

Analysis of Usage of Defect Tracking Systems

Sudhir¹, Shikha Khera²

¹M.Tech. CSE Student, SPGOI, Rohtak, Haryana (India)

²Assistant Professor, CSE Department, SPGOI, Rohtak, Haryana (India)

Abstract

"A problem well stated is half-resolved". This quotation really captures the fact that a well written defect report means that it should be treated as half-fixed defect (fault or bug). Defect Tracking Systems are the Integral and probably the most important part of a Defect Management System. This paper throws the light on effective and efficient usage of a defect tracking tool and also discusses about some misconceptions regarding defect tracking tools.

Keywords: *Defect Tracking Systems, Defects, Defect Management Systems*

I. Introduction

Defect tracking is a essential project management tool. They are called in many ways: Issue Tracking Systems, Trouble Tracking Systems, Bug Tracking Systems, Requirements Tracking Systems, etc.; however, their purpose remains the same: collecting requirements, their management, and tracking their progress.

There is an old saying that any program is never truly finished. There is always something that needs to be added, redone, or fixed. A document containing a call for an adjustment of a system is called a Change request.

✓ What Should Issue Tracking Systems Do?

When there are constantly new requirements coming, in such situations tools are necessary, which would allow somebody to fully and easily:

- Share the information across the team.
- Have an instant overview of the state of the software.
- Expertly decide about releasing
- Set and update the importance of individual fixes and adjustments
- Have a recorded history of changes

✓ Way Of Collecting Change Requests :

In the real world, there are various ways how to gather change requests:

- Crash reporting tools (Break Pad, Bug Buddy, Dr. Konqi, Windows Error Reporting, ...);
- Issue tracking systems (Bugzilla, JIRA, Rational Clear Quest, Trac, ...);
- Contact forms (web forms, integrated);
- Discussion boards;
- E-mails; etc.

As one can see, the so-called issue tracking systems are especially marked because those are tools primarily designed to do this job, to fulfill the above-mentioned requirements. All others are just substitutes, tools primarily designed for different purposes.

✓ Benefits Of Using Defect Tracking Systems :

Defect tracking systems, if used in the right way, can bring following benefits:

- Improve the quality of software;
- Increase satisfaction of users and customers;
- Ensure requests accountability;
- Improve communication in the team and also to customers
- Increase of productivity of the team; and
- Reduce expenses.

II. Change Requests

✓ Composition Of A Change Request:

A change request is a document containing a call for an adjustment of a system; it is of great importance in the change management process. A change request is declarative, i.e. it states what needs to be accomplished, but leaves out how the change should be carried out. Important elements of a change request are:

- **Reporter's Name** or ID. If there are questions, one needs to know who originated the request.
- **Type of change request** states what kind of request it is (bug, enhancement, new requirement, ...)
- A **brief description** of what the problem is. For example, "Fatal error when printing landscape." is a good description; short and to the point.
- **Build or Version number** of the code being built. 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.
- The **component** must be specified. This facilitates assigning the bug to a developer assigned to that part of the product.
- 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 is setup. List each and every step (don't leave any out) to produce the bug. 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.
- **Severity** of the request tells the reader of the bug how bad the problem is. Or in other words, say what the results of the bug are. Here's a common list for judging the severity of bugs:
 - Blocker – blocks development and/or testing work, production could not run;
 - Critical – crashes, loss of data, severe memory leak;
 - Major – major loss of function;
 - Minor – minor loss of function or other problem where easy workaround is present; and
 - Trivial – cosmetic problem like misspelled words or misaligned text.
 - o Of course, usually a blocker is handled by development team immediately.
- To who is the bug **assigned? Who** is going to be responsible for the bug and do the work on the bug? Some companies make it so, that there is a set person whose only

responsibility is to reassign tickets to the proper person.

