What Happened to Software Metrics?

J. Voas and R. Kuhn
National Institute of Standards and Technology (NIST)

In the 1980s, the software community was all 'a buzz' with seemingly endless 'potential' approaches for producing higher quality software. At the forefront of that was the field of software metrics, along with the corresponding testing techniques, tools, and process improvement schemes that relied on the software metrics. Later, there were also suggestions of legal remedies such as Uniform Computer Information Transactions Act (UCITA) and the licensing of software engineers as professional engineers. UCITA would have made software vendors liable for defective software and the licensing of software engineers would have allowed developers to be personally sued. Further, publications such as the *Software Quality Journal* were launched, and events such as the Annual Workshop on Software Metrics in Oregon were held for many years. Cyclomatic complexity, the Halstead metrics, Source lines of code (SLOC), Fagan Inspections, counting defects, predicting numbers of defects, reliability estimation and modeling, and many other metric-oriented ideas were floated as solutions to what was considered at that time as a software quality 'quagmire' that had to be eradicated.

The 1980s and 1990s were also times when software testing tools were numerous, often barely usable, and generally poor in quality themselves. This was a time when general-purpose software, whether produced for a niche market or mass market, was rich in defects and further offered little in terms of interoperability. And don't forget the musings back then that paying customers were simply beta testers with no way to opt-out. Looking back, promises of quality improvement in your software, "if you just do *X*", were commonplace and unfounded. Many competing ideas lacked usability, evidence of success, and a sound scientific underpinning.

Today, you no longer hear of most of those metrics-based technologies and their associated promises. Therefore, this virtual roundtable is, in part, a walk down memory lane. Our position here is not cynicism or partisanship, but rather seeking a deeper understanding of what happened, what went wrong (and right), and what survived and is still used today. Further, is there an opportunity for better metrics or hybrid metrics that leverage past metrics research[1]?

We asked a panel of 7 software metrics experts 11 questions to help explain the last 40 years of software measurement and where they believe we stand today. Our experts are: (1) Taghi Khoshgoftaar (Florida Atlantic University), (2) Edward F. Miller (Software Research, Inc.), (3) Vic Basili (University of Maryland, retired), (4) Jim Bieman (Colorado State University), (5) Ram Chillarege (Chillarege, Inc.), (6) Adam Porter (Fraunhofer Institute), and (7) Alain Abran (University of Quebec). We did not ask rhetorical questions, but rather questions that we believe remain unanswered, and if answered, could form a foundation for improved or new software metrics. We are strong supporters of software measurement, but we are equally firm believers in the need for solid evidence of benefits, and not simply anecdotes of successes for a particular metric or quality improvement approach.

---

[1] J. Voas and P. Laplante. "A Perspective on Standard Confusion and Harmonization", *IEEE Computer*, July, 2007

1. **If you could only recommend one static software metric and one dynamic software metric, what would they be, and why?**

Abran: In most fields of knowledge based on quantitative information, such as accounting, finance, engineering and medicine, a very large number of quantitative ratios (or other formulae) are recommended for various contexts and purposes; nobody in these fields would expect a single measure or quantitative formula to be sufficient for analysis and decision making. All of these industries have invested considerably in defining very strict standards for basic measures and their various combinations, as well as in data collection and analysis to establish multi-dimensional industry benchmarks against which to compare.

The software industry, by contrast, has mostly very unrealistic expectations that poorly defined 'metrics' can provide solutions to complex problems at almost zero cost. This is wishful thinking. From a more down-to-earth perspective, I recommend not one specific metric but a full set of measurement standards, as documented and recommended by the non-profit International Software Benchmarking Standards Group (see www.isbsg.org).

Bieman: There is no one static or dynamic metric for all organizations or purposes. Any recommendation for a measure depends on context. To answer this question for a particular organization, you need to know the goals for measurement and what questions that you want to answer.

For static metrics: If you need to know how much software you have, a software size metric is appropriate (see my answer to question 2). If you need to know something about design structure, there are numerous ways to measure code properties like coupling, cohesion, "complexity", etc. If you need to know how testable your system is, you can statically measure how many specified "test requirements" are contained in your system. For example, knowing the number of statements or branches can indicate the difficulty of achieving a particular level of statement or branch coverage during testing.

For dynamic metrics: Run time performance (time and space requirements) are clearly important for many applications. Another important and useful dynamic metric is the test coverage achieved for specified test criteria. Finally, the most important dynamic measure is the number and frequency of defects discovered (or failures reported) in a system after release.

Basili: If you asked if I could recommend one physics metric what it would be? Is it mass, energy? You would immediately tell me it is a ridiculous question. You should select measures based on what it is you want to know and what you are going to do with that information. The Goal Question Metric Approach (1984) set out to identify the relevant metrics by defining your specific goals for the measurement. These goals suggested the kinds of questions or models you want to use, and these define the metrics you need. The models you select provide the framework for interpreting the metrics.

Defining goals involves specifying the objects you are measuring, e.g. a product, process, a model, the focus of interest, e.g., cost, defect removal, change, reliability, user friendliness, the

purpose, e.g., to characterize, analyze, evaluate, predict, the perspective of the person wanting the information, e.g., the manager, developer, organization, and the context, e.g., the organization's characteristics and context variables. All of these help define what measures you need and how you are going to interpret them.

