

Crouching Tester Hidden Defect

A Paper on Where Bugs Hide

written by Adam Hughes, Senior Test Consultant, Planit Software Testing

Abstract

Bugs - just how many are there, and where do they hide? This paper looks at some of the different types of software bugs, what can lead to bugs occurring, where they might be hiding in your software, how many there could be, and how they can be found in the most efficient manner.

Introduction

Wouldn't it be nice if Testers knew where the bugs were going to be, before they started testing; if they could target certain areas of the code, even before they start designing test cases? Some organisations have a body of bugs that have been discovered on previous projects. Fewer have undertaken causal analysis against those bugs in order to understand their root causes. Fewer still have a catalogue of where each bug occurred, and can use that catalogue to identify where bugs are likely to occur on similar, future projects. A very small number of organisations can confidently predict how many bugs they need to find and fix, before it is safe to "go-live". For other organisations, it's a bit of a guess...

Kinds of Bugs

Boris Beizer is famous for his books and lectures on the topic of Software Testing. He is possibly most famous for initiating the largest study into bugs ever undertaken¹. The study looked at tens of thousands of bugs across the software development industry over a period of a decade, and found that there were **less than 200 bugs in the Software World**, but that these same bugs were constantly repeated across industries, and around the Globe. The bugs studied fell into the following high-level categories:

Bugs related to:	Percentage
Requirements and Specifications	24.31%
Code Structure	25.18%
Data	22.44%
Implementation	9.88%
Integration	8.98%
System and Software Architecture	1.74%
Test Definition and Execution	2.76%
Unspecified	4.71%

Several other, albeit smaller studies, have arrived at similar results, confirming Beizer's findings.

Why do Bugs happen?

An important thing to remember is that bugs are very rarely the result of malicious intent. Bugs may be exploited with malicious intent, but that is a separate subject (and a good reason for Testers to find bugs before people with malicious intent do exploit them). Generally, bugs are just...old-fashioned, human error...

Requirements

It's interesting to note in Beizer's study that almost 25% of all bugs can be attributed to requirements and specifications. So, whereas Developers often get the blame for bugs, the reality is, on average, **25% of bugs are already in place, even before a line of code is written**. Bugs in the requirements and specifications are typically due to content that is:

- Ambiguous;
- Incomplete
- Illogical; or
- Contradictory

If there are not clear and concise statements about what is required, then it is unlikely that the right thing will be built, **no matter how good** the design, and coding is. A great way to capture many of the bugs in the documentation is to get Testers to review the requirements to ensure that the requirements are testable. Clear and concise requirements don't just help Testers. They help Developers write great code. Simply, Testability Reviews are carried out because the more bugs found before requirements are signed off, the smoother the rest of the Initiative will run, the less Change Requests, bugs in the code, scope creep, calendar creep, code drops, and requests for additional funding there will be. The earlier you find the bugs, the easier it is to correct them (see below).

Specify	Design	Construction	Outcome
✓	✓	✓	Correct Function
✓	✓	✗	Correctable Errors
✓	✗	✗	Redesign Required
✗	✗	✗	Hidden Errors

Bug Clustering

One clear pattern that causal analysis of bugs shows, is that bugs tend to occur in groups, or clusters. Software bugs are just like any other bugs - where there is one, there is generally more.

Often, bugs occur in the most complex, or newest parts of the software. This is because they are the least understood, and least documented parts of the software. It is important to ensure that when new platforms, applications, functionality, or modules are introduced as part of a solution, that the necessary time is taken up front to ensure that everybody has the same view of what needs to be built, and why that precise solution is required. Poorly described requirements and specifications lead to differing interpretations. A misunderstanding can lead to tens of thousands of dollars worth of **rework**, serious project delays, and much loss of productivity. In any other industry rework is generally known as waste. Clear and concise requirements lead to less bugs.

Professional inexperience is a related source of bug clusters. It is more likely that a Developer with a few months of experience in a solution will code a bug than a Developer with ten years

experience in the same solution. Experienced Developers leads to less bugs.