✓ Change Request Types:

Change requests typically originate from one of five sources:

- Problem Reports That Identify Bugs, Which Must Be Fixed (Forms The Most Common Source);
- System Enhancement Requests From Users;
- Events In The Development Of Other Systems;
- Changes In Underlying Structure And /Or Standards (E.G. In Software Development, This Could Be A New Operating System); And
- Demands from senior management.
- Based on the above-mentioned description, one can categorize change requests into the following categories:
 - **Issue** can be described as "non-standard" or "not-awaited" behavior of the system. It can be a bug, but needn't be. This type is usually chosen by people who are not sure how to categorize their problem, and later is usually changed to some other type (bug/enhancement) or closed.
 - **Bug** A software bug (or just "bug") is an error, flaw, mistake, failure, fault, or "undocumented feature" in a computer program that prevents it from behaving as intended (e.g., producing an incorrect result). Most bugs arise from mistakes and errors made by people in either a program's source code or its design, and a few are caused by compilers producing incorrect code.
 - **Enhancement** This category contains requests for improvements or enhancements to an existing feature or task.
 - **Requirement** Humans usually classify things as requirements when they include new functionality, new features to a system, either determined by ourselves or requested by a customer.
 - **Task** This is a common category for both functional and non-functional tasks that need to be done. For example, in this category we can put requests for hardware upgrades or other environmental updates, etc.

- Of course, taxonomy of change requests can vary depending on company conventions.

✓ Change Request Workflow

The life of a change request starts, as it is opened by somebody. Usually, the submitter of a request does not know (and does not need to know) who the appropriate person for handling the request is, so it is assigned to a lead domain expert. The lead domain expert is the person responsible for assigning requests to the appropriate person in the development team.

After receiving a new request ticket, the first step is verification. A bug verifier 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 ticket can be re-assigned to a person who will be responsible for fixing the bug. If the bug is not written clearly, is missing some steps, or cannot be reproduced, it will be sent back to the bug reporter for additional information. Problems can be also marked as “Fixed Duplicates,” meaning that the problem is specified as the duplicate of the already fixed one.

The developer accepts the request ticket, which indicates that he/she has seen it and is aware that it is his/her responsibility to resolve it. The developer works on the request and based on the conclusions; he/she assigns a status to it, indicating what the next step should be. The problem described is an issue which will not be fixed. The following reasons are possible:

- The request is “Postponed,” meaning that because of technical reasons, time constraints, or other factors it will not be incorporated into the code until the next version of the product.
- Problem is marked as “Known problem,” indicating that the problem or a similar request is already known, and either a workaround exists for handling it, or it is approximately scheduled some time when the proper fix of the problem can be expected
- Issue is “Rejected” by the developer and should be forced to accept (through Council) or closed:
- Issue is “Rejected for other reasons” – for example, for the lack of information, change of conditions, impossibility to replicate the issue, etc.

- The developer has examined the issue, the product requirements, and the design documents and determined that the bug is not a bug; it is “Working as Designed”. What the product or code is doing is intentional as per the design.
- Issue is “Deferred” – meaning that the cost of fixing the bug is too great, given the benefits that it would produce.
- Problem is scheduled by the developer and will be fixed, what can result in following fix states:
- 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 regular basis, looking for all “Fixed status” tickets.
- Duplicate status, again, indicates that the problem is found as a duplicate of an existing issue.
- Resolved means that the problem has been taken care of, but no code has been changed. For example, bugs can be resolved by getting new device drivers or third party software.

After being **fixed**, the bug is reassigned back to the submitter, a representative, or tester, who retests the bug by duplicating the original circumstances. If the bug is fixed and everything now works properly, another test with slightly different circumstances can be performed to confirm the fix. If the bug passes the tests, it gets a **Verified** status. If the bug does not pass the test, the request is re-opened and is sent back to the developer.

After being tested, the ticket is **closed**. Here the lifecycle of the change request ends; however, a regression of the problem may occur in the future may occur, and a request can be **re-opened** and the cycle starts again.

III. Defect Reporting Best Practices

As soon as we run into a problem in the software, a new issue request (ticket) is created. In a well-written report, explain how to reproduce the problem;

- Analyze the error so that it can be described in a minimum number of steps;
- Include all the steps;
- Make the report easy to understand;
- Keep tone neutral and non-antagonistic;

- Keep it simple: one bug per report; and
- If a sample test file is essential to reproducing a problem, reference it and attach it.