Chillarege: It is hard to find commercial software organizations that have good metrics which are regularly measured and reviewed. When you do find them, the two most commonly recognized and understood metrics are the source lines of code and complexity. All said and done, there is a greater understanding of source lines of code in spite of the high variance that they display among programming languages. To a lesser degree complexity is understood. If there are just two that I am asked to recommend, these would be the two.

Khoshgoftaar: Recommending one static or one dynamic software metric is akin to suggesting a one-size fits-all solution, which is impossible in software engineering. Software systems development and software engineering measurements have both evolved dramatically in the past two decades, emphasizing a multi-faceted focal points of critical importance. Instead of focusing on a one-size fits-all software metric, we should expand our knowledge on intelligent methods for data wrangling and feature engineering toward best-exploiting the scores of auto- and expert-defined software metrics recorded by data collection tools.

Miller:

Static: On the static side, the general understanding is that the more complex a piece of software is, the harder it is to get right. So first off, I'd choose code size...using source lines of code (SLOC) is probably the simplest way to think about things. Simple as it is, the value of this is limited. I've seen VERY complex chunks of code that are very solid and reliable. And I have seen some collections of little components that you would think would work out of the box but which fail miserably when put through a test suite.

Dynamic: On the dynamic side of the question, if you're concerned about end-user quality, then test coverage metrics are the way to go. In the 1990's there was a LOT of discussion about which coverage metric was best. We recommended the "branch coverage" metric, but there were many fans of statement coverage, which was a lot easier to measure.

Porter: It really depends on what you want to use the metrics for. If you asked a construction worker to name their two most useful tools, they might think of tape measures and carpenter's levels as being indispensable for some jobs, but they'd swear by plumb lines and speed squares for other jobs. The job defines which tools are right, not the other way around.

Similarly, organizations don't have to just take the metrics that someone else defines. They can create their own. In many cases, that's a better way to go, because the user of the metrics knows best what they are trying to understand and do with the metric. Collecting data just because it is available does not yield insights. You must have a goal in mind to improve or understand your software development in some way, then define the data you want to collect based on that.

Having said this though, I find that counting Source Lines of Code often provides a valuable, easy to compute static metric for volume of work to be done. Similarly, I often look at line coverage percentage as a simple, easy to understand dynamic metric of testing effort.

2. **There was once a common belief that all static code metrics essentially boiled down to Source Lines of Code (SLOC). Was that true? If so, is it still true. If not true, why?**

Abran: The research findings from the late 1970s and early 1980s did indeed point to the overall conclusion that the various static code metrics had very strong dependencies on SLOC. Not much more research has been conducted since then to negate that conclusion. Personally, I am not a big fan of metrics based on SLOC because they are too dependent on technologies (e.g., programming languages, programming styles and local coding standards) and their different implementations by tool vendors or researchers, thereby inhibiting the reproducibility and interpretation of values from analysis models across technologies, tools and contexts.

Bieman: Many questions can only be answered if you know how much code there is in a software system, subsystem, or version. The short answer to the question is "yes", the most useful static software metric still is *the number of source lines of code (LOC)*. A key advantage of LOC is that developers generally understand how LOC is measured, and it intuitively indicates how much source code is contained in a method, class, function, etc. LOC is regularly used as the denominator in derived measures such as *defects discovered per KLOC.*

Of course, there are many limitations to LOC. Different programming and layout styles, as well as different counting protocols (do we count comments, declarations, etc.) can affect LOC.

Basili: Certainly SLOC is a reasonable static metric if you want to know how big something is, but of course it depends on context and purpose. Why are you using the metric? Are you using it to characterize your products? If so make sure you put like things in the same bucket, i.e., you have to make sure the context is the same, e.g., the programing language, possibly the application domain, etc. If you are within the same context, are you using it to evaluate, e.g., what process give the smallest product or to predict, e.g., what will most likely be the amount of resources needed to build the new product?

Chillarege: For a long time the function point community maintained a steady following among practitioners. However, the function point definition works best for classical business applications and poorly for many other scenarios. Regardless, it was the business app community that endorsed it and a successful practice for a long time. They are manual to capture and limited in applicability. Gradually they faded away. In addition, the back-firing tables that convert function points to source lines of code, always made me wonder, why using one would be much different from the other.

Thus, source lines of code, for all their perceived faults remain the core size metric of the day. Most current static code analyzers spit out the number. Thus, it is more visible in today's Agile teams, although the practice of using the metric successfully in projects may or may not be there.

Khoshgoftaar: Studies have shown the defect prediction capability of static code metrics, including SLOC. During the earlier times of software metrics research, limited availability of software metrics data and/or lack of good data collection tools influenced the general direction of research. But to say all static code metrics were being essentially equated to SLOC is not true in my opinion. However, one could argue that SLOC are more related to some similarly-simple metrics, such as Basic Halstead metrics, and there have been case studies showing different predictive powers of SLOC and complexity-based metrics, such as Cyclomatic Complexity. The answer lies in feature engineering with software metrics, as well as examining correlation between software metrics.

Miller: As far as I could see, SLOC was highly correlated with every other metric. Here I'm thinking of the Halstead metrics. So if they were all correlated, why not just use the simplest one to measure?

