Long-Term Productivity Based on Science, not Preference

Spencer Smith and Jacques Carette
Computing and Software, McMaster University, Canada
smiths@mcmaster.ca and 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, 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, but may actually lower product quality.

If developers are surveyed on inhibitors and catalysts, their answers will be as varied as the education and experiential backgrounds of the respondents. Their responses will be well-meaning, but they will undoubtedly come with problems and biases. For instance, developers may be guilty of the *sunk cost fallacy*, promoting a technology that they have invested considerable hours in learning, even if the current costs outweigh the benefits. As another example, developers may recommend against spending time on proper requirements, but this lack of support doesn't imply requirements are an inhibitor, only that current practice doesn't promote requirements [2]. Another perceived inhibitor is time spent in meetings. For instance, departmental retreats can be unpopular because of a lack of short-term benefits, but relationship building and strategic decision making may provide significant future rewards. The difficult trick is to know which meetings are useful, and which are not. As these examples illustrate, *we need to take developer and personal preference out of the development process and instead pick the artifacts/processes/tools that have a long-term impact.*

1 BUILDING BLOCKS

A scientific approach requires a solid foundation. The building blocks for scientific discourse is 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 for classification include the following: 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.

Another important language component is defining what we hope to achieve in terms of scientific software quality. Qualities that need to be unambiguously defined include reliability, sustainability, reproducibility and productivity. Software engineers have frequently attempted to define quality since the 1970s [5], but the 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 instances of the definition as possible. Once all the definitions are 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 then be used to propose a new set of quality definitions that allow for reasoning about quality.

2 PRODUCTIVITY

Our definition of long-term productivity [7] provides an example of how we envision quality definitions in the future. The definition meets the criteria listed in the previous section. Moreover, it provides a starting point for reasoning about improving productivity, as explored in subsequent sections. We define productivity as:

$$P = O/I$$

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

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

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 satisfaction and K is *practical knowledge*. Thus productivity is measured in "satisfying reusable knowledge per hour." The equation can be applied to measure the productivity of a single developer or a group of developers. The equation can also be used to assess a given intervention, where an intervention is a process or an output intended to improve productivity. An intervention process can be assessed to determine if it reduces I. For outputs we can back calculate the inputs required to achieve the touted benefits.

3 MEASURING

Proper science requires measurement. We can only determine whether a given intervention is a catalyst or inhibitor by measuring its impact. We can illustrate this via the above productivity equation. Although it cannot be directly measured, examining the parts provides insight.

The equation shows an integral over time to emphasize the time-frame. Some developers may show short-term productivity, say by producing a poor design that repeats the same, or very similar, code in multiple places. However, long-term productivity should favour the developer that refactors the design by replacing the repeated code with a function. A productivity measure for knowledge workers cannot just focus on quantity, quality is at least as important [1]. The proposed definition captures this by combining knowledge with satisfaction.

The input H is the number of hours worked. Depending on the context, this will be the total hours worked by a developer, or the

total hours required (possibly by multiple developers) to realize a given intervention (process or output).

The output depends on user satisfaction (S), where user satisfaction acts as a proxy for effective quality. How best to assess satisfaction should be studied in the future. As a starting point, satisfaction can potentially be approximated with such measures as numbers of users, number of citations, number of forks of a repository, the number of "stars", surveys of existing users, number of mentions in the issue tracker, and experiments to measure usability.

Productivity means increasing the effective knowledge (K) delivered to users over time, at a lesser cost. As the knowledge produced will be used by different kinds of users (such as internal developers and external users), it is important to weigh the satisfaction of each class separately. This explicit emphasis on all knowledge produced, rather than just the operationalizable knowledge (aka code) implies that human-reusable knowledge, i.e. documentation, is crucial. This reinforces the importance of the long-lived context. 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 measure 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 this 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 use the documentation necessary to make an assurance case. An assurance case [6] 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, an approach should be developed that synchronizes code and documentation. A promising approach is to generate all artifacts from a knowledge base [8]. 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] J. McCall, P. Richards, and G. Walters. 1977. *Factors in Software Quality.* NTIS AD-A049-014, 015, 055.

[6] 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.

[7] 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]

[8] 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.