As was already stated, there are many ways of how a request can end “in the trash” instead of being addressed. To raise the chance of an issue being fixed, there are couple rules which should be followed.

Rule #1 – Write good summary

This is one-line description of the problem which is the most important part of the report .

- The project manager will use it in when reviewing the list of bugs that have not been fixed.
- Executives will read it when reviewing the list of bugs that will not be fixed. They might only spend additional time on bugs with "interesting" summaries.
- The ideal summary gives the reader enough information to help to decide whether to ask for more information or not. It should include:
- A brief description that is specific enough that the reader can visualize the failure;
 - o A brief indication of the limits or dependencies of the bug (how narrow or broad are the circumstances involved in this bug?); and
 - o Some other indication of the severity (not a rating, but helping the reader envision the consequences of the bug).

Rule #2 – Explain how to reproduce the problem

- First, describe the problem. What is the bug? Do not rely on the summary to do this.
- Next, go through the steps that you used to recreate this bug.
 - o Start from a known place (e.g. boot the program);
 - o Then describe each step until you hit the bug;
 - o NUMBER THE STEPS. Take it one step at a time.
- If anything interesting happens along the way, describe it. (One must give people directions to a bug. Especially in long reports, people need landmarks.)
- Describe the erroneous behavior and, if necessary, explain what should have happened. (Why is a bug? Be clear.)

- List the environmental variables (configuration, etc.) that are not covered elsewhere in the bug tracking form.
- If the reader may have trouble reproducing the bug (special circumstances are required), be clear about these circumstances.
- Keep the description focused:
 - o The first part of the description should be the shortest step-by-step statement of how to get to the problem.
 - o Add "Notes" after the description, if there are any typical notes include:
 - o Comment that the bug will not show up if step X is executed between step Y and step Z.
 - o Explain the reasoning for running this test.
 - o Explain why this is an interesting bug.
 - o Describe other variations of the bug.

Rule #3 – If two failures are visible, write two reports

Combining failures on one report creates problems:

- The summary description is typically vague. Words like "fails" or "doesn't work" are used instead of describing the failure more vividly. This weakens the impact of the summary.
- The detailed report is typically lengthened. Bug reports should not read something like the following: “Do this until that happens, in which case do not do this until the first thing is completed and then the test case of the second part must also be complete, and sometimes you may see this, but if not then that....”
- Even if the detailed report is rationally organized, it is longer (there are two failures and two sets of conditions, even if they are related) and; therefore, more intimidating.
- Often, one bug gets fixed, but not the other.
- When reporting related problems on separate reports, it is a courtesy to cross-reference them.

Rule #4 – Eliminate unnecessary steps

Sometimes, it is not immediately obvious which steps can be dropped from a long sequence of steps in a bug. Look for critical steps. Sometimes, the first symptoms of an error are subtle.

A list now exists of all the steps that were taken to show the error. Now, the aim is to shorten the list. As each step is executed, any hints of errors must be examined. The following factors should be examined:

- Error messages (i.e. A message appeared 10 minutes ago. The program did not fully recover from the error and the problem evident now is caused by the poor recovery.)
- Delays or unexpectedly fast responses.
- Display oddities, such as flashes, repainted screens, a cursor that jumps back and forth, multiple cursors, misaligned text, slightly distorted graphics, doubled characters, omitted characters, or display droppings (pixels that are still colored even though the character or graphic that contained them was erased or moved).
- Sometimes, the first indicator that the system is working differently is that it sounds a little different than normal.
- An in-use light or other indicator shows that a device is in use when nothing is being sent to it (or a light that is off when it should not be).
- Debug messages: the debug monitor should be turned on (if it exists on the system) and should be monitored if or when a message is sent to it.

If, what seems like a critical step has been encountered, everything else from the bug report should be eliminated. One must now go directly from that step to the last one (or few) that shows the bug. If this does not work, individual steps or small groups of steps can be removed.

