CSU33012 – SOFTWARE ENGINEERING – REPORT

MEASURING SOFTWARE ENGINEERING

Liam Ó Lionáird [19335530]

January 3, 2022

In this modern world of data, humans can be measured in a dizzying myriad of ways. The long-elusive economic measure of 'worker productivity' is today an over-solved problem, a maze of pick-and-choose digital answers for any occasion. Few areas of modern labour are trickier to measure, however, than the very architects of those measurements—software engineering.

For a discipline with such a lofty scientific title, the 'engineering' of software development is harder to pick apart than that of its more physical, objective peers. The work of a software engineer is ephemeral, thought-driven, and steered by complex team dynamics; they are "knowledge workers", at odds with the "manual work" of other engineers [1]. The lineage of a single line of code is often too tangled to analyse for productivity; and its own 'effectiveness' in the whole of the software can vary so wildly as to render the notion useless.

Yet the digital nature of software engineering also makes it a tantalising fit for our new world of data-driven solutions. The search for buried truths of efficiency in an open ocean of code repositories is too alluring to pass over, and entire fields of research have embarked on the winding trail. What they have uncovered is well worth discussing here—as are the technologies that already illustrate these ideas, and finally the ethical questions that confront them all in turn.

* * *

The question of *what* to measure is the most open-ended, and at first glance the most trivial to answer. The atom of a computer program is a single statement or line of code, forming building blocks of functions and higher-level components. Simply measuring how much each software engineer writes over time should give a good estimate of 'productivity', even if a surface-level one—and this can be easily counted in 'commits' within a team's version-control system.

This measure can quickly fragment into complication, though. How are we sure which lines of code are more 'work-intensive' than others, or which components are more important for the project? A clever algorithm can take days

of thought, and reams of napkin-math, to solidify into what may amount to a single line of code—while elsewhere, big blocks of boilerplate are pasted into a file to solve a far more trivial problem. Code is not the only battleground of development, either; much of a team's work also lies in tracking issues, writing documentation, or simply discussing ideas. Several dangers of bias and inaccuracy can creep in, without deeper analysis of what exactly each developer is 'contributing' to the software.

Such a question has been tackled for decades by analysts in a similar field: measuring the *functional size* of a piece of software. In software development, *predicting* the required work in delivering a piece of software can be just as important as tracking the work in progress. An entire family of ISO standards exist for 'sizing' software in this manner, all based on the *function point* standard. Defined in 1979 by Allan Albrecht and John Gaffney of IBM, this system assigns 'weights' to each component of a software, based on component type and several other algorithmic factors [2]. Members of a team can also receive a score of productivity, resulting in a "function points per person month" metric for estimating the full development effort.

This parametric approach to measuring productivity was expanded upon by Barry W. Boehm in his *COCOMO* estimation model, published in 1981 [2]. It lists a number of attributes, or *cost drivers*, that affect software development—ranging from hardware constraints to product complexity to personnel skills—which are tallied with an elaborate table of multipliers to produce a final "effort" score. These early models still weigh developers' work in terms of lines of code, but the addition of other external development factors help to make their solutions more 'physical' and objective.

Still, the software development landscape has changed in the decades since. Most notably, the *process* of developing software has blossomed into several contentious philosophies, many of which lend themselves less easily to such planning. The straightforward 'waterfall' model was widespread in the early days of software engineering, as a holdover from other engineering industries—tasks were separated into linear 'phases', allowing checks to be made at each stage, during which the team's progress could be properly measured. However, this proved too inflexible for the needs of many clients, and often too slow to keep up with the ever-updating tech standards around it. In modern times, it has mostly been replaced with the 'agile' development process—where work is more iterative and less set-in-stone, but often notably faster and fresher as a result. Yet with this approach, there is far less room to conduct reviews of a team's productivity and progress, because 'progress' itself is so loosely defined within the project [3].

Recent iterations of old models (such as *COCOMO*'s own second version in 2000) have kept them up to date with emerging development methods and obstacles. Still newer approaches have arisen since, taking advantage of the increasing power of computers themselves. Rather than a manual spreadsheet-tally at the start or end of a project, *telemetry* software can analyse the progress of a development project in real-time. One example is the open-source *Hackystat* framework, first published by Johnson et al. in 2005 [4]. The framework

makes use of 'sensor' programs attached to development tools, which send back data on the project's progress. This style of analysis is claimed to "facilitate local, in-process decision making", which makes it potentially more useful for agile software development in particular.

* * *

To understand the effectiveness of a new software engineering metric, researchers must test them on datasets of software projects. Thankfully, modern analysts have access to far more open and large datasets than were available 40 years ago. Today, millions of code repositories are stored online through services such as GitHub, easily accessible through their APIs. The practice of analysing open repositories has become so common that an annual *Mining Software Repositories* conference has been hosted since 2004 [5]. (Dozens of papers are presented each year, reporting brand-new analysis methods and their results.) This movement has also opened up the doors to new areas in software development where data can be gathered—for instance, the issue-tracker sections of GitHub repositories—and new computing platforms to handle and process this data.

One major case study is Pluralsight's *Flow* service [6], which offers an advanced all-in-one software engineering analysis platform. Users can easily visualise the history and status of their Git projects, the contributions of its members, and a broad insight into the project's performance and any potential shortcomings. Flow goes deeper still in analysing the team members themselves, and the dynamics between them—how well they contribute, how often they communicate together, and the relationship between senior engineers and the developers they mentor. It bills itself as a comprehensive solution to the task of 'measuring software engineering', built on modern and sophisticated analysis tools.

* * *

Analysing a software engineering project necessarily boils down to analysing the *people* behind it—and this raises all of the usual ethical questions and privacy issues involved. First, in testing and mining: hosts of online code repositories may not consent to be studied in datasets, or may wish to be anonymised—and GDPR compliance must be enforced throughout by researchers in the field.

A more unnerving problem rears its head, however, when putting this analysis into real-world practice. What of the software engineers themselves? How much analytical oversight and productivity reports can be imposed upon a development team, before what starts as helpful aid transforms into needless micromanagement... or a force more totalitarian still?

This is the nexus at which modern software development clashes with modern software business. It is striking how the evolving techniques in this field over the decades have tracked so closely with our growing hunger for data, and with the changing of power throughout computer science. The architects of the first software engineering productivity tests were themselves software

engineers, often working in government—the 'function point' system was developed by IBM technical staff [2], and much of our understanding of how to *manage* software development originated with the complicated projects of the US Military in the 1970s [7]. With the consolidation of power in 21st-century Silicon Valley, a different mindset has entered the picture; this one rooted in a more corporate mentality.

References

[1] Drucker, P. F. (1999). Knowledge-worker productivity: The biggest challenge. *California Management Review, 41*(2), 79–94.

[2] Stutzke, R. D. (1996). Software estimating technology: A survey. *CrossTalk, 9*(5), 204–215.

[3] Cohn, M. (2018). *Incorporating governance or oversight into an agile project*. Mountain Goat Software.
`https://www.mountaingoatsoftware.com/blog/incorporating-governance-or-oversight-into-an-agile-project`

[4] Johnson, P. M., Kou, H., Paulding, M., Zhang, Q., Kagawa, A., Yamashita, T. (2005). Improving software development management through software project telemetry. *IEEE Software, 22*(4), 76–85.

[5] International Conference on Mining Software Repositories,
`http://www.msrconf.org/`.

[6] `https://www.pluralsight.com/product/flow`

[7] Ware, W. H., Patrick, R. L. (1983). *Perspectives on oversight management of software development projects.* Rand Corporation.
`https://www.rand.org/pubs/notes/N2027.html`