or wait for a better fix to be made available later.

Step	Activity	State Transition	From State	State/To
1	Initial discovery of the defect. Recording of the reproduction steps, effect on the product/user, assignment of initial severity	Submit	(New)/Submitted	
2	Triage review of the defect, assignment of work priority and passed to the development team for action	Assign	Submitted/Open	
3	Defect repair, package into next available build. Assign to test team for verification	Resolve	Open/Resolved	
4	Verification of the repair nominally passes but results of the repair has adverse affects on other areas of the application. Recommendation (from test team) that the repair approach should be re-considered and addressed at a later time.	<i>Refer</i>	Resolved/Submitted	
4	<i>Review of repair recommendation. Determination to correct in a future project phase / iteration</i>	<i>Postpone</i>	Submitted/Postponed	

Defects That Are Temporarily Fixed, To Be Fully Repaired At a Later Time

Perhaps one of the most difficult of defect decisions is to apply a temporary fix to a reported defect. An example of this situation could arise when a vital feature of a product is failing just prior to a beta test release. In this case the immediate fix could be simply to disable the feature and, since the temporary fix requires testing, the desired resolution is to “fix” the defect and verify the “fix” is working as expected. At a later time the real problem would be scheduled for resolution, repaired, verified, and closed.

The defect model approach to this would be to “clone” the original defect and link the two defects together (linkage feature of the defect tool employed). One defect then can be processed with the temporary repair, tested, and closed. The second defect then should be appropriately notated, linked to the other defect, and postponed to a future test/release cycle. While one may consider also tagging these as duplicates, this is not recommend as the repro steps clearly would be different for the cloned defect once the temporary repair was deployed.

Once the original (temporary) defect repair is verified and deployed, that defect can be closed with resolution *fixed*. The “cloned” (and linked) defect can be closed with resolution postponed and at a later time re-opened, (re-)reviewed and otherwise treated as a new defect, albeit with a history but otherwise processed through the defect process as a new defect.

Defects That Are Duplicates of Other Defects

It is common in many systems to have duplicate defect reports for a large variety of reasons. Perhaps most common is simply multiple users and/or multiple testers reporting the same defect unaware that others have already reported the problem. Similarly applications that have wide test distributions (for example WinXP Beta) and/or automated bug reporting systems can generate a very large number of duplicate defect reports (Deuel, 2004) Looking narrowly at test teams that not only discover and report defects but also have access to the defect database, a best practice is to search to see if anyone has already reported the defect before opening a new defect.

Two lines of response to duplicate defects also are common depending on just how obvious the duplication. If the defect is patently obvious, for example multiple reports of “Company title YYYYY is miss spelled on XXXX report header” the duplication typically can be detected in the triage process. The second path would occur when the defect actually needs to be researched prior to determination that it is a duplicate, for example consider these two reported defects:

1. Unable to log-on as application administrator
2. User log on for known valid user rejected

The root cause of both defects, upon investigation, may be the same problem in which case they might be considered duplicate defect reports as the defect repair is the same.

The defect process activities for these two duplicate paths would appear as follows. For the obvious duplicate:

Step	Activity	State Transition	From To State	State/
1	Initial discovery of the defect. Recording of the reproduction steps, effect on the product/user, assignment of initial severity	Submit	(New)/Submitted	
2	Triage review of the defect and detection of the duplicate report. Linking of the duplicate defect(s) to the first reported case of the defect (typically the lowest number defect) and assignment to test for validation	Resolve (as duplicate)	Submitted/Resolved	
3	Assigned tester verifies that the duplicate defect in fact is the same as the first defect making any appropriate annotations in the master defect as required	Duplicate	Resolved/Duplicate	

Note that there are two possible state transitions in this process flow (*Close*, and *Duplicate*). The less obvious duplicate process flow would appear as follows:

Step	Activity	State Transition	From To State	State/
1	Initial discovery of the defect. Recording of the reproduction steps, effect on the product/user, assignment of initial severity	Submit	(New)/Submitted	
2	Triage review of the defect which does not detect the duplication. Defect is appropriately prioritized and passed onto the development test for analysis or correction	Assign	Submitted/Open	
3	Development team identifies that this is in fact a duplicate of another defect, notes in the defect log and links to the master defect associated with the root problem.	Resolve or Duplicate	Open/Resolved (noted as a duplicate)	
4	Assigned tester verifies that the duplicate defect in fact is the same as the master defect making any appropriate annotations in the master defect as required	Close or Duplicate	Resolved/Duplicate	

Again note the multiple state transitions identified (*Resolve / Duplicate*, *Close / Duplicate*). In both of these process flows processing of duplicate reports can be problematic and to some extent dependent on the defect processing tool used, its capability for linkage of duplicate reports, and team policy on processing of duplicates. Looking into these issues yields four threads:

1. Tool capabilities, ability to link defects together in a meaningful manner
2. Local policy on closure of duplicate defects
3. Metric reporting on duplicates
4. Testing the repair to duplicate defects

Most defect management tools have the ability to link defect records. Useful in a variety of situations this features allows related defects to be associated and is often used for duplications, regressions defects, and as a means to supply additional information about a defect. This same mechanism can be used to attach duplicates together in a forward/backward linkage structure, the “master” (typically the first reported instance of a duplicate defect report) pointing to all the duplicates, the duplicates pointing to the master.

For systems (like Clearquest) that allow a “true” duplicate status to be established the system automates the downstream side of the defect management process by flagging all the linked duplicate entries as resolved when the master is resolve, and closed when the master entry is closed. For simple, obvious duplicates this process can be a time saver for the QA team as the individual duplicated entries need not be individually resolved and closed, but, there are situations where indeed the individual verification may in fact be warranted. Consider the following two scenarios:

Test team consisting of 5 testers who have divided 30 reports between them for testing. Every report has some common elements, and some elements distinct to the individual reports. “Duplicate” defects are discovered and reported, once for every report where the defect was observed.

- d) Defect #1 – Trivial defect, for example a date format (specification that the date should be localized, but it is fixed to a mm/dd/yyyy format).
- e) Defect #2 – All reports have a sort ascending / descending function but sort descending is not working.

Since the 5 testers report the same problem on all 30 reports there is one “master” defect and 29 duplicates for each of the two problems.

In both cases the dev team identifies the problem as a common component that once repaired should immediately be corrected on all reports. In the case of the first defect (localization of date display) it probably would be safe to link the 29 duplicate defect records to the master and then simply verify that the fix was correctly display on the one report associated with the master defect. Having verified that the localization is in fact working correctly on this report there is little risk (in this case) in a mass closing of the duplicate defects. Good software test practice would probably cause the tester closing the master to check at least one other report to verify that duplicate designation was in fact correct, but beyond that likely would not warrant individual verification and defect closure on all 30 defect reports.

In the second case (missing sort descending) one might argue that while the defect may in fact be duplicate from a software design perspective, that they should in fact be individually verified by the test team to make certain that not only is the sort descending actually working, but that the results of the display is in fact correct.

In this discussion one can see that the term “duplicate” is not quite as precise a term as we might like to have, and can be viewed from three different perspectives:

1. Duplicate defect report – The very same problem is in fact reported more than once – Typical of a large test team doing a “bug bash” where several reports could come in at the same time from different testers and/or different locations. Very typical in Alpha and Beta test cycles.
2. Duplicate reporting of problems from different contexts, for example the localization of the date format - - Same problem, different reports.
3. Reporting of the same underlying (software code wise) problem where from the tester’s perspective it is not at all clear that a common software component is involved (e.g. defect #2 – sort descending not working).

Use of “duplicate” status is clearly useful in the first case (multiple reports of the exact same problem) as there is very little value added in closing multiple bugs with the exact same test steps.

Use of duplicate status (and mass closing of resolved defects) may in fact be acceptable in the second case (e.g. date localization fix). This likely comes down to a best practice or management policy for a project. While it is highly likely that the single fix did in fact resolve the problem for all 30 reports, there is the possibility that one or more reports failed to use the correct common component identified by the developer as the root cause. The correct method to verify that it indeed has been corrected is to individually inspect all 30 reports (again) and close the 30 “duplicate” defect individually having successfully verified the repair. Another way to view this situation is that the defects are duplicate from a developer’s perspective (same code change will fix all the reported problems) but not duplicate from the tester’s perspective (verification steps are different in some aspect).

