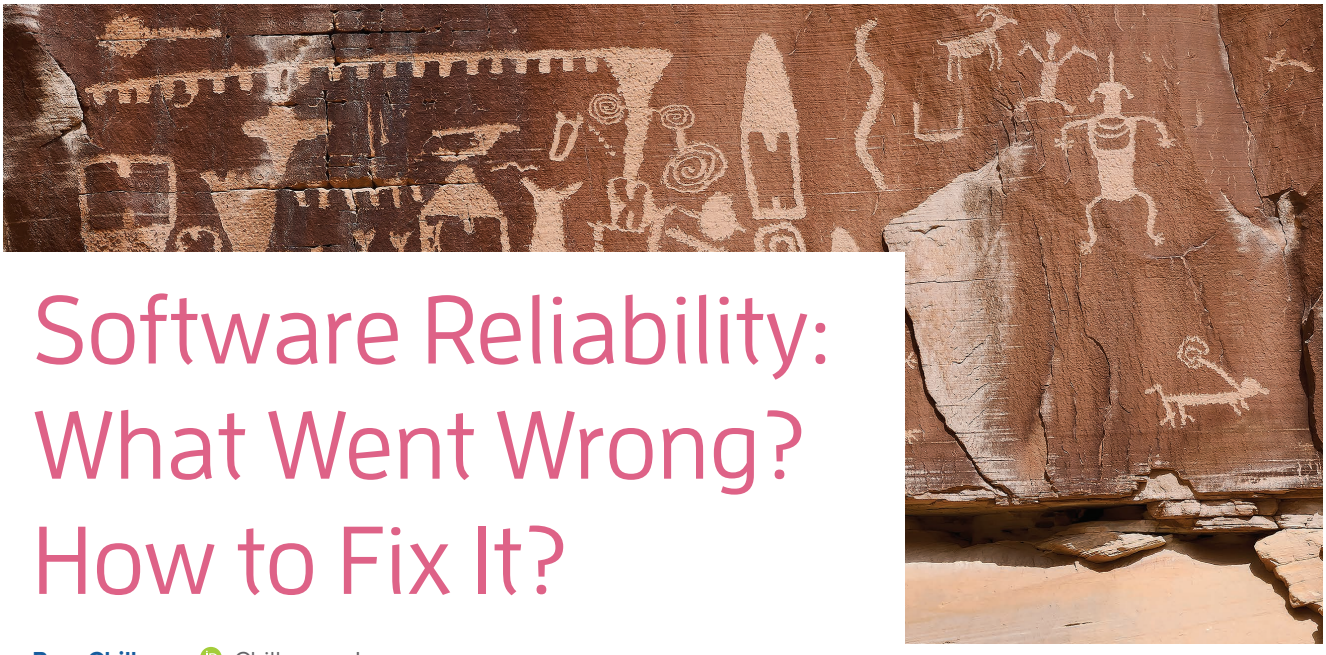

Software Reliability: What Went Wrong? How to Fix It?

Ram Chillarege, Chillarege, Inc.

Almost 50 years ago, software reliability was defined with a hardware mindset. While the research community grew, its industry acceptance was muted. Rethinking the definitions of failures and faults will usher meaningful research and create value to the practice.

From the average person to tech gurus, the phrase "software reliability" conjures up endless frustrations that plague the use of modern software. However, the academic community intended for a narrow definition of "software reliability": Failure of running software. This often alluded to the catastrophic stoppage of a program or service, measured by metrics such as the mean-time-between-failure. All good, when one considers that most industrial products are rated by similar metrics. But therein lies the issue. Just one metric? Software demands new notions of reliability to fully capture its behavior and remain meaningful, especially today.

Digital Object Identifier 10.1109/MC.2024.3431968
Date of current version: 21 October 2024

The notions of software reliability were developed over five decades ago. It evolved as a dual to its hardware cousin and soon found acceptance in the software testing community. But that is where it remained. With time, software reliability's importance waned since it primarily served as an assessment metric. Its ability to provide insight into the causes of poor reliability was superficial and certainly not actionable. To make things worse, software testing is the tail of an organization that is driven by head strong architects and developers whose priorities are dictated by the whims of the business. Today with 30 million software engineers worldwide, clamoring for every new incremental technology that can boost its productivity, software reliability remains on the sidelines.

Can this be changed? Most certainly, yes. Yes, if the software failure event is refined closer to the customer experience. And if the fault models capture the true nature of the corrective experience.

HARD → SOFT

The original definition of a software failure was simple: A program either worked correctly or incorrectly. It appeared

like the dual of a digital hardware failure. The simplicity had elegance but failed to grasp the complexities in software failure. This classical notion of software reliability should be called *HARD software reliability*. That would explain the difficulty that most practicing software engineers have with accepting the contributions in the area of software reliability.

The reality of software failure is far more nuanced. For example, an enterprise financial application could be considered working for thousands of customers, while a subset may complain of a failure if the foreign currency service feed is delayed by 5 min causing upheavals among a few brokers. Imagine, a group of new mobile customers of an application go viral and find that their registration process demands personal information that could be a legal issue in that nation. One asks, is that a software failure? Is that a functional software failure or would that be deemed nonfunctional since it is a security issue and not a functional capability?

The classical notions of software reliability do not have a language to express these issues that plague the software developer. Neither do they have the means to aggregate these customer and business frustrations. What makes this shortcoming even more glaring is that over 90% of software tickets an organization sees are software failures that may never require a line of code to fix the issue. Instead, they may require changes to feeds, data, configurations, access levels, documentation, or UI that is auto generated based on region.