Source code obfuscation creates a lot of problems. For example, in JavaScript so much is lost in removal of the context that is contained in the comments and the source expansion tricks didn't work well. At the end of the day, SLOC came to dominate people's thinking.

Porter: In the late 1980s and early 1990s, a lot of software metrics research was focused on defining metrics that could assess the quality of existing software and/or predict quantities such as expected development effort or the number of latent faults in a code base. Comparative studies of these different metrics, however, generally failed to show that these metrics were significantly and repeatedly better than using just lines of code. In this sense, it's reasonable to say that all these metrics were no better than just using lines of code.

However, metrics can be defined over lots of different software development artifacts, available at different times, and used for lots of different purposes. So simply saying all static metrics boil down lines of code is too simplistic.

3. **Back then many organizations were sold on the idea of process metrics such as the Capability Maturity Model (CMM). The US DoD invested heavily in that idea, and some have argued that this added significant financial burdens to military IT and software systems. Did it work? And where is CMM today?**

Abran: Organizations without well-managed processes are unpredictable in terms of cost, duration, quality, functionality delivered, etc. All of these uncertainties lead to very poor quality, very high costs often due to extensive reworking within projects, and considerable waste when projects fail.

Process improvement models have been designed and adopted primarily to manage the risks and uncertainties associated with out-of-control development processes. Organizations that properly manage their software processes achieve significantly more predictability and reduce their overall project risks and costs. The organizations I have observed that have implemented these management concepts are successful and well managed, whether or not they had adopted the CMM model.

Bieman: A number of government agencies and companies require organizations to achieve specified CMMI levels before they can bid on a software development project. A CMMI evaluation makes the development process visible through the measurement of numerous process attributes. According to the CMMI Institute (http://cmmiinstitute.com/), organizations in 98 countries make use of CMMI. Approximately 14,000 CMMI appraisals were conducted during the past 10 years. Most (76%) of the appraised groups had fewer than 100 employees, and more than 70% of the appraised organizations use an agile development process. The number of appraisals has been increasing at a rate of nearly 20% per year, with the greatest increases in China, United States, India, and Mexico.

Chillarege: Watts Humphrey explained to me that "the software engineering metrics were not at a point where they could measure the quality of software that was acquired by the government. And thus, contractually there was no realistic way to enforce an acceptable criteria for software. Therefore, he strongly felt that the only way forwards was to ensure that the processes of the suppliers was acceptable. Which, in turn would result in good software being delivered.

At its core, the CMM is predicated on the premise that the process is far more measurable and controllable than the work product – namely, the software code. This set in motion the management of software for the next couple decades. India, in the late 90's, aspired to get into the software business, and the CMM provided and excellent vehicle to systematically gain process skills and establish credibility in the market. The interest in the US was muted, for a variety of reasons.

Software being an intellectual activity defied many of the classical techniques of process control used in manufacturing. There have always been attempts to make a conceptual mapping, and the few positive results amplified. Yet, although the premise was mostly unproven, it gave management a clear framework to direct work and a ready assessment of achievement.

Basili: The concept of capability maturity was based upon the original idea of Philip Crosby and was used to assess the maturity of an organization. He never meant it as a prescription for building a mature organization but as a mechanism for finding out weaknesses in the organization. So the goal was not to keep adding process until you get to level 3 and then start dropping or refining them. That process takes you through too many culture changes and can get quite expensive before you shrink process back at level five.

Khoshgoftaar: The original CMM, formulated by SEI, and its successors have been put to pasture since their initial introduction and use. This was largely due to lack of deep integration of CMM into an organization's processes. Investment by US DoD did lead to acceptance of CMM by large military defense contractors. However, in many cases significant maturity success was achieved only after incorporating CMMI, the CMM Integrated project approach. One could deduce that CMMI was a result of lessons learned from the stand-alone practice of CMM by organizations.

Miller: Yes, but only indirectly. There were many papers at QW/QWE that pivoted off the basic CMM idea, bending it to fit into the needs of "quality assurance and testing" organizations. The key notion was "maturity" -- both of the internal process used and of the technical maturity of the team of programmers/developers/testers involved.

Going back to the day of Harlan Mills' "Chief Programmer Teams" from the early 1970's, everyone pretty clearly understood that democracy in programming work was no virtue. But not everyone could fill the shoes of a Mills-like "chief programmer" and the compromise seems to have been to develop metrics for the entire team. Which then led to the CMM and all that followed it. Why it worked is, to me, pretty simple: it forced people to think in process terms and pay attention to outcomes.

Porter: The CMM is not a set of process metrics, but rather a set of key processes areas that, when implemented effectively, should help companies improve the quality of their software while controlling cost. One of the goals of a CMM organization is to define and implement process metrics that capture quality drivers specific to that organization.

CMM is based on well-studied notions of statistical process control and continuous process improvement. The general idea was that if you can measure a process, then you may be able to repeat it. If you can repeat it, then you may be able to improve it. If you can improve it, then you may be able to fine tune it, and so on. In essence, first get consistency and control, then you'll be able to go for improvement.

The CMM framework assumed that in order to go through these steps, organizations needed certain capabilities, such as configuration management, etc. This makes a lot of sense and I've seen companies improve vastly by working through this framework.