The third cases (not functional sort descending) most likely should not be flagged as duplicate, even though the development team may argue that this, like case #2, is a common component. The key here is the test verification. The correct functioning of the sort descending only starts with the feature working. Verification that the sort descending is working correctly in the context of the individual reports requires not only that the feature executes, but that the results which are context sensitive actually make sense for the specific reports. In this case we suggest that the defect not be flagged as “duplicate” and be individually verified and closed. This is a situation where the aforementioned linkage of defect records can be useful as the whole set of “duplicate” defects can be linked for

information purposes without being subject to a potentially erroneous mass closure of the defects without individual verification.

Considering the variety of reasons why duplicate defect reports may be created then the operative question becomes when is a duplicate report really a duplicate? From the developer perspective it likely is when the *cause* of the defect is the same as for other reports. From a program manager's perspective it likely is when the *reported problem* is the same as another defect. From the QA analyst's perspective duplicate problems likely can be *verified using* the same *test case*, or set of test cases. Our model supports both perspectives by allowing the defect model user to flag defects either as mass duplicates, flag as duplicates and link to related defects, or reclassify the apparent duplicate as a truly stand alone defect – one that requires a specific target test case to validate the repair.

One side note here has to do with defect metrics. Most teams that use defect metrics will have “duplicate” defects reported in some summary fashion and the designation of a particular defect being a duplicate of another of course drives this metric. The general desire is to get the overall “true” defect count down to the true unique defects in the test cycle. Where the duplicate defects are in fact obvious (case #1) this is probably appropriate as there is no additional work (dev or test) associated with resolving the duplicate defect reports. Where there is, or should be, additional research, repair, or verification of a correctly repaired defect to flag this as “duplicate” actually minimized the true cost of defect repair. Since verification of defect repair is a valid project activity wherever individual defect verification is required it would be most accurate to not flag the associated defects as “duplicate” from a metrics reporting perspective. Again project management policy needs to be established here. Do not allow the defect tool to drive the policy; the tool must be used in such a way that product quality metrics are accurately reported.

Re-Opening of Closed Defects

Aside from the trivial case of an inappropriately closed defect, there are two reasons our proposed model supports re-opening of defects already closed. The first and most important is to deal with defects postponed for a particular product or test cycle that need to be re-addressed in the next test or product cycle. The second deals with one method of addressing regression defects – defects that had been found and repaired in an earlier product test/release cycle but appear again in the current cycle.

The defects postponed for a test/release cycle and then closed (see in 5.2, Defect that must be postponed) can and should be reviewed at the next test/release cycle. The discussion of what the reported problem was, recommendations, decisions, etc. will all be in the defect notes. The decision for the product/project manager is simply whether these should be re-opened (re-activated) or not for this test/release cycle. A recommended approach here is simply to re-open all postponed defects and re-triage the defects with appropriate notations in the defects simply starting afresh as if they were newly reported defects, albeit with a “history”.

While this approach may work for many projects, other may choose to selectively re-open postponed defects as some defects truly may be “really closed”. For example, a postponed defect for a feature that finally was removed from a product could, and should be closed permanently. In this case the triage team decision would likely leave the defect closed, change the resolution from “postponed” to “won't fix” and note the explanation in the defect history.

Defects that are re-opened using our model are simply dropped into the submitted state, ready for (re-) review, prioritization and action as warranted in the current test/release cycle.

Regression defects – those defects that were seen before, thought to be repaired, and appear again – can be dealt with in two different methods. Both of these techniques are supported by the model, the determinate here is more metrics related than process related.

The recommended approach to regression defects is to consider the defect as a new defect, which is to follow normal defect reporting processes and open a new defect describing the current repro steps, expected results and actual results. If in analysis it is determined that this is actual a regression defect the linkage of the new defect to the (now closed) previously reported defect can give additional context to the “new” defect report. The linkage can provide the development team valuable history on when the defect was last reported, how it was resolved, etc.