As serious as this issue is, a little humor must not be misplaced: We have a HARD software reliability problem. And we need a truly SOFT software reliability fix. The rigidity of our notions of failure and fault could benefit from a softer touch. And from this, we will gain tremendous leverage in modeling and providing real value to the software development business.

SOFTWARE FAILURE

Let's first focus on failure and how it affects the customer. When software failure occurs, it delivers pain. It does not have to be catastrophic. It might merely be poor response time or difficulty of usage. That is still a failure as far as the customer sees it. It can also be quite complex to identify. For instance, the bank records may show a transaction with an incorrect value, but the account balance is fine. And it may take that one customer who checks their accounts to have noticed it. Another example may involve a Telco operator that is getting reports of dropped calls in a few zip codes while other customers in that very region are enjoying the streaming service with no glitches.

Software failure needs to be captured by the nature of pain and the degree of pain. I call these *impact* and *severity*. Severity is often captured on a scale of one to four. And impact is captured by around a dozen categories: reliability, performance, usability, integrity, security... and so forth. Just these two parameters allow us to describe, quite adequately, the consequence of the failure on the customer. The categories of impact can be grouped into functional and nonfunctional too. The only two functional categories are: capability and new-requirements. It might seem strange that a failure attribute of pain is marked as new-requirement. It is just the nature of our software business that quite often the service stream identifies new features through customer demand.

SOFTWARE FAULT

One of the definitions of a fault is that it is the cause of the failure. But what exactly is the fault? A fairly exact representation is possible if we ask what was changed to rectify the fault. The change could occur in different artifacts: code, data, design, or even documentation. The object touched is called the *target*. And more specifically the change made in that target is called the *defect type*. Examples of defect type for the code target are assignments, checking, algorithms, and so on. The defect types describes the semantics of the change in the space of the target. Data would have a different set of types as compared to code. The combination of target and type provide a fairly complete description of the software fault. Multiple targets are often involved in any one fix. For example, the fix for a failure may need a design change, a code change, and a data source change. So, this one failure would be ascribed an impact and severity, and the corresponding fault would have six other attributes: three targets each with its defect type.

As an interesting aside, this structure for faults nicely captures events that IBM service teams called "non-defects." Traditionally, development teams and management focused on the classical "defect" which required a code change. The nondefects got

> Today with 30 million software engineers worldwide, clamoring for every new incremental technology that can boost its productivity, software reliability remains on the sidelines.

left by the wayside. As time evolved, nondefects became 90% of the service tickets, and there just was not the data or analysis to gain insights to bring processes under control. Orthogonal defect classification (ODC) viewed all changes with a uniform abstraction, be they defect or nondefect. This uniformity allowed for root cause analysis of nondefects leveraging tools and practices that already existed for defects.

Here are a couple examples of a nondefects just to illustrate the concept. One can have a poor performance in a cluster of machines that provide a cloud service because a human configured the virtual machines incorrectly. Contrast that with an enterprise warehousing application where a series of customer orders were deemed incorrect because the UI prompted users in a manner that confused them. The code was correct, but the UI design was confusing. Both these instances required changes to correct the situation, but they were not code changes. In the first case, a manual process was reworked and hopefully documented to avoid a repeat failure. The failure would be tagged as nonfunctional, performance with the fault tagged with target of manual procedure. In the latter, UI designers need to get involved to ensure the help text provides greater clarity. While the impact is functional, the fault is tagged with target documentation.

Fault and failure data when captured by these attributes can lead to a great deal of understanding. While data is captured at the level of an individual defect, patterns emerge when looking across aggregates in time, process or customer groups. The ODC metrics are quite different from the single-dimension mean-time-between-failures metrics of the past. Relationships between groups of data and their patterns can be mapped to the underlying development processes yielding behavioral insights of skill groups.

> What makes this shortcoming even more glaring is that over 90% of software tickets an organization sees are software failures that may never require a line of code to fix the issue.

ODC

These ideas on the expanded view of software fault and failures evolved into a framework that I called Orthogonal Defect Classification (ODC).[a] At the core, it began with a clear articulation of software faults and failures providing clarity beyond what was understood in the '90s. Today, it has additional categories such as trigger, the force that activates faults, and source that subdivides code across legacy, technology, and platforms. The collection of categories are mapped into a framework across four principal dimensions that link cause and effect. ODC data can be captured across the life cycle and historical releases to create a gold mine of organizational patterns. There are specific techniques to analyze customer segments and link their pain points to the development process. The same data, sliced differently, allows one to study technology platforms and skill groups within the organization.

Our experience with ODC across hundreds of projects in different verticals: telco, retail, warehousing, industrial applications, real-time systems, embedded systems, and mobile has demonstrated the value and durability of this concept over the past 30 years.

Software engineering management is complex. And sophisticated tools are necessary to deal with the complexity to provide focused analysis to problems. While the classical hard software reliability struggled due to its narrow definition, softening of the failure and fault model got us started over four decades ago. ODC, which evolved from there, reframed how we need to think of these data and created a far broader framework of business and technical insight.

Software reliability that is true to its SOFT nature is better served when expressed in a language closer to what's experienced. The ODC terminology partitions the space of failures and faults into distinct event and action groups. Now, measures on these subspaces become meaningful to the customer and the developer. In today's world where security issues now dominate the software landscape, ODC provides a language to clearly articulate the consequence of events and the ramifications of remedial measures.

[a] Orthogonal defect classification: https://en.wikipedia.org/wiki/Orthogonal_defect_classification

RAM CHILLAREGE is a president of Chillarege, Inc., Raleigh, NC 27613 USA. Contact him at ram5@chillarege.com.