However, none of this meant that a company with a given level of capabilities would always and automatically produce a better product, faster, and at lower cost than companies with lower CMM levels. It really depends on what the company is doing with these capabilities and whether and how fast the underlying development requirements were changing. Once specific CMM levels became prerequisites for getting contracts with DoD, some companies were only interested in getting the credential, not interested in using their capabilities to get better. Additionally, the process itself became more heavyweight, hard to adjust, document-focused, and costly, ultimately becoming less cost-effective for many practitioners.

4. **The software metrics of the early 1990s were mainly static, however the behavior of software is dynamic. Do we have newer static metrics that better reveal software behavior and semantics than only software syntax?**

Abran: I am very puzzled by this view of static versus dynamic metrics of software code, as if coding was the only software development artifact to monitor and control. For instance, the quality and size of requirements is the foremost artifact underlying the whole development process; ambiguous and incomplete requirements specifications lead to major problems, including continuous reworking throughout all subsequent development stages, improper planning and monitoring and, of course, incomplete or inaccurate definitions of testing

artifacts. I have not seen significant advances in SLOC-related metrics since the early 1990s; however, there have been significant advances in requirements specification and architectural measurement that can be extended throughout the full lifecycle to ensure traceability in later project phases and normalization of various technology-dependent ratios.

Bieman: I am often interested in the design structure at an intermediate level of abstraction. For example, you can analyze a software design (and implementation) in terms of the existence and number of realizations of various design patterns and the connection between design pattern realizations. Another measure that can be very useful in analyzing the *testability* of a system is to count the number of test requirements that must be covered by test cases to achieve particular test coverage criteria. We can also understand more about a design by categorizing and counting design pattern realizations.

Basili: There were lots of dynamic metrics in the 1990s, e.g., reliability, performance. It is not clear a static metric can provide insight into the dynamic behavior of software, unless you look at the variation of that metric over time. Reliability and performance metrics are in common use in many organizations, e.g., look at the more recent work of Elaine Weyuker and Tom Ostrand at their work applying reliability models at AT&T.

Khoshgoftaar: Software development is a complex process, with many variable attributes including development methodology and project objectives. Therefore, it is difficult to determine a consistently good software metric to predicting software behavior. Case studies have shown SLOC and other simpler metrics are good defect predictors for some project, while not so for other projects. Likewise, newer metrics, both static and dynamic, are shown to be of varying effectiveness at predicting software behavior for different projects. Current focus on software metrics has tended to a combination of syntax descriptors and dynamic software attributes. In general, the novelty of newer static metrics will vary from expert to expert; however, the discussion should also include feature selection for optimal metrics' selection based on modeling goals.

Miller: A lot of research effort in the software testing community has been put into trying to extrapolate beyond the structural metrics, but I've not seen much that really reveals anything particularly valuable in terms of predicting trouble spots.

In a different arena, you will find that there are bunches of patents and patent applications dealing with manipulation of a webpage DOM and extraction of user oriented metrics that can be extracted from delivered web pages. Which is very neat and sophisticated, even if the importance and application are not fully clear. Even so, there are some very big companies that collect such things as "DOM settling time" from remote machines all over the world in spite of the fact that that's not really a significance performance bottleneck for all but the hairiest web pages.

Porter: One very interesting trend in modern software development is model-driven software engineering. Models are increasingly being used, especially for embedded, cyberphysical systems, to specify requirements, analyze prototype implementations, and even generate system code.

Metrics defined on models rather than on source code are currently being developed. These metrics have many desirable properties. For instance, they are defined at the requirements or behavioral level which is often more understandable to the end customer than source code/implementation level metrics are.

5. **Structural metrics measuring dynamic behavior have been around for decades. The most commonly cited are statement coverage, branch coverage, and modified condition decision coverage, plus a few module-level coverage metrics for object oriented code. What percentage of developers in your industry or profession that use one or more of these metrics? Are there other dynamic metrics that are used?**

Bieman: I don't have concrete, quantitative information concerning the use of coverage tools in industry. Anecdotal evidence from discussions with industry practitioners and the wide availability of coverage tools suggests that the coverage achieved during testing is commonly measured.

Basili: Coverage metrics have been in use for decades and have been refined for newer development paradigms and languages such as object-oriented design. There primary use has been to identify the quality of the tests that have been run. Of more importance is requirement coverage, to assure that all requirements are appropriately covered and checking the requirements coverage vis-à-vis the various code coverage metrics. Coverage metrics don't measure the dynamic behavior of the software product but the quality of the test suite. They are still being used quite commonly to cover unit test as well as system test.

Khoshgoftaar: A look at the PROMISE software project repository provides a good indication of the large extent of organizational use of structural code metrics to model software project behaviors, including dynamic. Researchers have used execution-based metrics, such as computational time, to model dynamic behavior. More recent studies have categorized dynamic software metrics, as cohesion-based dynamic metrics, coupling-based dynamic metrics, and execution traces' based metrics. Some studies have shown the superior power of such metric over traditional structural metrics for different predictive studies.

Miller: This brings to mind the software coverage metrics war -- begun in the 1970's (I had a part in that) and continuing into the early 1990s. Statement coverage was easy to measure, but gave you a false sense of security. MCDC was harder to achieve, and because of that got far less traction. The harder coverage measure, path or verification condition coverage, was very hard to measure, and got almost no traction.