Rule #5 – Variations after the main report

The failure may look different under slightly different circumstances. For example:

- The timing changes if additional two sub-tasks are performed before the final reproduction step is hit.
- The failure will not show up at all or is much less serious if something else is put at a specific place on the screen
- The printer prints different characters (instead of the characters described) if the file is made a few bytes longer.

This is all useful information for the programmer and it should be included. But to make the report clear:

- It should start with a simple, step-by-step description of the shortest series of steps needed to produce the failure.

- The failure must be identified. (Descriptions of how it looks like or what impact it will have.)
- A section should be added that states "ADDITIONAL CONDITIONS" that describes, one by one, the additional variations and the effect on the observed failure

IV. Overcoming Objections

Programmers, as humans, operate under time constraints and competing priorities. For example, outside of the 12-hour workday, some programmers are faced with fatigue and apathy, instead of fixing bugs.

Some things will often motivate programmers to fix the bug:

- The situation with the bug is very poor.
- The complexity piques the programmer's curiosity.
- It affects many people.
- Arriving at the fix is trivial.
- It has embarrassed the company, or a similar bug embarrassed a competitor.
- One of the bug's cousins embarrassed the company or a competitor.
- Management wishes for the bug to be fixed.

Programmers avoid spending time on a bug due to the following reasons:

- The programmer cannot replicate the defect.
- Strange and complex sets of steps are required to induce the failure.
- Not enough information is known and it will take a lot of work to solve them.
- The programmer does not understand the report.
- The solution is unrealistic (e.g. "corner case").
- It will take a lot of work to fix the defect.
- A fix will introduce too much risk into the code.
- There will be no perceived customer impact.
- The fix is unimportant (nobody cares about such a minor error or unused feature). It is not a bug, it is a feature.
- Management does not care about bugs of this type.
- The programmer does not like or trust the person with the requests, or the customer who is complaining about the bug.

V. Conclusion

One of the most important things a tester does is to write "Defect Report" or "Bug Report". A defect report is a technical document which describes the failure mode in AUT (Application under Test). How well a tester creates a bug report directly affects how likely the programmer is to fix it. Reporting a bug is nothing but a skill. Some testers get much better response from development team than others due to their well written defect report. The objective is not to write the perfect defect report, but to write an effective defect report that conveys the proper message, gets the job done and simplifies the process for everyone. It is worth the effort to learn how to write effective bug reports. In this paper, we have discussed practices that can be applied to make a Bug Report effective.

