Editor in Chief: **Diomidis Spinellis**
Athens University of Economics and Business, dds@computer.org

Reflecting on Quality

Diomidis Spinellis

MONITORING ROADWORK quality differs completely from actually building roads. When the road is being built, you take samples of materials and test them in a lab. When it's ready, you use specialized equipment to look for slippery or uneven pavements. And when the road is opened to traffic, you set up cameras and other sensors to see how it's used. In software engineering, we have it much easier because we can monitor how we build software (the process), the software we build (the product), and the product's actual use, simply with yet more software.

There's a deep reason why software systems are reflective—why software is used to build and analyze them. Large software systems form the most complex artifacts designed and built by humans. Managing this complexity requires tools of matching capabilities, so these are necessarily also software. Examples include compilers, runtime libraries, version control systems, issue trackers, application servers, and OSs. Without these the modern software industry would grind to a halt.

The Process, the Product, and the Product's Use

The tools that manage the development process provide ample opportunities to monitor its quality. Every code commit is a heartbeat that can trigger static analysis (for instance, in the form of style checks and code smell detectors); unit, integration, and regression tests; and, inevitably, test coverage analysis. The test results identify possible areas of concern in three dimensions: product functionality, software modules, and developer teams. Feature requests and bug reports on the project's issue tracker let us assess daily progress and, again, pinpoint problem areas.

IDEs provide finer-granularity data on how software is developed before a change matures for an eventual commit. This data can describe crashes, automated refactorings, and newly created entities. Logs of online code reviews reveal the details of caught (and missed) snafus. We can even apply sentiment analysis and other natural-language-processing techniques on the project's email lists, forums, and chat logs to improve our understanding of the developers' performance.

When the software runs, we can either have it instrumented to blab about its quality or apply other tools to it to make it talk involuntarily, as it were. An important element of internal instrumentation is assertion statements: logic fuses that blow when our assumptions about the program state no longer hold. Logging instrumentation, typically implemented through purpose-built libraries and frameworks, can provide extensive details about what's happening in a system, thus letting us reason more deeply about possible problems.

We can also apply software tools that probe or slightly modify the software's internal workings to give us data regarding its functioning. With CPU-profiling tools, we can find where the code spends most of its time, memory profiling lets us see where memory is allocated and leaks, and tracing tools show us library and OS interactions. More intrusive tools help us locate out-of-bounds memory accesses, parallelism bugs, or security vulnerabilities. And when things go south, we can collect and process the details of crash reports, such as the stack trace and the software's log up until the crash.

Finally, we can monitor how our customers actually use the software. We can easily do this externally—for example, by looking at web-server logs or key presses. However, we can obtain much better results if we instrument the software to log its use: invocations, input and output data, button clicks, command exe-

IEEE Software Mission Statement

To be the best source of reliable, useful, peer-reviewed information for leading software practitioners—the developers and managers who want to keep up with rapid technology change.

 0740-7459/16/$33.00 © 2016 IEEE

cutions, latency, and throughput. If our software provides a cloud-based service, all we need is some additional logging. Otherwise, our software must ship the corresponding data back to the mothership over the Internet. Increasingly, software also gives its users an explicit say by letting them vote on feature requests or prompting them to fill out satisfaction surveys.

Explicitly designing our development process, our product, and its use to generate precisely the data we need reduces the collection effort and improves the data's quality. This can involve trivial adjustments, such as configuring the format and retention of log files, or larger-scale software instrumentation initiatives. Invariably, well-designed software and processes are also easier to monitor. For example, in one case I could obtain precise usage data from a desktop application because its hundreds of diverse commands were all uniquely identified with a mnemonic string and were dispatched from a single central point.

Exploiting the Data

With so much data reflecting the software's quality readily available, flying blind is inexcusable. When managing a software business, we must ensure that the types of data I outlined are generated, collected, and, more important, used. At a minimum, their widespread availability throughout an organization (subject to appropriate confidentiality safeguards) can help all stakeholders generate the intelligence they require. Other organizations might deliberately institute detailed monitoring procedures for collecting data, triggers that get pulled when something goes wrong, and corrective actions to fix problems. Dashboards, alarms, and periodic reports help us access the data when needed. If all this sounds like a tall order, there are also companies that collect and analyze software data as a service.

Data-driven quality management enables the efficient allocation of finite (and perennially constrained) resources. Recently, while going over a software product's crash logs, I found that just two easily fixed crashes caused more than 20 percent of 1,200 collected crash reports. Other areas in which we can utilize the collected data include feature selection, software performance optimization, team allocation, development process tuning, bug triaging, software evolution planning, hardware allocation, and marketing-channel selection. Significantly, we can expect that our actions' results will later show up in the data we collect, thus giving us feedback on whether we're going in the correct direction.

For extra points, we can look for opportunities arising from integrating process, product, and usage data. For example, consider driving profile-based optimization from actual usage data rather than synthetic benchmarks. Or, we can investigate how software crashes map back to the static analysis or code reviews of the corresponding change.

In the future, when the world is an Internet of Things running on software, road builders and other engineers will have at their disposal the wealth of data we developers take for granted. At that point, we'll have to tell our engineering colleagues how over time we learned to use data to build the quality software our society deserves.