The discussions were fascinating, building the tools to make the measurements was exciting, but in reality only a small number of developers, it seems to me, ever had the resources to use these tools they way they were intended. Besides, when a budget crunch hit, coverage testing was one of the first steps to toss out. So, probably overall usage of test coverage metrics was < 1%, very sad to say.

But it may be worth mentioning the modern practice of delivering a "new version" of a product -- possibly fully instrumented as well -- to a subset of your user community and waiting for the complaints to roll in. A kind of "involuntary crowd-testing" process. It's sneaky, but very effective at ironing out goof-ups inexpensively!

Porter: While I don't have a well-validated percentage to report, I would say that the use of simple test coverage metrics has increased substantially in recent years. One reason for this is that use of automated testing tools and environments have exploded in the last decade. It's increasingly easy to build large test suites, execute those test suites and capture test coverage information as a nearly free by-product. However, coverage usually isn't sufficient -- you need to evaluate the quality and quantity of the test cases as well. A single test case that executes all the lines of a system is not very useful.

6. **Software reliability modeling and theory played a role then and now. What is the state of software reliability models today; specifically, what percentage of developers in your industry or profession that use reliability modeling? And is there one or two you recommend over others?**

Bieman: I don't have good information about the use of software reliability models. Actually, I don't know that they are commonly used in most organizations.

Basili: Then and now the most effective use of reliability measurement is when the system is operational and to predict how the system will perform in practice. See my answer to question 4.

Chillarege: Software reliability is probably one of the more extensive uses of software metrics. This is "software reliability" in the broadest possible definition and not the specific academic definition of the term. By that token, defect rates, backlog, closer time, customer satisfaction, first time fix, re-create, criticality, pervasiveness, trigger, etc. are all terms that would come under the umbrella of software reliability.

Some of these terms and measures are commonplace in the industry, whereas they may still be unheard of in academic articles. On the other hand, there are numerous academic articles that discuss various nuances of software reliability that will be foreign to the most experienced software engineers in industry. This is a chasm that is not unusual, but one that has been bridged, just barely, in the past 20 years. As a consequence, the industry hobbles along without being able to leverage a fairly large community of academic researchers. In spite of this, if one takes this broader perspective, software reliability metrics are probably the most widely used software engineering metrics. Far more than the metrics that have to do with size, complexity or productivity.

Miller: John Musa's work was seminal in this area, but there is that nagging issue about it that there is no "wear out" phenomena to drive the model. That always struck me as a fundamental stumbling block. Without some underlying "theory", a statistical analysis is meaningful only for one methodology and one team. Change anything and the numbers could go anywhere.

It was not something we ever put any stock in because we always fixed (or at least documented) every error we could find, as fast as we could. Zero outstanding critical errors was the continuous goal.

Porter: Back in the 80s and early 90s software reliability growth models were heavily investigated. In more recent years, however, there's been relatively little new reliability research appearing in the main academic software engineering conferences.

However, as cluster and grid computing models became more popular in the late 90s and early 2000s, practical measurements and applications of reliability (and availability) metrics were and continue to be used and improved. Descendants of these concepts are used in today's cloud computing infrastructures.

As far as recommending reliability models, you should fit the model to the data, not the other way around. Unfortunately, there is a lack of simple, out-of-the-box reliability modeling software packages for software developers to experiment with. Reliability modeling also generally requires that your testing environment accurately reflects your operational environment, which is difficult or impossible to do in many cases (think of cloud-based services or mobile computing).

7. **Software testing techniques and tools are often based on metrics, such as SLOC, code complexity, logic complexity, etc. What do you see as the relationship(s) today between metrics and testing?**

Abran: Metrics per se are only inputs into quantitative models looking for relationships across a number of variables. The challenge is that such relationships have been inadequately investigated to figure out which threshold values are meaningful in various contexts, including the very specific context of the software programs being tested.

Most of these code complexity and logic complexity metrics correspond to algorithms that capture only some of the targeted aspects—none of which directly represent what needs to be tested. By contrast any functionality measured by a Function Points method represents what functions must be tested under various sets of conditions; therefore, the identification of these for measurement purposes can be reused directly as functional scenarios for testing purposes from both the developer and user perspectives and their quantitative information can be used for various analyses.

Bieman: A developer can use one of the readily available coverage tools to determine whether coverage goals are met. However, testers know that their goal is not to achieve 100% coverage. Rather it is to find 100% of the faults. Unfortunately, no tool can tell you that.

Basili: I believe the most common metrics for testing are coverage metrics.

Chillarege: Software testing can potentially be one of the beneficiaries of good metrics. Especially given the numerous research ideas on methods to better test software. However,

this is hardly the case in industry. The testing community at best has traceability between stated requirements and test cases. Statement coverage is the next step-up, on rare occasions. Anything beyond that is unusual.

To put it in perspective, one needs to recognize that software testing continues to be one of the least advanced methods in the software development process. The product groups are most often better than their cousins in IT. Most testing is manual. Test automation tends to be the high watermark for many organizations. While there is a broad recognition of the value of automated testing, its penetraiton in the practice is still relatively low. It is also the case, that building a completely automated test environment is non-trivial. DevOps and Agile have paved the way to encourage CICD and with it a focus on automation. The good news is that it has picked up a lot more these past couple years that it has in the last decade. .

