Defect Detection By Developers

D.T.V. Ramakrishna Rao
Infosys Technologies Limited

Poor quality caused by defects continues to be a major problem facing the software industry. Unlike the traditional way of handling this problem where testers detect defects, this article suggests approaches where developers detect defects. This approach builds upon existing techniques, augments certain activities that developers often already engage in, and focuses on detecting defects in existing code. The end result is little additional effort in defect detection, easier fixes, and enhancing the effectiveness of the original intention of the activities.

Bugs plague almost all software systems. Indeed, more than half the time spent in a typical software project is on bug fixing [1]. Given the severe consequences bugs may have and the significant percentage of project effort associated with them, tackling bugs is of fundamental importance.

Defect detection is the primary strategy used to tackle defects, with testing being the predominant way defect detection is accomplished. In the past decade, inspections have been increasingly used to supplement testing. Still, defects present a significant problem. The software industry must continue to find new *cost-effective* ways to supplement current strategies to find defects.

This article proposes new approaches to detect defects. Developers engage in certain activities, often to detect defects in their *new* code. The new approaches are additions to these activities towards detecting defects in the *existing* code. These additions require little extra effort. Since defects in the existing code are detected almost as a by-product of these activities, they are referred to as *by-product defects* (BDs)[1]. This strategy has resulted not only in efficient detection of BDs but also in easier fixing.

I, along with a team at Infosys Technologies, have been applying these approaches for the past eight years on large and complex software systems and have found hundreds of defects.

Some developers use variations of the activities discussed in this article, but they are by no means universally used. Also, we have not seen a clear articulation of them in research literature related to software development. Hence, description of these activities can be construed as a description of a set of best practices for software developers. The primary contribution, however, is the addition made to these activities towards detecting BDs.

The primary sections of this article:

- Identify defect consciousness, an important ingredient for detection of BDs.
- Describe the activities during which developers find BDs.
- Relate experiences in deploying the activities.

Defect Consciousness

By defect consciousness, we mean the understanding of defects: what they are, what their types are, how they show up, what causes them, etc. Novices tend to lack this knowledge as programming courses generally do not emphasize this knowledge vis-à-vis writing programs. Developers normally gain this knowledge with experience, but not consciously. That leads to gaps in their knowledge. Hence, it is important to gain defect understanding consciously.

I found certain approaches to be rather effective in cultivating defect consciousness. Some books and papers focus on defects (for example, [2]), and are useful for a general understanding of defects. This understanding should be supplemented with defects in specific projects that developers are working on, as there are often defect types idiosyncratic to a project. Projects often have coding and other guidelines in this regard. While guidelines tend to be just bland statements, I found that augmenting a guideline by pointing at a specific fixed bug report (as an illustration) has many benefits. As the saying goes, "an example is better than a precept" [3]. It illustrates a defect in action: how it looks in code, how it shows up during execution, how it is debugged, etc. It helps in recognizing defects easily. Developers should also develop a life-long habit of understanding bug reports fixed by others.

The following activities may detect defects even without defect consciousness, but they will be more effective coupled with it.

Activities to Detect BDs

This section describes a set of activities during which BDs may be detected by developers: reading modules, regular reviews, triggered reviews, regular unit testing, and triggered unit testing. The activities are deployed in a typical industrial software development manner with dozens of developers working on a large and complex software system.

Description of each activity is organized as follows:

- What is the activity?
- When is it done?
- Why is it done?
- How is it done?
- How might BDs be detected as part of it?
- How does detecting BDs help the primary purpose of the activity?
- Why is it easier to fix BDs compared to defects detected by other means (such as traditional testing)?

Reading Modules

A module is a single file or a collection of files used to achieve a specific functionality. Developers should read a module for two reasons:

- **Reading modules for learning.** Reading code is the best (and sometimes the only) way of understanding the functionality implemented by a module. For example, in the case of networking protocols, protocol understanding from standards can be clarified and crystallized by reading the code implementing the protocol.
- **Reading modules while working on a bug or an enhancement.** Developers should completely read those modules that they are modifying as part of their work. It helps in not introducing bugs by making sure that *all* the required changes are made for their work in that module.

Many techniques exist for reading code [4]. A technique particularly effective to detecting BDs is active reading [5], which is a form of critical reading of the code: read a few lines, try to paraphrase them in your own words, ask questions, then try to answer those questions. I have seen in practice that reading a module once is not enough; reading twice is often sufficient. Active reading, coupled with defect consciousness, is very effective in both understanding code (primary purpose) and in detecting BDs.