The alternative approach would be to reactivate the “old” defect, flag the “new” defect as a duplicate of the original defect and use the reopened defect as the controlling defect to drive to a new resolution. This method provides the

duplicate feature of linkage with the benefit that when the “master” (i.e., re-opened) defect is resolved, the duplicate (i.e. new) defect is automatically resolved at the same time.

CONCLUSION

We have derived a Full Life Cycle defect model which can be used by organizations that develop software products for others as well as for those that develop software applications for themselves. We have described this model, the various states, as well as the paths and parameters that the defect process may take in this model. In addition, we have discussed the impact and intersection of this defect process model with the project life cycle model and the testing model.

Our proposed model strikes a balance between loosely organized 2, 3, or 4 state models and the tightly controlled (10 plus) process state models, that supports the three primary actors in the defect management process. Project management actors can control the overall flow of defects from the Submitted, Postponed, and Closed states (including the ability to reopen closed defects for additional work.) Development actors can continue to fix resolvable defects, suggest that reported defects are in fact duplicates of other defects, and recommend that certain defects be postponed either permanently or for the current project cycle. QA/Test actors have also have the option of flagging defects as duplicate from a test perspective (same test can verify multiple defects) as well as the normal close and reject (defect not fixed) actions typical of most defect management systems.

In the end it is the flexibility of the process that determines the usefulness of the process (Goldin, Rochell, 2002), not the specific tool value of the defect process. Our model can be easily implemented in IBM/Rational Clearquest, Microsoft Team Systems, as well as many of the commercial and freeware defect/bug tracking systems. The key here then is not the implementation or similarity to existing implementation but the understanding of what really needs to happen within the defect management process itself. With this understanding the combination of tools, tool enforced state transitions, and manual procedures can implement our Full Life Cycle defect management process.

Our next step is to develop an infrastructure to support the recording and management of defects according to this comprehensive defect model. We are also investigating ways of using this model to help gather and evaluate defect process metrics.

REFERENCES

Beniaminy, I. and Joseph, D. (2002). Reducing the “No Fault Found” Problem: Contributions from Expert System Methods. Aerospace Conference Proceedings, IEEE Vol. 6.

Black, R. (1999). Managing the Testing Process, Wiley.

De Man, J. and Ebert, C. (2003). A common Product Life Cycle in global Software Development, Proceedings of the Eleventh Annual International Workshop on Software Technology and Engineering Practice (STEP'03) - Volume 00, IEEE Computer Society, Washington D.C.

Davis, A. (2003). The Art of Requirements Triage, Computer, Volume 6, Issue 3, March, 48.

Deuel, R. (2004). Automated Bug Tracking: The Promise and the Pitfalls, IEEE Software, January/February.

Goldin, L. and Rochell, L. (2002). Software Development Bug Tracking: Tool Isn't User Friendly or User Isn't Process Friendly - Lecture Notes In Computer Science; Vol. 2349, Proceedings of the 7th International Conference on Software Quality, Springer-Verlag, London, UK.

Humphrey, W. (1989). Managing the Software Process, Addison-Wesley, p. 191.

IBM (2005). IBM® Rational® ClearQuest® Deployment Kit, Usage Model Guidelines, Version 1.1, Jan., p. 4.

Jacobson, Booch, Rumbaugh. (1999). *The Unified Software Development Process*. Addison-Wesley. 302.

Kipman, A. (2006). Inside a MS Build Bug Triage Meeting.
<http://channel9.msdn.com/ShowPost.aspx?PostID=26641>

Microsoft. (2005). Process Guidance documentation provided in the documentation for Microsoft Visual Studio,
http://guides.brucejmack.biz/MSF%20Agile%20Beta%202%20PT/Wss/Process%20Guidance/Supporting%20Files/Bug_StatesandTransitions.htm

Rahman, A. (2004). The Framework of a Web-Enabled Defect Tracking System Advanced Communication Technology 2004 Conference Proceedings, Volume 2, 609 – 695.

StickyMinds.com. (2006).
<http://www.stickyminds.com/tools.asp?Function=Search&topcat=DFTRK&Kind=toolsimple>

