The Measurement of Software Maintenance and Sustainment

Positive Influences and Unintended Consequences

Rob Ashmore, U.K. Defence Science and Technology Laboratory
Mike Standish, U.K. Defence Science and Technology Laboratory

Abstract. Software metrics can provide valuable information to decision makers and can assist with the management of the software supply chain. However, a poorly chosen set of metrics may have negative unintended consequences, resulting in software that is more expensive to maintain. Among other things, a crucial component in the intelligent and effective use of software metrics is a sound, system-level understanding of the underlying software sustainment process. This article illustrates one mechanism that can be used to develop this understanding and highlights the benefits that it can deliver.

Introduction

In this paper we are interested in what motivates the choice of a particular set of software metrics, especially when these metrics are to be used as part of a Performance Based Logistics (PBL) [1] contract. These types of contract are being implemented for large and complex software systems such as the F-35 Lightning II [2]. In particular, we are concerned with how the chosen metrics will influence the behavior of the software maintenance and sustainment system. Of particular concern is protecting against negative unintended consequences that may occur if metrics are chosen poorly.

Initially, we consider how measurement can affect the behavior of human systems. We then develop some simple, and incomplete, models that illustrate how poorly chosen metrics could drive the software maintenance and sustainment system toward undesirable behaviors. We close by summarizing how to protect against this risk.

Measurement and Behavior

The relationship between measurement and system behavior has been the subject of a number of studies of complex human systems (e.g., implementation of Government policies [3], the English National Health Service [4], and U.S. Veterans Health Administration facilities [5]). In addition, the unintended consequences of publishing performance data for U.K. public sector organizations have been studied; this topic is discussed in the following paragraphs, which are heavily based on Smith [6].

In his abstract, Smith states, “the performance indicator philosophy is based on inadequate models of production and control.” To put that quote into the context of this article, Performance Indicators (PIs) can be considered analogous to software metrics, and the phrase “models of production and control” relates to a system-level model of the activity that is being undertaken, which in our case is software maintenance and sustainment.

Smith goes on to note, “the findings of the paper are therefore likely to be relevant to any situation in which performance data — whether directed at political, agency or managerial control — play a significant part in guiding the activities of the organization.” This situation is precisely the one we are interested in, where performance-based measures (i.e., software metrics) are used to influence the behavior of the software maintenance and sustainment organization so that defined outcomes are achieved for the warfighter.

Potential Unintended Consequences

Smith highlights a number of negative unintended consequences of using performance data to influence system behavior. These are all based on observations of U.K. public sector organizations and can be grouped into eight distinct types:

- **Tunnel vision**, when management focuses on quantified aspects of performance rather than overall quality.
- **Suboptimization**, where narrow, local objectives are prioritized over the wider objectives of the organization as a whole.
- **Myopia**, which involves the pursuit of short-term targets at the expense of legitimate long-term objectives or outcomes.
- **Measure fixation**, where managers focus on the metric, rather than the objective for which the metric was developed.
- **Misrepresentation**, where the reported metrics do not match the behavior on the ground.
- **Misinterpretation**, where those to whom the metrics are reported make incorrect or inappropriate decisions.
- **Gaming**, where behavior is deliberately altered so as to exploit loopholes in the measurement system.
- **Ossification**, where an overly rigid measurement system prevents innovation.

Mitigation Strategies

Smith advances a number of strategies that may be used to mitigate the risk of such unintended consequences. Two prime examples are the following:

- Involving staff at all levels when setting metrics; this readily protects against suboptimization.
- Retaining flexibility in the chosen metrics and not relying on them exclusively for control purposes; this readily protects against ossification.

While these strategies provide some level of mitigation, a more holistic approach involves gaining a system-level understanding of what is being measured. This is a key consideration, which is highlighted in the abstract of Smith’s paper. Although the development of a completely accurate model is impossible, relatively simple models can still provide a helpful level of understanding. More importantly, they allow the effects of measurement to be monitored so that, if necessary, the adopted measures can be altered.

One way of representing system-level models is by using Causal Loop Diagrams (CLDs). To illustrate these diagrams, we begin with a simple example relating to technical debt and its potential effect on software release schedules.

Technical Debt and Preventive Changes

The term "technical debt" was coined by Ward Cunningham in a talk at the 1992 Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) conference [7].It refers to code that is known to be 'not quite right' but a decision has been made to postpone making it right. Often, but not always, the 'not quite right' piece relates to the way new code integrates with an existing software architecture.

