An unused analyzer is as useful as a broken one

ANDREA LEPORI

Analyzers, integral to elevating the dependability and efficiency of software, often face a challenge in widespread adoption. This article contends that, initially, the actual utilization of these tools outweighs their perfection. We explore the prerequisites for achieving widespread adoption, examining both human-centric factors and the technical strategies employed by industry giants like Google, Facebook, and Apple to surmount adoption barriers. Subsequently, the discussion shifts towards strategies for enhancing and expanding analyzers to encompass a broader spectrum of bug types, paving the way for a more resilient and adaptable approach to software development.

Analyzers play a crucial role in addressing the escalating demand for robust code bases in the realm of complex software development. The continuous evolution of computer systems necessitates the creation of increasingly intricate software to harness the capabilities of more powerful computers. The rising expectations of users contribute to the demand for programs and applications that push the boundaries of computer program capabilities. In contrast to the past, where computers were limited to a niche audience with knowledge of their internal workings, today's ubiquitous use extends even to those with limited technical proficiency. Simultaneously, an expanding portion of critical sectors, including banking, healthcare, and the food supply chain, relies on computer programs for efficient management. While this reliance is generally successful, occasional deviations from intended behavior can lead to catastrophic consequences, highlighting the fallibility of human-developed programs. Recognizing this challenge, a viable solution involves employing automated tools, such as analyzers, to identify and rectify errors in the development process. Analyzers emerge as essential components in enhancing the reliability and performance of computer programs, mitigating the potential for disastrous outcomes stemming from human error in the development phase.

Resistance among developers to adopt analyzers is evident. Like any novel tool, there is an initial learning curve to overcome. The immediate value of analyzers might not be readily apparent, especially for seasoned senior developers with extensive experience in the field. The prospect of integrating an analyzer into their workflow may seem burdensome, potentially accompanied by irrelevant advice in the form of false positives. In the subsequent paragraphs, I will delve into specific challenges that contribute to the friction experienced by developers in embracing analyzers.

Earning developers' trust involves minimizing false positives and providing valuable insights — a critical aspect emphasized by both Google and Facebook. Research suggests that a false positive rate exceeding 10% can erode developers' confidence in the analyzer. In such cases, warnings may be disregarded or treated less seriously, rendering the analyzer ineffective. To address this challenge, it is crucial to calibrate the false positive rate to a sufficiently low percentage, a strategy recognized by both Google and Facebook. While this adjustment may result in more false negatives, potentially missing some bugs, we argue that this trade-off is preferable to having an analyzer that is consistently ignored by the development community.

Providing prompt feedback is crucial in the effectiveness of an analyzer. Even if the analyzer is flawless with zero false positives, delayed feedback, diminishes its impact on the program's quality. The widely acknowledged principle that catching bugs early reduces fixing costs underscores the importance of timely feedback. Furthermore, as highlighted by Facebook, if the analyzer concludes its analysis after the developer has moved on to another task or project, it introduces substantial overhead and requires additional mental effort to revisit and address the previous project. This is not only understandable from a practical standpoint but also recognizes the potential demotivating effect and significant cognitive strain associated with revisiting and fixing a completed project.

Integrating comments directly into the development path is essential. Requiring developers to go out of their way to access analysis results may discourage the use of the analyzer. For instance, creating a new web app to display current analysis results involves opening the app and locating the relevant function or line — a process that adds unnecessary steps. On the contrary, incorporating the analysis tool seamlessly into the development workflow ensures that every developer encounters it naturally. This integration can occur at various points in the workflow, such as directly within the Integrated Development Environment (IDE), as demonstrated by Apple, provided the analyzer is sufficiently fast. However, this approach may limit users to a specific IDE, and if the analyzer's speed falls short, it can lead to a frustrating experience. Alternatively, the tool can be inserted into the build environment, following Google's example, or integrated into continuous integration processes, presenting results before merge requests.

Ensuring warnings are unignorable is paramount. Following the suggestions outlined in the preceding paragraphs should yield a robust analyzer. However, if a developer chooses to deliberately dismiss all analyzer results, it undermines the progress made. While this might be considered a contentious approach, elevating the status of warnings to errors could enhance code quality and consistency.

Enhancing and broadening the scope of analysis is the next step. With a practical analyzer in place, the opportunity arises for continual improvement. In a similar vein to how software engineers are tasked with writing test cases, the idea of crafting analysis patterns to expand the capabilities of the analyzer is worth exploring. Engaging the engineer intimately familiar with the codebase on a daily basis allows for the identification of potential bugs. Although not as straightforward as creating test cases, for large enterprises with extensive codebases, allocating resources to develop analysis patterns may prove beneficial. Equally critical is soliciting feedback from analyzer users, establishing mechanisms for reporting false positives, and tracking which bugs identified by the analyzer have been addressed. This feedback loop facilitates an assessment of the performance of each analysis module or pattern, ensuring that false positive rates are manageable. Neglecting this feedback loop risks reverting to a scenario where the analyzer is underutilized or becomes a source of frustration.

In conclusion, establishing robust foundations is essential for the initial development of an analyzer. Initially, the detection rate may be low, but the paramount concern is ensuring frequent utilization of the analyzer.