The software testing services that are sold are often time and materials contracts. And most testing vendors are reluctant to automate since they perceive it as a net loss of revenue. The leading edge vendors do take a longer term perspective and see automated testing as a long term win-win.

Khoshgoftaar: Software testing techniques and tools are not limited to guidance provided by different software metrics, including static code metrics and dynamic metrics. It is known that the software testing phase often suffers due to compressed deployment time frames, prompting the output of metrics-based predictive models to guide software testing. However, considerable portion of testing is also guided by test cases' planning and test case code coverage. The criticality of the project influences its software testing emphasis. But in today's general agile development environment software development and testing are iterated in a compressed time frame. Towards that the emphasis on guidance by metrics on software testing and testing tools tends to become higher.

Miller: The are some more modern metrics, oriented to web pages, that I've noticed. One of these is a "heat map" generated based on users' recorded GUI activity on a web page front. At least a couple of vendors are offering heat maps based on data consolidated across many users, in some cases even without the users' permissions. What is attractive is that you get a cleaner picture of what the users think is important. With that kind of data, you know better where to focus testing -- right where the users really were looking.

But thinking historically, I have to admit I'm a skeptic about whether actual testing was or wasn't guided by any metric other than "what's important right now." I mean, test teams focused on the latest additions to an application -- rightly enough -- but I don't recall any teams that systematically measured and then tested in response to a metric of any kind.

Porter: Software is designed to execute in particular ways. Software testing metrics try to capture how many of those ways are exercised by the testing process. Popular testing metrics do a reasonable and generally cost-effective job of helping developers understand how thoroughly they are testing their software. I would also point out that rather than viewing 100% coverage as an overriding goal, developers often use coverage information to point out where their test suites are inadequate. For this reason and because even complex code coverage metrics can be

prohibitively expensive to collect (especially for very large systems), lighter-weight dynamic test coverage metrics will an interesting research topic in the near future.

8. **Process improvement was meant to suggest that a better process and better organization would produce better software. Did that ultimately occur, and can you suggest examples?**

Abran: In organizations where I have seen sound and continuous process improvement, I have observed considerable improvement in the developer's credibility, from all perspectives: quantity of functional requirements delivered (quantified objectively using ISO recognized measurement methods), quality delivered and predictability, as well as significant reductions in the number of failed projects, i.e., projects are abandoned in a timely manner where appropriate. In addition, I have noted higher maturity levels (also leading to a better understanding of process capability) and more realistic expectations (instead of inflated claims of delivery within an impossible schedule and unrealistic budgets).

Bieman: I believe that paying careful attention to the development process and organization will lead to better software. Currently, many, if not most, software development organizations are using some form of an agile process (e.g., Scrum). Following a well-defined process can only improve the quality an always evolving process produces.

Basili: Sure it has. The best example I have is the work in the NASA Software Engineering Laboratory in the 1980's and 1990's where we were able to show how various methods were able to reduce cost and improve quality (as measured in resources expended and defects delivered). The improvement came from evolving the processes to meet the particular context based upon measurement and feedback. More recently, look at the work of Lionel Briand and his Software Validation and Verification Laboratory.

Chillarege: When process improvements are successfully implemented the gains are phenomenal. But the instances where there is continuous improvement are rare. In our work, we have seen improvements that are so explosive that the numbers are embarrassing. At IBM one of the process improvements where ODC was at the foundation for the insights and guidance, yielded savings of over $100M. The same technology when applied at Nortel yielded similar results, as quoted by their executives at ISSRE Keynotes. In each instance, senior management understood the methods used, and was the primary sponsor. The work was executed by a small technical team that had access and influence in the organization. In both instances, the work spanned between 1-3 years. When process improvements are attempted by an organization without the guidance of experienced people, they often fail due to poor implementation and lack of skill.

This may not seem as a surprise to anyone. However, what stands out after 20 years of implementing process improvements across the globe, as to how few organizations support and implement them successfully. The outsourcing of software, which is often billed as a time and materials contract as opposed to fixed-price does not encourage process improvement. This places the responsibility on contract negotiation which often has the business and vendor

management that are unable to find and leverage the necessary software engineering knowledge to successfully build process improvement into the contract.

Khoshgoftaar: To a certain extent, yes, an improved focus on better process and better organization has resulted in less faulty software. The CMMI Institute and the SEI maintain reports of software development organizations that have measurably benefited from improving their development and organizational process, where the ratings of organizations have improved in the CMMI models. However, those examples typically come from high-assurance and/or mission-critical software projects which have much to lose due to poor software.

Miller: Again, I hate to be pessimistic, but no, I don't think that in general the process improvement movement made many inroads. Which is, in a way, quite sad. Because having a better process almost certainly improves the quality of the product that's generated. The real world jumps in here. Programmer/developers chase bug reports more than doing something systematic.

Porter: As long as you don't read the word "better" to necessarily mean more detailed, more formal, more rigid, etc., then I would say yes. Better organizations using better processes (as defined by them) will produce better software. For a concrete example, we worked for many years with a local company called Keymind. They decided to follow the CMMI approach, investing heavily in measuring their performance and improving their skills and tooling. Their investments ultimately paid off and they became a truly excellent organization and were recognized widely for their innovative products.

