Long-Term Productivity Based on Science, not Preference

Spencer Smith
McMaster University
Hamilton, Canada
smiths@mcmaster.ca

Jacques Carette
McMaster University
Hamilton, Canada
carette@mcmaster.ca

Our goal is to identify inhibitors and catalysts for productive long-term scientific software development. The inhibitors and catalysts could take the form of processes, tools, techniques, environmental factors (like working conditions) and software artifacts (such as user manuals, unit tests, design documents and code). The effort (time) invested in catalysts will pay off in the long-term, while inhibitors will take up resources, and can lower product quality.

Developer surveys on inhibitors and catalysts will yield responses as varied as the education and experiential backgrounds of the respondents. Althought well-meaning, responses will predictably be biased. For instance, developers may be guilty of the *sunk cost fallacy*, promoting a technology they have invested considerable hours in learning, even if the current costs outweigh the benefits. Likewise developers may recommend against spending time on proper requirements, not as an indication that requirements are not valuable, only that current practice doesn't promote requirements [2]. Another perceived inhibitor is time spent in meetings. For instance, the lack of visible short-term benefits renders department retreats unpopular, even though relationship building and strategic decision making may provide significant future rewards. Evaluating the usefulness of meetings is difficult. Rather than relying on preference and perception, As these examples illustrate, *we need to measure the long-term impoact of development choices to make wise ones.*

1 BUILDING BLOCKS

A scientific approach requires a solid foundation. The building blocks for scientific discourse are: an unambiguous language for communicating concepts, formulating hypotheses, planning data collection, and analyzing models and theories. To start with, we need to classify the software under discussion. Likely dimensions include: general purpose scientific tools versus special purpose physical models, scientific domain, open source versus commercial software, project maturity, project size, and level of safety criticality.

We also need to be precise about our software quality goals. Qualities such as reliability, sustainability, reproducibility and productivity need precise definitions. Attempts have been made since the 1970s [6], but the resulting definitions aren't usually specific to scientific software (as shown by the confusion between precision and accuracy is the ISO/IEC definitions [4]). Moreover, the definitions often focus on measurability, where the first priority should be conceptual clarity, analogous to the unmeasurable, but conceptually clear, definition of forward error, which requires knowing the (usually unknown) true answer.

For each relevant quality we recommend collecting as many distinct definitions as possible. Once collected, they can be assessed against the following criteria (based on IEEE [3]): completeness, consistency, modifiability, traceability, unambiguity and abstractness. The understanding gained from this systematic survey and analysis can be used to either choose solid definitions, or proposed new ones. In all cases, the definitions should enable *reasoning about quality*.

2 PRODUCTIVITY

Our definition of long-term productivity [8] provides an example of our vision, and meets our criteria. We define productivity as:

$$P = O/I$$

$$I = \int_0^T H(t)\,dt$$

$$O = \int_0^T \sum_{c \in C} F(S_c(t), K_c(t))\,dt$$

where P is productivity, I is the inputs, O is the outputs, 0 is the time the project started, T is the time *in the future* where we want to take stock, H is the total number of hours available by all personnel, C represents different classes of users (external as well as internal), S is *user satisfaction* and K is *effective knowledge*, and F is a weighing function that indicates "value". Thus productivity is measured in "value per year." and is a mixture of external and internal value produced. *Value* should not be equated with money; measuring the productivity of free software development is just as important as for commercial software.

While the most straightforward use of such a formula is to measure productivity of a team, it can also be used in "what if" scenarios to assist in planning interventions, i.e. changes intended to improve productivity.

Measuring over too short a time-frame will assuredly give warped results. This leads some to argue that productivity shouldn't even be measured [5].

3 MEASURING

Proper science requires measurement. We can only determine whether a given intervention is a catalyst or inhibitor by measuring its impact. Let us examine in more details the consequences of our proposed definition.

First, the time integrals emphasize that productivity is something that happens *over time*. The most interesting kind of productivity is that of an organization over the span of years. Measuring over too short a time frame is one of the main sources of *technical debt*[**?**] as it devalues planning, team work, being strategic, etc.

Secondly, as Drucker [1] reminds us, quality is at least as important as quantity. Here we use a proxy for quality, namely *user satisfaction*. It is important to note that unreleased products and unreleased features induce **no** user satisfaction. A broken product might be even worse, and produce negative satisfaction.

The input H is the number of hours worked by the team, including managers and support staff, as appropriate. To optimize productivity, we want to make I, and thus H, *small*. This is the raw input being applied, whether effective or not.

We use user satisfactor (S) as a proxy for effective quality. How to measure this is left for future study. It can be approximated by measures such as numbers of users, number of citations, number of forks of a repository, number of "stars", surveys of existing users, number of mentions in the issue tracker, and usability experiments.

Probably the trickiest part is *effective knowledge* (K). The idea is that while source code embodies operational knowledge that has the potential to directly lead to user satisfaction, a project usually also generates a lot of *tacit knowledge* about design, including the rationale for various choices. This is the kind of knowledge that is lost when employees leave, and is the most costly to build and replace. In other words, human-reusable knowledge such as documentation factors in here. The best measure for knowledge is an area for future exploration.