When a defect is found while reading a module, it is much easier to fix compared to a traditional testing-found defect, for the same reason that inspection-found defects are easier to fix compared to testing-found defects (one can find a defect's location, what kind it is, as well as its cause) [6].

Comparison With Walkthroughs

Code reading (reading a module) as advocated in this article is a form of review, walkthrough, or inspection [6]. But there are notable differences. This section explores these differences and their implications.

Predominantly, code walkthroughs are used as a mechanism to review the implementation done by a developer. So, I will first compare walkthroughs and code reading in the context of a developer working on an enhancement to the code base.

A walkthrough is done to detect defects by a set of reviewers after a developer completes implementation. In contrast, code reading is done to detect defects by the developer during implementation. Hence, defects will be found sooner in code reading.

The intention of the walkthrough is to detect defects in the enhancement, so the developer walks the reviewers through his or her changes of the modules. The walkthrough focuses on the changes to a module per se, whereas in code reading, the focus is on the entire module even when only a few changes are made to it. The change in focus helps code reading by detecting even more defects than walkthroughs would. A module in a system might have gone through many modifications over time. As a module evolves, it tends to lose unity, becomes more complex, and hinders maintainability. Because of such evolution, some bugs tend to get introduced. When reading the whole module, there is a higher possibility of detecting those bugs.

Less often, code walkthroughs cover entire modules for special objectives (e.g., security audits). Code reading mainly differs in the way it is structured to efficiently and simultaneously achieve a novel combination of objectives: critical understanding of a module, detection of defects in the module (aided by critical understanding and defect consciousness), and ensuring that changes the developer is making to the module are complete and consistent (again, aided by critical understanding and defect consciousness). I am unaware of an existing code walkthrough that achieves the same objectives.

Regular Reviews

Reviews or inspections are used in some organizations to find bugs in the submitted artifacts [6]. During code reviews, reviewers try to find bugs in the submitted code changes with the help of additional documentation such as checklists and source documents (e.g., designs and requirements of the code changes). But during reviews, they may find bugs not only in the code changes but also in other documents. For example, Gilb [6] observed that inspections often find bugs in the source documents.

"A walkthrough is done to detect defects by a set of reviewers after a developer completes implementation. In contrast, code reading is done to detect defects by the developer during implementation."

I am not, however, aware of any prior observations that reviews may find bugs in code that is not part of the changes. But indeed, reviews are helpful in finding such bugs. If code changes show that a function is modified, reviewers should read related code surrounding the changes, functions that call the changed function, and functions called by the changed function. Reviewers need to understand the related code and, in light of that understanding, check whether the code changes have any bugs. For a reviewer with defect consciousness, the process of understanding the related code using active reading (as previously discussed) provides opportunities for uncovering bugs in that code and help in a more effective review.

BDs found during regular reviews, just like BDs found during reading modules, will be easier to fix.

Triggered Reviews

The reviews in the previous section are conducted before checking code changes into the code base; in some instances, however, there is a need to review code changes afterwards. Suppose you are making changes to your private copy of a code base while working on a bug or an enhancement. When multiple developers are working on the same code base, what you are doing may be affected by what others are checking into, in turn necessitating further changes. Therefore, you should go through each of the check-ins and review those that are related to your changes carefully.

Unfortunately, it is not always easy to know when a check-in is *related* to your change. From experience, I've found the following check-ins require careful review:

- **Check-ins that modify the modules you've changed.** It is very important to scrutinize such check-ins, as they are very likely to affect your changes. You are also in a good position to review those check-ins because of the familiarity with the changed modules (having followed the first activity: reading modules).
- **Check-ins that modify the subsystems you've changed.** Large complex systems are normally divided into subsystems, which in turn are divided into modules. For example, in a networking system, transmission control protocol implementation may constitute a subsystem. If a check-in modifies the subsystem you are changing, it is likely to affect your changes.
- **Check-ins that modify the subsystems you depend upon.** Software systems normally have a set of subsystems that are utilitarian in nature. They are often organized as libraries (e.g., a string-processing library). The rest of the system uses these subsystems. Changes to these subsystems tend to be rare but are not totally unheard of. If a check-in modifies such a subsystem that you are using, you need to update your changes. Moreover, all the changes to utility subsystems should be studied for continuing education on the project, as these subsystems are frequently used.
- **Check-ins related to your subproject.** Large enhancements tend to be implemented by multiple developers as subprojects. If your changes are part of a subproject, you should actively review all of the check-ins in the sub-

Activity	No. of Bugs	Percent
Reading Modules	32	48
Regular Reviews	7	10
Triggered Reviews	12	18
Regular Unit Testing	10	15
Triggered Unit Testing	6	9
Total	**67**	**100**

Table 1: *Bugs Found by the Activities*

project for three reasons. First, the check-in may be directly or indirectly related to your changes. Second, being part of the subproject means that you are in a good position to review the check-ins. Third, most subprojects are such that you may be working on yet another part of the subproject immediately after completing your current part. Hence, it is important to keep track of the design and implementation of the subproject on a continuous review basis.