But process improvement is not just CMMI-type approaches. I know many organizations, for instance, that swear by their adoption of agile methods. After adopting and institutionalizing these practices, they now produce better, more cost-effective software than they did before the switch. I know other organizations that invested strategically in building specialized domain knowledge within their development team. Again, they now produce better software than they used to and are more effective in their specific customer markets.

9. **Once COTS products became the standard for software distribution, and source code was no longer available to customers, where did metrics fit into this new software distribution model?**

Abran: Indeed, SLOC-based metrics are almost irrelevant overall in a COTS context. Industry in general has not used new software metrics, even though they could have used Function Points to manage a number of COTS implementation and maintenance issues, including normalization of data collection to facilitate internal and external benchmarking for portfolio management and to objectively verify claimed productivity improvements with COTS.

Bieman: Customers can (and do) still measure the size of COTS products in terms of the number of bytes of storage (both RAM and disk). Dynamic measurements can still be used.

Basili: Good question. This changed the game for source code metrics and forced whole new processes to be developed to take COTS into the equation. It provides a good example of why metrics need to be defined for the context.

Chillarege: The COTS and metrics connection is at best remote. It is poorly understood by government and academia. It's a failure, that has gotten away in front of our eyes. In any mature engineering discipline, customers are protected against the failures of that engineering discipline, in one form or another. Software has managed to skirt this issues all along. No engineering method which has matured (and 50 years later, Software has certainly matured) can be allowed to deliver a service in society and claim it is not accountable on key parameters that affect society: Reliability, Injury, Productivity, Safety, etc. Yet, the software industry has managed to escape all of these. It is only today, that the threat of software security (or lack of it) has finally caught attention. The initial run up on security was accepted after several embarrassing disclosures by large firms. Today, with its impact on politics, it has finally garnered more attention.Yet, the focus is mainly on protection and damage control and not on the fundamentals of the technical area or the technology.

There is enough blame to go around all the stake holders. But, the largest one should be showered on the Academic community. For years, the technical communities that wielded the influence, such as programming languages, were critical of software engineering disciplines such as metrics and reliability that focused on behavioral aspects of programming. The consequence is that funding and generations of students and researchers were guided with priorities that ignored these very industry relevant areas. Today, those very purist disciplines that wielded influence have been commoditized and we have a dearth of technical effort in needed software engineering areas. I founded and headed the Center for Software Engineering at IBM Research and thus understand these dynamics only too well. The Center was unique in its day, and while it lived in the midst of the largest Computer Science department in the world it worked directly across the 50,000 software engineers sprawled that built products across IBM. My predictions, then was that Software Engineering, at large, will be sorry for the disposition that was held by the software technical community. Although a minority opinion then, IBM had the wisdom to let my opinion be heard, albeit without the needed follow through of investment and action. Now, 20 years later we are witnessing the consequences, that we, the collective technical society, chose. It could have been different. And it will be different in the future.

Khoshgoftaar: Since source code is not available for COTS software, the evaluation metrics tend to fall under categories of cost, return-on-investment, reliability, availability, and general black-box testing metrics. Often the development organization may rely on quality certifying entities that conduct independent testing on COTS products and maintain data on the products quality and related features for acquisition teams to review. A development organization that relies heavily on COTS, for example, may maintain an internal quality check list against which all COTS products are measured.

Porter: Metrics are not restricted to source code. In fact, organizations can and do define metrics over non-source code development artifacts, including requirements, user-visible display

screens, system resource files and more. The example of COTS is a good one. In some work we're doing for a large government organization, we've been very interested in metrics to help understand how much COTS customization this organization will need to perform.

We looked at many measures and identified that you need to look at the COTS development processes/activities, not just development of the glue-ware and integration of COTS components. For example, you also need to consider what it takes to learn the capabilities of the COTS products, to configure COTS components to satisfy requirements, to resolve issues with other interfacing development teams, and to enhance individual COTS products. Each of these activities requires significant effort, can cause great difficulty for a project, and are usually not fully planned and allocated the effort needed to develop a project.

10. **If you were to recommend 3 references to students or practitioners on the fundamentals of software metrics, what would they be?**

- Abran, *Software Metrics and Software Metrology*, John Wiley & Sons, 2010. John Wiley & Sons, 2010. (Abran, Bieman, Khoshgoftaar)

- ISO 15939: *Software Measurement Process (*Abran*)*

- ISBSG, Data collection questionnaire of the International Software Benchmarking Standards Group (www.isbsg.org) and related. (Abran)

- D.W. Hubbard, *How to Measure Anything: Finding the Value of Intangibles in Business*, John Wiley & Sons, 2010 (Bieman)

- N. Fenton and M. Neil, *Risk Assessment and Decision Analysis with Bayesian Networks*, CRC Press, 2012. (Bieman)

- N. Fenton, J. Bieman. *Software metrics: a rigorous and practical approach*. CRC Press, 2014. (Khoshgoftaar)

- J. Musa, *Software Reliability*, 1988 (Miller)

- M. H. Halstead, "Elements of Software Science," May 1977 (Miller)

- H. Hecht, "A Survey of Software Tools Usage," NBS 500-82 (NIST), 1981 (Miller)

- W. Humphrey*, "Introduction to the Personal Software Process."* Addison-Wesley, 1996. (Porter)