Conversely, boredom can also lead to bug clusters. If a Developer has been working on largely the same area of code for a long period of time they can become bored. Familiarity may not breed contempt, but it can breed bugs. Developers are like anyone else - they make less mistakes when they find their work stimulating. Variety of work leads to less bugs.

Like all areas of life, mistakes are more likely to occur when we are in a rush. If there are external time pressures to get all of the code written by the end of the month, chances are, there will be more bugs in the code written at the end of the month in the rush to complete *on time*. If a Developer has been working twelve hours a day for two weeks to get the code finished, it is increasingly likely that tiredness and fatigue will lead to more bugs occurring in the code. Allowing enough time for coding leads to less bugs. This in turn can be linked to the initial estimates, if enough time was not allocated in the plan for a piece of development then the developer may be rushed to complete and held to the estimate.

Bug clusters can also be compounded. Assigning an inexperienced Developer to a poorly spec'ed, complex module of software, and then giving them a short deadline to complete the work in, is almost certainly going to lead to compounded bug clusters. When this occurs, bugs commonly hide behind other bugs, and cannot be found until related, upstream bugs have been resolved. This is often why severity 1 bugs are not identified until late into the testing cycle.

Different Technology, Different Bugs

Just as different plant species are susceptible to attacks from specific pests, different technology platforms are susceptible to particular bugs.

Many mainframe platforms have been around for over forty years. They tend to be stable, secure (at least within themselves), are very controlled from a path point of view, and are driven by procedural languages that dictate exactly how a workflow must happen. Mainframe bugs tend to be data bugs, functional bugs, and usability bugs.

Web platforms are still a new technology compared with mainframes, and even today, web languages are evolving. Web platforms tend to have more stability bugs, more security bugs (both at the application level, and at the user sanctioning level), more syntax bugs (although these are progressively being solved with more advanced editors, and better compilers), more interface and network bugs, and navigation bugs. There also tends to be more data bugs, and functional bugs on web platform applications, compared with mainframes. This is often because web Developers are less experienced than mainframe Developers, and as discussed before, inexperienced Developers tends to lead to more bugs.

Anecdotally, there are almost certainly less usability bugs in web platforms, but those usability bugs are often much more harshly judged. A usability bug on a mainframe is generally a lower severity and priority than exactly the same bug on an internally-facing only web application, used by exactly the same User base. Mainframe usability bugs are often more acceptable because that is the way the mainframe has always been. In short, the importance of a bug is not based solely on its isolated, tangible

attributes. The bigger picture often influences the way in which a given bug is perceived.

Models to Calculate the Number of Bugs in your Code

There are several models commonly used to calculate the number of bugs that may exist in code.

Function Point

Alan Albrecht is widely credited as having developed Function Point Analysis as a method to estimate software development costs. Soon after, Capers Jones introduced the concept of extending the Function Point model to establish the cost to fix bugs present in a piece of code. The fundamental approach to Function Point Analysis is to ask the following five questions, followed by an additional thirteen secondary questions (if required):

- The **inputs** to the application
- The **outputs** that leave the application
- The **logical files** maintained by the application
- The kinds of **inquiries** that the application supports
- The **interfaces** between the application and other applications

There have recently been several offshoots of Function Point models - Test Point Analysis, and Use Case Point Analysis. They use variations of the same underlying foundation. These models to calculate Points, and subsequently calculate the number of bugs, can be accurate, but must be undertaken with the due level of diligence. Getting it right requires a **great deal** of training and experience, and even then, is still very time-consuming.

Lines of Code

Boris Beizer¹ said there was a bug in every line of code that he ever wrote. That's a frank view expressed by a noted authority on both coding, and testing.

Most Developers would argue that there are far less than a one bug in every line of their code. As a result, there is much conjecture over what the right number of bugs per one hundred lines of code is. There is even conjecture over what one line of code is. Do you count every single physical line (LoC), including comments and blank lines; or only the source lines of code (SLoC); or if a logic statement in the code wraps over several lines, do you just count the logic statements (LLoC)? There may be some truth to the view that all of the conjecture is an attempt to avoid being measured against a standard? It is important to balance that view with the knowledge that the code is a product of the quality of the requirements and specifications supplied to Developers.