The first step of reviewing a needed check-in is to understand the code changes made by it, and then assessing its impact on your changes. For a reviewer with defect consciousness, the process of understanding the checked-in code using active reading also provides opportunities for uncovering defects in that code.

BDs found during triggered reviews, just like BDs found during reading modules, will be easier to fix.

Regular Unit Testing

While modifying software, developers can also conduct unit tests to detect bugs in their changes. Their testing can be characterized by two aspects:

- **Tunnel vision.** They tend to concentrate only on the behavior of their changes.
- **Focus on end results.** The tests are often conducted simply to verify the outputs.

This type of unit testing is not very effective in uncovering bugs for two reasons.

First, it is not only important to check the output but also to check the entire processing that led to the output. Sometimes, intermediate processing may be incorrect, but the final result may turn out to be correct. For example, in a program if either function A or B returns true, further processing is undertaken. For a particular input, both the functions should return true but, due to a bug, function B returns false—yet the final result is as expected. Hence, the bug went undetected. These bugs in the intermediate processing may manifest in the future.

The second issue involves changes in large and complex systems: Developers may not be aware of the repercussions of their changes in other parts of the system.

More effective unit testing would take these two points into account. Developers should go through the processing of changed modules in minute detail either using a debugger in single-stepping mode or enabling *tracing* on the modules (a module supports tracing by printing debug output of its processing in great detail). To observe the impact of the changes on the rest of the system, enable the log messages at all levels and assertion checking on the entire system.

Developers should analyze the previously mentioned processing details and the messages produced by the system for anomalies (defect consciousness will aid here). Every anomaly needs to be analyzed to check whether it represents a bug, and (if so) whether it is pre-existing or if it was introduced by the developer's changes. If the anomaly is not a bug, it should enhance the understanding of the system for the developer. If the anomaly is a bug introduced by the developer, it obviously needs to be fixed. If the anomaly is an existing bug, the developer has found the bug and should open a bug report. The report should include the analysis already done to fix the bug faster in the future.

The enhanced unit testing helps both in detecting existing bugs in the system and in more effectively detecting bugs made by a developer.

Triggered Unit Testing

The activity described in this section applies in the same context as described in the Triggered Reviews section: The new check-ins may affect the changes being made. That section suggested reviewing the check-ins in this context. However, reviews may not catch all of the interactions that may exist between the check-ins and your changes. This is when testing becomes useful. Empirically, testing and reviews are shown to be complementary in their defect detection abilities [7].

After merging the check-ins with your changes, do not make any further changes to the code. Run a representative set of passed tests that you previously used for testing your changes (an application of regression testing in a development context). If the test fails, investigate to distinguish between two possibilities: Your code may need to be updated (in light of the check-ins), or there is a bug in the check-ins.

Of course, if your code needs to be updated, do so immediately. And, even if it represents a bug in the check-ins, the investigation already conducted is a good starting point for fixing the bug. Therefore, the complete investigation details should be a part of your bug report in order to fix the bug faster in the future.

Case Study

The described activities have been applied in multiple projects in our organization and have helped in finding hundreds of BDs. To show the benefits of these activities, I present a case study.

This case study shows the results of applying the activities by a single developer in a span of two years while enhancing and fixing bugs on a very large software system. The developer had about five years of experience at the beginning of the case study. The system was a C/C++ based mature networking software system having more than 50 million lines of code and was maintained by more than 100 people. The development process is typical of industrial software development. When a developer is enhancing or fixing a bug, he or she will do unit testing and submit the implementation to peer review. After the review, code is checked-in. When all enhancements and bug fixes for a release are in place, the testing team conducts integration testing and system testing.

Table 1 shows the distribution of BDs detected as a result of applying the activities by the developer. When the developer applied the activities, the detected defects were found in both the developer's new code and the existing code (BDs). Table 1 shows only BDs and not the defects detected by the developer in the new code. The BDs represent defects detected that leaked from the described formal stages of defect detection. For example, BDs detected by triggered review and triggered testing are missed by unit testing and peer review.