- R. B. Grady and D. L. Caswell, *Software Metrics: Establishing a Company-Wide Program*. Prentice-Hall, 1987. (Porter)

- T. Ball, Jung-min Kim, A.A. Porter and H.P. Siy, "If Your Version Control System Could Talk...", *ICSE '97 Workshop on Modeling and Empirical Studies of Software Engineering*, May 1997. (Porter)

- van Solingen & Berghout, The Goal/Question/Metric Method, McGraw Hill, 1999. (Basili)

- V. Basili, M. Zelkowitz, F. McGarry, J. Page, S. Waligora, and R. Pajerski, Special Report: SEL Software Process-Improvement Program *IEEE Software*, vol. 12(6): 83-87, November 1995 (Basili)

11. **Are current metrics cost effective? What aspects of software development are not being adequately addressed by metrics today, but could be? What are some fruitful areas for metrics research?**

Abran: The key issue with software metrics is not cost, but whether or not they support decision making. There are many software metrics for the whole lifecycle, and many software tools for automated measurement, but metrics with terrible designs should be quickly dropped as measurement methods when the next generation of better-designed metrics becomes available.

'Metrics' without analysis models or meaningful, context-specific data thresholds are useless. Individual organizations and industries must invest in building analysis models relevant to their contexts, and must collect historical data to bootstrap their own models and threshold values for decision making. The professional practices of metrics tool vendors, as well, need to improve considerably. At present, whatever metrics they propose in their tools are subject to their own interpretation, without any traceability to well-documented benchmarks or international standards.

In my opinion, a considerable amount of research on software metrics is wrong-headed. I have seen too many research papers where researchers collect a large number of metrics on the sole basis that they are easily automated. Then, using whatever open-source data they can put their hands on, without verifying the underlying quality, they try to figure out which ones might lead to more accurate outcomes for whatever purpose. This all-too-common approach is more closely related to random searches than a sound and proven research methodology.

Bieman: Most of the metrics used are relatively cheap to apply. However, misuse of metrics can add costs due to misdirecting developers. I'd like to see more research in two areas:

Evaluations of the measurable benefits and costs of applying common design advice and process advice in terms of time to market, delivered faults, and maintainability. Evaluating

maintainability may be the most difficult kind of study, as maintainability depends on external requests for repairs and new features.

The use of Bayesian networks to build causal models for decision making under the inherent uncertainty involved in software development. Such models have the potential to evaluate alternative software process arrangements and observe likely outcomes in terms of delivered functionality, time to deliver, and product quality.

Chillarege: Orthogonal Defect Classification is a concept I invented more than 25 years ago. It's based on a simple research finding that has deep consequences to all of software engineering metrics. It confounded me that the basic premise of software failures (and faults), were that they were treated as homogenous. I also did not understand just what they were counting. So I went up to the IBM Poughkeepsie lab, that was just up the road, and started studying the defect stream. Most of what I saw would not quite map itself into the models, and that got me thinking. I started fooling around with the data and discovered that sub-populations would behave very differently.

ODC extracts the semantics contained in defects into four principal groups, and within each, it bins them into independent categories. This multi-dimensional categorical data behaves like eigenvalues in the software development process space, thus creating a new measurement system. A dozen different process measurements and evaluations can be performed with ODC data. It has changed how one performs root-cause-analysis, reducing the effort required by two orders of magnitude.

Khoshgoftaar: In the larger scheme of things, current software metrics are generally cost effective. However, their extent of usage and role is dictated by project and organizational goals. An area of software development that could use further insight via measurements is human impact of software quality. Currently, this is generally measured via defects metrics and process metrics. An interdisciplinary focus between software engineering and psychology may yield useful insights. Another area that is beginning to be looked at is influence of Big Data analysis on existing software development practices.

Porter: Cost-effectiveness is really context dependent. For many projects, for example, measuring line coverage during nightly automated testing is a no-brainer. However, I was recently talking with someone about an ultra large scale system they worked on for which capturing even basic line coverage information was actually infeasible. In addition, metrics will be more cost effective if the definition of the metrics and the process to gather these metrics are carefully designed so as to be cost effective. Unfortunately, many companies put together an ad hoc measurement plan and most do not take advantage of data automatically captured by development/test tools nor plan for automated preprocessing/compilation of related data for easier analyses.

As for fruitful areas not fully studied today, I think there's a lot of room for defining and validating metrics over development models. Model-driven development approaches are increasingly finding their way into standard practice. As this trend continues, there'll be a need for metrics defined over these models.

Summary

We hope this retrospective was thought provoking. The diversities and similarities in opinions from the panelists made it more interesting. For example, we had near consensus that it's not possible to pick a useful single metric. Our panelists argued for matching metrics to goals and the need for strict metric definitions. And we had rough consensus that SLOC is weakly correlated with several metrics, but not sufficient as a metric by itself. Our panelists were generally supportive of CMM, and they surprised us when none of them discussed the use of structural coverage metrics as a check on the quality of requirements-derived tests.

We thank them for sharing their expertise and for their candor. So what do you think: were software metrics relevant back then, and if so, are they still relevant? And if they are, what's the best way to incorporate software measurement into modern day software development, and into software product and services delivery? After all, that is their purpose.

Disclaimer