References

- [1] Aaen, I. (2003). Software Process Improvement: Blueprints versus Recipes. IEEE Software 20(5), 86–93.
- [2] Adeel, K., (Aug. 2005). Defect prevention techniques and its usage in requirement gathering – industry practices , Proceedings of Engineering Sciences and Technology, ISBN 978-0-7803-9442-1, SCONEST, IEEE Computer Society Publisher, pp 1-5.
- [3] Boehm, B. and V. Basili (2001). Software Defect Reduction Top 10 List. Computer 34(1), 135–138.
- [4] Bassin, K. and P. Santhanam (2001). Managing the maintenance of ported, outsourced, and legacy software via orthogonal defect classification. In Proceedings of the IEEE International Conference on Software Maintenance, Washington, DC, pp. 726. IEEE Computer Society.
- [5] Boehm, B. , Basili,V., (2001) Software Defect Reduction Top 10 List Journal Computer archive Volume 34 Issue 1, IEEE Computer Society,, CA, USA
- [6] Boehm, B. and V. Basili (2001). Software Defect Reduction Top 10 List. Computer 34(1), 135–138.
- [7] Børretzen, J.A. , Conradi, R., (2003). "Results and Experiences From an Empirical Study of Fault Reports in Industrial Projects", Proceedings of the 7th International Conference on Product Focused Software Process Improvement, Amsterdam, pp. 389-394
- [8] Børretzen, J.A. , (Sept. 2009) .Investigating the Software Fault Profile of Industrial Projects to Determine Process Improvement Areas: An Empirical Study", Proceedings of the 14th European Systems & Software Process Improvement and Innovation Conference, Potsdam, Germany, pp. 212-223.
- [9] Card, D. N. (2002). Managing software quality with defects. In Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopmentpp. 472–474.
- [10] Card, D.N., (October 2006). Myths and Strategies of Defect Casual Analysis, Proceedings of twenty-fourth Annual Pacific Northwest Software Quality Conference, Portland, Oregon, pp 469-474.
- [11] Chillarege, R. and K. Prasad (2002). Test and Development Process Retrospective -A Case Study Using ODC Triggers. In Proceedings of the International Conference on Dependable Systems and Networks, Florence, Italy, pp. 669–678. IEEE Computer Society Press.
- [12] Drago, J., (2011). "Role of Defect Management Software in Software Development Life cycle", Available at -<http://ezinearticles.com/?Role-of-Defect-Management-Software-in-The-Software-Development-Life-Cycle&id=6205930>.
- [13] Fredericks, M., Basili ,V.,(1998). "Using defect tracking and analysis to improve software quality". Report DACS-SOAR-98-2, Experimental Software Engineering Group, University of Maryland, College Park, MD, USA
- [14] Fairley, R. E. and M. J. Willshire (2005). Iterative rework: The good, the bad, and the ugly. Computer 38(9), 34–41.
- [15] Howles, T. and S. Daniels (2003). Widespread Effects of Defects. Quality Progress 36(8), 58–62.
- [16] Humphrey, W. (2002). Winning with Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.
- [17] IEEE (2004). IEEE Standard for Verification and Validation/IEEE Std. 1012-2004.
- [18] Jones, C.(2009)," Software Engineering Best Practices: Lessons from Successful Projects in the Top Companies". McGraw-Hill, New York, NY, USA, 2009. ISBN 007162161X.
- [19] Kaner, C. (2002). Bug Advocacy. Retrieved from Cam Kaner Home Page: <http://www.kaner.com/pdfs/BugAdvocacy.pdf>
- [20] Khilman, A., (2005). "Defect Management Process in Software Testing", Thesis submitted