In all, the developer detected 67 defects. Reviewing code is found to be particularly effective. Almost half of the defects were detected during reading modules and about 75 percent were detected in some form of reviews. Triggered activities detected 27 percent of the BDs. Two main reasons accounted for their success: division of large enhancements into pieces to be done by multiple developers, and frequent modification of certain modules. Testing activities, however, should not be discounted; testing-detected defects, though fewer, tended to be of higher severity.

The case study shows that, using the activities, a developer can actually detect existing bugs in the system—just like a tester. More importantly, these defects are missed by formal stages of defect detection. The number of defects detected by the developer is of the same order as detected by a test engineer in the same timeframe. It is as if a test engineer was acquired for free!

Conclusion

This article discussed a set of activities for developers to detect defects in their new code, and augmented the activities to detect defects in existing code (BDs). This strategy has resulted in the following advantages:

- Detecting defects with little additional effort.
- Easier fixing of these defects compared to defects found during traditional testing.
- Enhancing the primary purpose of the activities that are augmented to detect BDs.

The current deployment of these activities falls far short of their potential utility. For more effective deployment, this article provides a starting point: Developers should enhance their defect consciousness and follow the activities as described. For long-term retention of these activities, they should be integrated with the development methodologies.

From a larger perspective, this article makes a small contribution regarding how developers may contribute more towards the quality of products. The current development methodologies do not fully utilize the expertise of developers in detecting defects. Proposed here are some strategies to utilize their knowledge. The techniques discussed basically fall into two categories: review and testing. There are many chances/reasons for developers to read or test code. Every such chance should be exploited to detect defects in the existing code, just as was done in this article.◆

Acknowledgment

The author sincerely thanks the CrossTalk Editorial Board, Kasey Thompson, Piyush Jain, Saravana Prasad, Thomas George, and Lilly Vasanthini for their comments on earlier versions of this article.

References

1. McConnell, Steve. Professional Software Development. Boston: Addison-Wesley, 2003.
2. Ploski, Jan, et al. "Research Issues in Software Fault Categorization." ACM SIGSOFT Software Engineering Notes 32(6): 6, 2007.
3. Answers.com. "Example Is Better Than Precept." 2008 <www.answers.com/topic/example-is-better-than-precept>.
4. Laitenberger, Oliver, and Jean-Marc DeBaud. "An Encompassing Life Cycle Centric Survey of Software Inspection." The Journal of Systems and Software 50(1): 5-31, 2000.
5. Clayton, Richard, Spenser Rugaber, and Linda Wills. On the Knowledge Required to Understand a Program. Proc. of the 5th Working Conference on Reverse Engineering. Honolulu, 12-14 Oct. 1998: 69-78.
6. Gilb, Tom, Dorothy Graham, and Susannah Finzi. Software Inspection. Boston: Addison-Wesley, 1993.
7. Jalote, Pankaj, and M. Haragopal. Overcoming the NAH Syndrome for Inspection Deployment. Proc. of the 20th International Conference on Software Engineering. Kyoto, Japan, 19-25 Apr. 1998: 371-378.

Note

1. BDs are so named not because they are introduced as a by-product of some activity, but because they are *detected* as a by-product of an activity.

About the Author

D.T.V. Ramakrishna Rao is a senior technical architect at Infosys Technologies Limited, in Bangalore, India. He has 14 years of experience in industrial software development with a primary focus on building high-end networking systems. He has published 10 papers in networking and defect analysis. He holds a master's degree in computer science from the Indian Institute of Technology in Kanpur, India.

Infosys Technologies Limited
44 Electronics City, Hosur RD
Bangalore – 560 100
India
Phone: 91-80-41166508
Fax: 91-80-28521695
E-mail: ramakrishnadtv@infosys.com

Coming Events

May 4-6
Sea – Air – Space 2009
National Harbor, MD
www.seaairspace.org

May 4-7
SATURN 2009 Conference
Pittsburgh, PA
www.sei.cmu.edu/architecture/saturn/2009

May 4-8
Software Testing Analysis & Review
Orlando, FL
www.sqe.com/stareast

May 11-12
2009 IEEE International Conference on Technologies for Homeland Security
Waltham, MA
www.ieeehomelandsecurityconference.org

June 1-4
DoD Enterprise Architecture
St. Louis, MO
www.afei.org/brochure/9a05/

June 8-12
Better Software Conference & EXPO
Las Vegas, NV
www.sqe.com/bettersoftwareconf

June 15-19
SIGMETRICS/Performance 2009
Seattle, WA
http://conferences.sigmetrics.org/sigmetrics/2009

June 28-July 3
21st Annual FIRST Conference
Kyoto, Japan
http://conference.first.org

2010
Systems and Software Technology Conference
Salt Lake City, UT
www.stc-online.org