The reality is, it doesn't matter how you count lines of code, so long as the measure is applied **consistently** across all code under review. If you want to use lines of code as a measure, use whichever one is easiest for your organisation to apply consistently.

So, which model?

George Box, and William Deming are both reputed to have said, "All models are wrong; some models are useful"².

The truth is, a different model is required for every organisation. Which model depends very much on the organisation's skill at all of the tasks upstream of coding, such as the analysis and design. It

also depends on the Developer's skill and familiarity with the platform, software language, application, and modules in question, as well as the time that has been given to the Developer to complete the work. And, the profile of every project will be different, because the people involved and their skills change over time.

If you haven't

- (i) got a taxonomy of all of the bugs your organisation has found;
- (ii) undertaken a causal analysis of those bugs; and
- (iii) can predict where the bugs will be in your new code;

...then you need a model that will help you **begin** to understand your likely bug load while you build your taxonomy. One simplified model that broadly takes into account Developer skill levels, as well as the complexity of the solution, appears below. Whilst Beizer said that there was a bug in every line of code he ever wrote, there are some sensitivities to consider when it comes to modeling bug loads. Introducing a model that tells Developers everything they do is wrong is not going to win your Testers a lot of friends. It's recommended that you first try a less confronting model, such as the one below, which is based on multiples of two (2).

Bugs per 100 Lines of Code (LoC)				
	Complexity	Junior Developer	Intermediate Developer	Senior Developer
Create New Code	High	64	32	16
	Medium	32	16	8
	Low	16	8	4
Modify Existing Code	High	32	16	8
	Medium	16	8	4
	Low	8	4	2

This model is just something to use as a **starting point**. You will need to adjust the model as you learn more about your bugs at your organisation. Again, "All models are wrong; some models are useful".

Planit Bug Hunts

Ever increasing pressures on time and resources mean we need to ensure that we are getting **quality, trustworthy software**, and identifying "buggy" areas as early as possible. Planit Bug Hunts do not rely on pre-prepared Test Cases, or an in-depth knowledge of the system under Test to be successful. What matters is the skill and experience of the Testers in knowing the areas where bugs are likely to be hiding. Planit Bug Hunts can be targeted on specific areas of the software - if the whole system is not ready, the high risk Business critical processes can be targeted, or Bug Hunts can be used on requirements documentation, or Help Text / User Guides. In fact, a Bug Hunt can be carried out at any strategic point throughout the project.

Building or re-using existing bug taxonomies sets goals for testing on the most likely areas where bugs may exist, and can give early indications on the state of readiness of a system, allowing more accurate forecasting and estimation. Using exploratory techniques and session sheets allows for upfront planning and control of testing with the flexibility to target new areas, should bug clusters be discovered, as testing progresses.

The steps are minimal upfront preparation, including a short meeting with key Stakeholders, to identify major risk areas. From there, the sessions can be planned at a high level, and, adjustments to the sessions are encouraged through the learnings gathered during test execution. Once initial planning is done, the sessions are executed, and the results are best presented in the form of trends.

This is not meant to replace traditional testing that uses traceability back to requirements and acceptable coverage, but to complement it. A Planit Bug Hunt, using our bug taxonomies will give you early visibility of bugs, and allow for the focusing of traditional testing, to cut out redundant tests and enhance high risk areas, ultimately increasing Business confidence.

References

¹ 'Bug Taxonomy and Statistics' is an Appendix of "Software Testing Techniques (Second edition)"; Boris Beizer (Appendix amended by Otto Vinter); First published by Van Nostrand Reinhold in 1990; ISBN 0442206720.

² Quote from George Box as it appears in the book, "After the Gold Rush: Creating a True Profession of Software Engineering (DV - Best Practices)"; Steve C McConnell; First published by Microsoft Press in 1999; ISBN 0735608776.

About the Author

Adam Hughes is a Senior Test Consultant with Planit. His professional experience totals more than seventeen years at some of Australia's largest blue chip companies. Most of this time has been in IT, as both a Tester, and Business Analyst. He has found his passion in the Software Testing industry.