- in Faculty of Information Technology, MERA Technical University.
- [21] Lars Ola Damm (2007) "Early and Cost-effective Fault Detection", Dissertation submitted to Blekinge Institute of Technology, Sweden
- [22] Leszak, M., D. Perry, and D. Stoll (2000). A Case Study in Root Cause Defect Analysis. In Proceedings of the 22nd Int. Conference on Software Engineering, Limerick, Ireland, pp. 428–437. ACM Press.
- [23] Leszak, M. (2002). Classification and evaluation of defects in a project retrospective. *Journal of Systems and Software* 61(3), 173–187.
- [24] Lyu, M., (2007) "Software reliability engineering: A roadmap". In FOSE '07: 2007 Future of Software Engineering, pages 153–170, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5.
- [25] MacCormack, A., etal (2003). Trade-offs between Productivity and Quality in Selecting Software Development Practices. *IEEE Software* 20(5), 78–86.
- [26] Madsen, P. (2004). A Software Development Process based on Test-Driven Development and Testing by Contract. In Proceedings of the 11th Nordic Workshop on Programming and Software Development Tools and Techniques, Turku, Finland, pp. 225–240.
- [27] Mast, J. (2004). A Methodological Comparison of Three Strategies for Quality Improvement. *International Journal of Quality & Reliability Management* 21(2), 198–213.
- [28] Mathiassen, L., J. Pries-Heje, and O. Ngwenyama (2002). Improving Software Organizations: From Principles to Practice. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.
- [29] Maximilien, E. M. and L. Williams (2003). Assessing Test-Driven Development at IBM. In Proceedings of the 25th International Conference on Software Engineering, Portland, Oregon, pp. 564–569. IEEE Computer Society Press.
- [30] McGarry, J., D. Card, C. Jones, B. Layman, E. Clark, J. Dean, and F. Hall (2001). Practical Software Measurement: Objective Information for Decision Makers. AddisonWesley Professional, Boston, MA.
- [31] Mohagheghi, P., R. Conradi, and J. A. Bretzen (2006). Revisiting the problem of using problem reports for quality assessment. In Proceedings of the 2006 international workshop on Software quality, New York, NY, pp. 45–50. ACM Press.
- [32] (Myers, 2004) G. Myers, The art of software testing, Wiley and Sons, 2004
- [33] Ngwenyama, O. and P. A. Nielsen (2003). Competing Values in Software Process Improvement: An Assumption Analysis of CMM from an Organizational Culture Perspective. *IEEE Transactions on Engineering Management* 50(1), 100–112.
- [34] Niazi, M., D. Wilson, and D. Zowghi (2005). A Maturity Model for the Implementation of Software Process Improvement: An Empirical Study. *Journal of Systems and Software* 74(2), 155–172.
- [35] Ostrand, T.J., Weyuker E.J., (2005), "A Tool for Mining Defect-Tracking Systems to Predict Fault-Prone Files", 1st International Workshop on Mining Software Repositories, pp. 85-89.
- [36] Purushotham Narayan (2003), Software Defect Prevention in a nutshell, isixsigma Publisher, Publishing date 11 june 2003.
- [37] Runeson, P., C. Andersson etal (2006). What do we know about defect detection methods? *IEEE Software* 23(3), 82–90.
- [38] Runeson, P., M. H. Jonsson, and F. Scheja (2004). Are found defects an indicator of software correctness? an investigation in a controlled case study. In Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04), Washington, DC, pp. 91–100. IEEE Computer Society Press.
- [39] Shull, F., V. Basili etal (2002). What we have learned about fighting defects. In Proceedings of the Eight IEEE Symposium on Software Metrics, Ottawa, Canada, pp. pp. 249–258. IEEE Computer Society.
- [40] Suma, V., Gopalakrishnan, T.R., (2010). "Impact Analysis of Inspection Process for Effective Defect Management in Software Development", *American Society for Quality (ASQ) Journal, Software Quality Professional, USA, Vol. 12, No. 2.*
- [41] Tian, J. (2001). Quality assurance alternatives and techniques: A defect-based survey and analysis. *Software Quality Professional* 3(3), 6–18.
- [42] Tian, J., (Feb 2005). Software Quality Engineering: Testing, Quality Assurance and Quantifiable Improvement, Wiley John & Sons Publisher, ISBN 13-9780471713456
- [43] Tomaszewski, P. and L.-O. Damm (2006). Comparing the fault-proneness of new and modified code: an industrial case study. In Proceedings of the 2006 ACM/IEEE international symposium on International

- symposium on empirical software engineering, Rio de Janeiro, Brazil, pp. 2–7. ACM Press.
- [44] Vasudevan, S., (2005). Defect Prevention Techniques and Practices, Proceedings of fifth annual International Software Testing Conference, Bangalore, India
- [45] Van Solingen, R. (2004). Measuring the ROI of Software Process Improvement. IEEE Software 21(3), 32–38.
- [46] Voas, J. (1999). Software Quality's Eight Greatest Myths. IEEE Software 16(5), 118–121.
- [47] Wagner, S. and T. Seifert (2005). Software quality economics for defect-detection techniques using failure prediction. In Proceedings of the third workshop on Software quality, pp. 1–6. ACM Press.
- [48] Watts, S.H., (1989). Managing the Software Process, Defect Prevention, ISBN 0102180952, 9780201180954, Addison Wesley Professional Publisher.
- [49] Wagner, S.,(Sept.2008). A literature survey of the quality economics of defect-detection techniques, Proceedings of the 2008 ACM/IEEE International Symposium on Empirical Software Engineering, Rio de Janeiro, Brazil.
- [50] Westland, J. C. (2004). The cost behavior of software defects. Decision Support Systems 37(2), 229–238.
- [51] Zimmermann, T., Battenberg, N., Premraj, R., Kim, S., (2010). “Extracting Structural Information from Bug Reports” Retrieved on 06 June 2010.