4 ARTIFACTS

The software development process outputs different forms of knowledge. This knowledge is typically distributed across multiple software artifacts. Potential artifacts include requirements, specifications, user manual, unit tests, system tests, usability tests, build scripts, API (Application Programming Interface) documentation, READMEs, license documents, process documents, and code.

Although documentation is often neglected for scientific software, we anticipate that explicit evidence, and a long-term viewpoint, will show which documents are catalysts for different classes of users. As an example, for long-lived project with significant contributor turnover, the users tasked with training new developers will consider onboarding documents as catalysts.

As we gain understanding on measures of productivity, those measures can be used to determine the state of the practice for different scientific software domains. We can estimate K and S for existing artifacts. (Unfortunately, it is unlikely that records will be available for the corresponding H values.) We can then potentially compare real projects to the K and S values for the artifacts recommended by software engineering textbooks. The combined information can then be analyzed to determine how K is distributed. We know that knowledge will be duplicated. The data will allow us to understand usability reasons (related to S) for the different views of the same knowledge. Putting this all together, we hope researchers will find the most valuable K and when and how it should appear in future artifacts for different classes of users.

Besides looking at K and S, another way to judge the utility of documentation is to look at the documentation necessary to make an assurance case. An assurance case [7] presents an organized and explicit argument for correctness (or whatever other software quality is deemed important) through a series of sub-arguments and evidence. Documentation will be necessary, but through assurance cases the developers will only create the documentation that is relevant and necessary.

5 PRODUCTION METHODS

The previous section focused on increasing O, but another way to improve productivity is to reduce I. The production methods, or process, used to build software has a significant impact on I. One of the likely reasons that developers focus on code, and code related artifacts (like testing files and build scripts), is that documentation is difficult to keep in sync as the project evolves over time. Out of sync documentation is arguably worse than no documentation; therefore, a case could be made that developers are "improving" productivity by not producing documentation. Of course, a better approach to improve productivity is to capture valuable knowledge in relevant documentation that is continually in sync with the code.

In the future, we recommend developing an approach that synchronizes code and documentation. A promising approach is to generate all artifacts from a knowledge base [9]. Using the understanding of artifacts gained from the work previously proposed, a generator can be taught to create artifacts with satisfactory knowledge. Since the approach is generative, repetition of knowledge is fine. This means that documents can be tailored to different classes of users, thus further improving productivity. A generative approach can potentially produce high S and high K, while reducing H.

6 CONCLUDING REMARKS

Our position is that decisions on processes, tools, techniques and software artifacts should be driven by science, not by personal preference. Decisions should not be based on anecdotal evidence, gut instinct or the path of least resistance. Moreover, decisions should vary depending on the users and the context. In many cases this will mean that a longer term view should be adopted. We need to use a scientific approach based on unambiguous definitions, empirical evidence, hypothesis testing and rigorous processes.

By developing an understanding of how input hours, satisfaction and knowledge are related to productivity, in the future we will be able to determine what interventions have the greatest return on investment. We will be able to recommend software production methods and artifacts that justify their value because the output benefits are high compared to the required input resources.

REFERENCES

[1] P.F. Drucker. 1999. Knowledge-Worker Productivity: The Biggest Challenge. *California Management Review* 41, 2 (1999), 79–94.

[2] Dustin Heaton and Jeffrey C. Carver. 2015. Claims About the Use of Software Engineering Practices in Science. *Inf. Softw. Technol.* 67, C (Nov. 2015), 207–219. https://doi.org/10.1016/j.infsof.2015.07.011

[3] IEEE. 1998. Recommended Practice for Software Requirements Specifications. *IEEE Std 830-1998* (Oct. 1998), 1–40. https://doi.org/10.1109/IEEESTD.1998.88286

[4] ISO/IEC. 2001. *ISO/IEC 9126. Software engineering – Product quality.* ISO/IEC.

[5] Amy J. Ko. 2019. *Why We Should Not Measure Productivity.* Apress, Berkeley, CA, 21–26. https://doi.org/10.1007/978-1-4842-4221-6_3

[6] J. McCall, P. Richards, and G. Walters. 1977. *Factors in Software Quality.* NTIS AD-A049-014, 015, 055.

[7] David J. Rinehart, John C. Knight, and Jonathan Rowanhill. 2015. *Current Practices in Constructing and Evaluating Assurance Cases with Applications to Aviation.* Technical Report CR-2014-218678. National Aeronautics and Space Administration (NASA), Langley Research Centre, Hampton, Virginia.

[8] Spencer Smith and Jacques Carette. 2020. Long-term Productivity for Long-term Impact, arXiv report. https://arxiv.org/abs/2009.14015. arXiv:2009.14015 [cs.SE]

[9] Daniel Szymczak, W. Spencer Smith, and Jacques Carette. 2016. Position Paper: A Knowledge-Based Approach to Scientific Software Development. In *Proceedings of SE4Science'16 in conjunction with the International Conference on Software Engineering (ICSE).* In conjunction with ICSE 2016, Austin, Texas, United States. 4 pp.