Incurring technical debt allows, for example, a new release to be produced sooner than otherwise would be the case. This usually comes at a longer-term cost, as indicated by the debt metaphor. In particular, as this level of debt grows, it becomes more difficult to make changes, slowing down future releases. Ultimately, an unchecked growth in technical debt is likely to shorten the lifespan of the software, hastening the need for its replacement.

Paying back the debt involves a software change, specifically one that aims to ease future maintenance. This is one of the four types of change discussed by Williams and Carver [8]:

- **Perfective** changes result from new or changed requirements.
- **Corrective** changes occur in response to defects.
- **Adaptive** changes occur when moving to a new environment, or to a new platform, or to accommodate new standards.
- **Preventative** changes ease future maintenance by restructuring or reengineering the system.

Figure 1 illustrates some of these concepts. Initially, we focus on the items shown in black text; i.e., we focus on the reinforcing loop, which is named "impending bankruptcy." This may be interpreted as follows:

- An increase in software schedule pressure leads to an increase in technical debt. (The "**S**" on the arrow means that an increase in the quantity at the arrow's tail leads to an increase in the item at its head; that is, they move in the Same direction.)
- The increase in technical debt leads to a reduction in the ease with which future changes can be met. (The "**O**" on this arrow means the quantities it joins move in Opposite directions.)

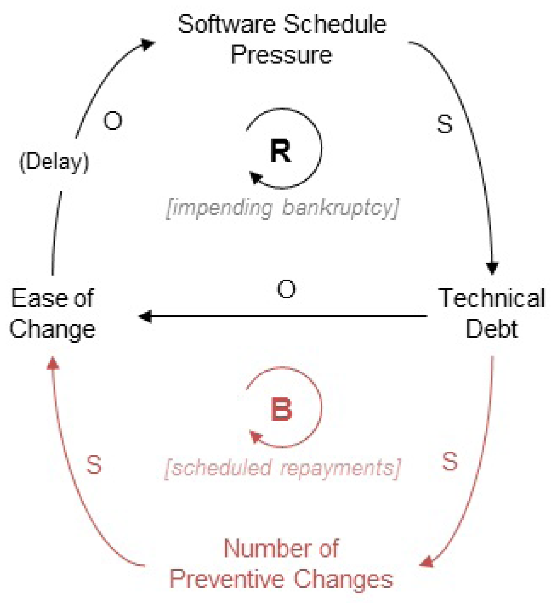

Figure 1. Initial causal loop diagram for technical debt

- The reduction in the ease with which future changes can be met leads, in turn, to schedule pressure for future releases (hence the delay).

Overall, working around the loop, an increase in schedule pressure leads to a further increase in schedule pressure. Hence, this is a reinforcing loop and is annotated with an "**R**" (in simple terms, a loop that has an even number of "**O**" arrows will be Reinforcing and a loop with an odd number of "**O**" arrows will be Balancing). The system-level effect is that increasing technical debt to achieve one release will, if left uncorrected, make it more and more difficult to complete future releases. Ultimately, this will reduce the software's lifespan or, extending the debt metaphor, lead to technical bankruptcy.

It is appropriate to note that the preceding description (and the associated CLD) is just one selection from a range of potential situations. For example, a small amount of technical debt residing in a stable part of the software (i.e., a part where few, if any, changes are made) might be maintained indefinitely with little adverse effect. More generally, any CLD is just one way of representing a system. Different representations, and hence different system-level behaviors, are often plausible. This is one reason why flexibility should be retained in the chosen metrics. It also highlights the importance of monitoring system behavior and comparing it with expectations.

With that caveat in mind, consider the red text in Figure 1; i.e., where we focus on the loop named "scheduled repayments." This may be interpreted in the following fashion:

- An increase in schedule pressure leads to an increase in technical debt; this prompts an increase in the number of preventive changes, which makes future changes easier, thus re-balancing schedule pressure across future releases.

In this case, an initial increase in schedule pressure works through the loop to result in a reduction in schedule pressure later. This is a balancing loop (as denoted by the "**B**").

Applying Smith's observations on unintended consequences to this simple example suggests that using software metrics based solely on the time taken to complete the current release risks the system being driven to behave as in the "impending bankruptcy" loop; that is, the longer-term future of the software will be jeopardized by tightly focusing on short-term issues. In contrast, including metrics that encourage the implementation of perfective changes is, according to this model, more likely to drive the system into a balanced behavior, which should yield through-life benefits.

