# 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. Although 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 impact of development choices to make wise ones.*

## 1 BUILDING BLOCKS

A scientific approach requires a solid foundation. The building blocks for scientific discourse are: communicating concepts via an unambiguous language, 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 [7], 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 propose new ones. In all cases, the definitions should enable *reasoning about quality*.

## 2 PRODUCTIVITY

Our definition of long-term productivity [9] 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* [6] 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 satisfaction ($S$) 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 PRODUCED

Software development typically produces many artifacts, such as requirements, specifications, user manuals, unit tests, system tests, usability tests, build scripts, API (Application Programming Interface) documentation, READMEs, license documents, process documents, and code. We regard all of these as containing *knowledge*, albeit encoded in different forms. Furthermore, it is crucial to recognize that the knowledge of a single product is *distributed* amongst those artifacts. In particular, the various artifacts contain many copies of the same core knowledge — by design.

To understand the importance of certain artifacts, it makes sense to look at the productivity impact of their presence/absence. For example, long-lived projects will inevitably encounter contributor turnover. How long should it take for new contributors to be productive? How much training by peer mentors will it take? Could some documentation be written that would shorten this learning period and, just as importantly, reduce the time it takes from experienced people? Of course, documentation that is out-of-date could be even worse: a false sense of knowledge that results in even more wasted work that needs repairing.

As we gain understanding on measures of value, we can use them to evaluate the state of practice in different research software domains. We can estimate the knowledge $K$ embedded in, and the user value $S$ derived from, existing artifacts. In particular, we can compare these to the artifacts produced by recommended processes from standard software engineering textbooks. For example, we can test the hypothesis that knowledge duplication between code and requirements, coupled with the fact that requirements get desynchronized from the code and the tenuous link to user value, is the likely reason for low adoption of requirements in scientific software development [2].

Nevertheless, documentation remains useful, especially for the very long term. Another means to judge the utility of documentation is to look at assurance cases. An assurance case [8] presents an organized and explicit argument for correctness (or whatever other software quality is deemed important) through a series of sub-arguments and evidence. Assurance cases gives at least one measure of which documentation is relevant and necessary.

## 5 PRODUCTION METHODS

One way to improve productivity is to waste less on non-productive or counter-productive activities. That code is the most visible artifact that contributes user-value, along with with testing (because quality is an extremely important factor in user-value) explains the inordinate focus on just those artifacts. Furthermore, the deemphasis on documentation, even to the extreme of some methodologies having none, can feel like productivity improvements in the short term! A better approach would be to capture knowledge in ways that keeps it continuously synchronized between the various artifacts where it appears.

One promising approach is to generate all artifacts from a single knowledge base [10]. This relies on a solid understanding of the contents of all of the artifacts present in the software engineering process. Our proof-of-concept shows that this is possible. As the artifacts are now generated, knowledge duplication is not a problem. Even better, the knowledge is synchronized-by-construction. Furthermore, it becomes easy to tailor artifacts, documentation as well as code, to different classes of "users".

## 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 most cases of interest, this means 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 where input hours are spent, what most contributes to user satisfaction, and how to leverage knowledge produced, we can determine what has the greatest return on investment. We will be able to recommend software production processes that justify their value because the long-term 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] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. 2012. Technical debt: From metaphor to theory and practice. *IEEE Software* 29, 6 (2012), 18–21.

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

[8] 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.

[9] 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]

[10] 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).* Austin, Texas, United States. 4 pp.