Of course, both of these examples are simplistic and fail to capture the full complexities of system behavior. Nevertheless, they still provide useful insights on the potential unintended consequences of adopting a particularly narrow set of software metrics.

A Simple Model of Software Maintenance and Sustainment

Figure 2 provides a larger, but still incomplete, model of software maintenance and sustainment; a separate model of the same activity was discussed by Ferguson, et. al., in "Modeling Software Sustainment" [9]. Like the model discussed in the previous sub-section, the one presented here is just one representation of a

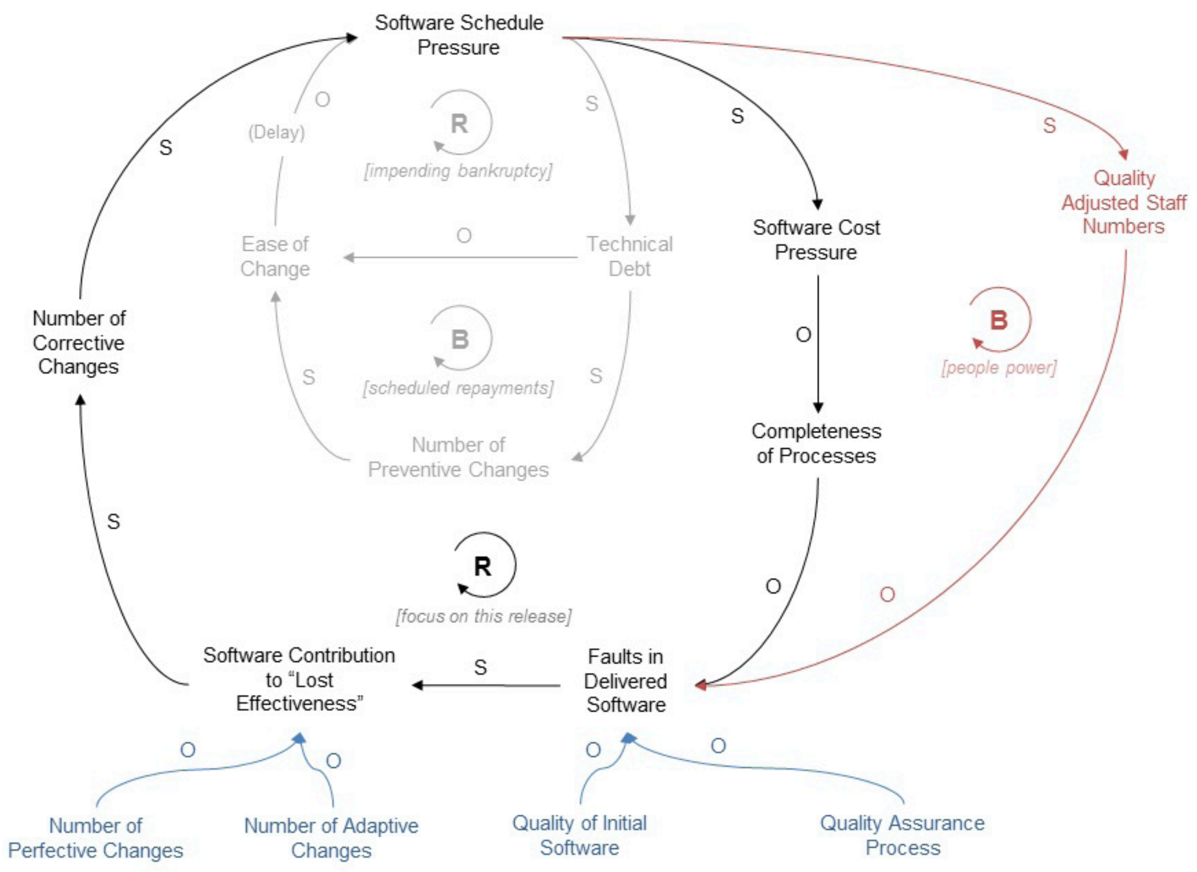
Software Schedule Pressure
(Delay)
O
S
R
[impending bankruptcy]
S
S
Quality Adjusted Staff Numbers
Ease of Change
O
Technical Debt
Software Cost Pressure
Number of Corrective Changes
B
[scheduled repayments]
S
S
O
B
[people power]
Number of Preventive Changes
Completeness of Processes
S
R
[focus on this release]
O
O
Software Contribution to "Lost Effectiveness"
S
Faults in Delivered Software
O
O
O
O
Number of Perfective Changes
Number of Adaptive Changes
Quality of Initial Software
Quality Assurance Process

Figure 2. A simple model of software maintenance and sustainment

potential behavior. Nevertheless, it captures some important features, which can be used to inform the choice of metrics.

The following points highlight some of the key nodes in the model:

- The central portion of the model (shown in gray) reproduces the relationship discussed previously (shown within Figure 1).
- On the bottom left of the diagram is a node that captures "Software Contribution to 'Lost Effectiveness.'" Note that this node also captures effectiveness that is "lost" due to an inability to meet new requirements.
- The lost effectiveness highlighted in the previous node is likely to affect the "Number of Corrective Changes" that are required.
- The "Software Cost Pressure" node is self-explanatory. It is included in the model as it addresses a metric that, by definition, has to be included in any PBL contract (i.e., cost).
- The "Completeness of Processes" node is included to represent the balance between maintaining a planned release date (which can involve reducing the amount of time available for testing) and, for want of a better phrase, the level of "thoroughness" involved in the process.
- The "Faults in Delivered Software" node implicitly captures both the number and the importance of any faults that are present (i.e., it covers issues like whether a workaround is available, the number of missions that would exercise the faulty software, and so on).
- The "Quality Adjusted Staff Numbers" node (shown in red on the right of the diagram) combines both the number of staff that are available and their quality. This node covers all software-related staff, including, for example, managers, developers, reviewers and testers.
- On the bottom of the diagram are four external inputs, which are shown in blue text:
  - The first two external inputs, "Number of Perfective Changes" and "Number of Adaptive Changes" capture the impact of new requirements, whether these arise, for example, from a desire to conduct new missions (perfective) or a need to accommodate new standards (adaptive).
  - The next external input relates to the "Quality of the Initial Software;" i.e., the quality of the software when the system achieves Initial Operating Capability (IOC). Unlike new requirements, which may arise multiple times during the aircraft's life, this external input is a one-off.
  - The final external input relates to the "Quality Assurance Process." This is included as the Quality Assurance (QA) organization is independent of the software development one and, as such, QA should be able to, for example, prevent inappropriate short cuts (i.e., ones that violate agreed procedures) being taken.

Reinforcing Loop – "Focus on This Release"

Initially, we consider the reinforcing loop named "focus on this release." This interpretation of system behavior can be described as follows:

- An increase in "lost effectiveness" due to shortfalls in software functionality increases both the demand for corrective changes and the pressure for a new release to be produced quickly. This is assumed to lead to an increase in cost pressure, a link that is somewhat debatable; sometimes doing things more quickly also means doing them more cheaply.

- The combination of increased schedule pressure and increased cost pressure reduces the completeness of the processes (e.g., the number of resources devoted to testing). In turn, this increases the number of faults in the software, which further increases "lost effectiveness."

As with all the causal loops discussed in this article, this description presents a very simple view of what is, in reality, a complex system; it also contains general statements for which specific counter-examples could be provided. For example, the "lost effectiveness" may not be felt immediately, as the new release may be expected to fix most (if not all) of the faults that were impacting effectiveness. Likewise, the relationship between schedule pressure and testing completeness is not entirely governed by cost pressure. Nevertheless, this discussion again highlights the potential risk of focusing too tightly on cost and schedule metrics for each upcoming release in isolation.

Balancing Loop – "People Power"

Finally, we consider the balancing loop named "people power." Much of the description of this loop is similar to the previous discussion. The new features may be summarized as follows:

- An increase in schedule pressure is used to justify an increase in staff numbers and/or quality. This acts to reduce the number of faults in the delivered software, thus providing a degree of balance to the system.

Yet again, this description is somewhat simplified and idealistic. It could be argued, for example, that schedule pressure will not lead to an increase in staff numbers and/or staff quality (e.g., because suitable staff cannot be recruited, or because there is a perception that there is no spare time for training). Likewise, there is some evidence that simply adding more staff to a late-running software project may make it later (this is sometimes referred to as Brooks' Law and is described in [10]). Conversely, bringing experienced staff with detailed knowledge of the project's history back onto the team could be beneficial.

Despite these limitations, this loop does illustrate the benefit that can be obtained by combining an appropriate set of metrics with an understanding of the desired system-level behavior. In particular, measuring the planned staff attributes at the beginning of a release cycle and comparing this with the likely schedule pressure — which may be informed by data from previous releases — should help drive the overall system into a more balanced state than otherwise would be the case. Achieving and maintaining such a state is an important aspect of the through-life cost-effectiveness of defense software; it should also help make software release schedules more predictable.

Conclusions

It is well understood that measurement influences behavior. However, when applied to complex human systems that are involved in software maintenance and sustainment, the inappropriate use of metrics can have negative unintended consequences.

There are several strategies that can be used to mitigate the risk of unintended consequences. However, the most comprehensive mitigation strategy involves gaining a system-level understanding of the process that is being measured and using that understanding to identify likely responses to different measurement choices. The system-level understanding should also be used to monitor the measurement-induced effects so that, if necessary, corrective action can be taken.

To illustrate the concept, a simple CLD representation of the software maintenance and sustainment process has been developed. This indicates that, for example, focusing solely on metrics associated with the current release could have negative impacts on through-life cost-effectiveness. Although this observation (and others discussed in this article) may be helpful, the key conclusion is that any proposed set of software maintenance and sustainment metrics should be accompanied by the following:

- A system-level description of the process that is being measured.
- A description of how the metrics are intended to influence the system toward the desired behavior, including how they might interact to generate unintended consequences.
- An explanation of how the risk of unintended consequences will be mitigated. This should include a description of how the effects induced by the metrics will be monitored and how the selection of metrics will be altered if necessary.

Acknowledgements

Lieutenant Commander Steven "Dutch" Holland, R.N., provided valuable assistance as the concepts discussed in this article were developed.

Disclaimer and Copyright

ADDITIONAL READING

- Meadows, D.H. Thinking in Systems: A Primer. Chelsea Green Publishing, ISBN 1-603-58055-7, 2008.
- Sterman, J. Business Dynamics: Systems Thinking and Modeling for a Complex World. McGraw-Hill Higher Education, ISBN 0-071-17989-5, 2000.
- Coyle, R. System Dynamics Modelling: A Practical Approach. Chapman and Hall/CRC, ISBN 0-412-61710-2, 1996.

REFERENCES

1. Defense Acquisition University. Performance Based Logistics Community of Practice. Retrieved from https://acc.dau.mil/pbl. (Accessed 2015, February 10.)
2. Huff, L. & Novak, G. (2007). Performance-Based Software Sustainment for the F-35 Lightning II. CrossTalk – The Journal of Defense Software Engineering. 20 (12) 9-14.
3. Johnsen, Å. (2005). What Does 25 Years of Experience Tell Us About the State of Performance Measurement in Public Policy and Management? Public Money & Management. 25 (1) 9-17.
4. Mannion, R. & Braithwaite, J. (2012). Unintended consequences of performance measurement in healthcare: 20 salutary lessons from the English National Health Service. Internal Medicine Journal. 42 (5) 569-574.
5. Powell, A., et al. (2012). Unintended Consequences of Implementing a National Performance Measurement System into Local Practice. Journal of General Internal Medicine. 27 (4) 405412.
6. Smith, P. (1995). On the Unintended Consequences of Publishing Performance Data in the Public Sector. International Journal of Public Administration. 18 (2/3) 277310.
7. Cunningham, W. (1992). The WyCash Portfolio Management System. Proc. OOPSLA (Object-Oriented Programming, Systems, Languages & Applications), ACM. Retrieved from http://c2.com/doc/oopsla92.html.
8. Williams, B. J. & Carver, J. C. (2010). Characterizing Software Architecture Changes: A Systematic Review. Information and Software Technology 52, 31 51.
9. Ferguson, R., Phillips, M. & Sheard, S. (2014, January/February). Modeling Software Sustainment. CrossTalk, 19-22.
10. Brooks, F. P. (1995). The Mythical Man Month. Addison-Wesley, ISBN 0-201-83595-9.

ABOUT THE AUTHORS

Rob Ashmore is a principal software specialist at the U.K. Defence Science and Technology Laboratory (Dstl), a trading fund of the U.K. Ministry of Defence. He has over 20 years' experience in defense software, covering all aspects of the software life cycle. He holds both bachelor's and master's degrees from the University of Cambridge and is a Chartered Scientist (CSci) and a Fellow of the Institute of Mathematics and its Applications (FIMA).

Mike Standish is a senior engineer in systems at the U.K. Defence Science and Technology Laboratory (Dstl). He has gained experience of all aspects of software and systems life cycles through over 10 years within the defence sector. He holds a Bachelor of Science in software engineering and a Master of Science in Strategic Information Systems. He is currently undertaking an Engineering Doctorate in Systems. He is a Chartered Engineer (CEng) gained via the British Computer Society (BCS).