Applied Macromeasurements to Ensure On-Time Delivery, an Agile Approach to "Metrics"

Michael James, CST
Danube Technologies, Inc.
http://www.danube.com
michael@danube.com
3 January 2007

Abstract

This article challenges the value of traditional metrics for managing product development schedules and presents a reality-based alternative which is compatible with Agile approaches such as Scrum and XP. It is written for development managers or Scrum Product Owners who want to make decisions based on empirically-derived schedule forecasts instead of shooting in the dark.

Part I -- Why should Product Owners focus on the big picture?

Management by micromeasurements leads to micromanagement.

Examples of micromeasurements are:

- tracking intermediate development milestones such as requirements documents, design documents (UML, database schema diagrams, etc.), test plans, and other activities typically depicted on the Gantt charts of waterfall and RUP projects
- "metrics" such as number of Source Lines of Code (SLOC) created
- in Scrum, the daily Sprint Burndown Chart[1]
- tasks[2] and the actual hours expended on them

You get what you measure. If you are only interested in showing how hard you've tried to meet a goal, traditional metrics will serve you well. Unfortunately they tend to create "perverse incentives" that detract from the larger goals of delivering working maintainable product quickly.

We have observed that project after project will appear to be on track according to traditional tracking methods, only to disappoint customers when it comes time to show a functional product[3].

Does this mean the organization leader should cease all measurement and hope for the best? Certainly not! Agility is not anarchy -- agility entails a continuous feedback loop between the development team and the marketplace its product is targeted toward. Inspect and adapt! The Product Owner's feedback regarding the extent a team's activities meet organizational goals is necessary to avoid entropy. This leader's first responsibility is to provide vision and feedback to what extent the team's efforts satisfy the vision.

---

[1] The Sprint Burndown Chart is a valuable tool for team self management. Excessive management attention to team self-management artifacts will lead to finger-pointing and "looking good for the boss," impeding the candid interaction among team members necessary for hyperproductivity. Ref: http://www.danube.com/blog/michaeljames/Sprint_Burndown_Chart_vs_Product_Release_Burndown_Chart

[2] In Scrum, "Product Backlog Items" generally represent demonstrable functionality while "tasks" are the specific activities teams devise to complete them -- a "what vs. how" distinction.

[3] The Who. "Won't Get Fooled Again." *Who's Next.* London: Polydor Records, 1971

Part II -- What are some useful big-picture measurements?

If you're interested in meeting business objectives, your metrics and targets should focus on real, irrefutable progress. If you are ultimately interested in rapid sustainable shipment of functioning product, why not align your metrics accordingly?

Velocity

Scrum emphasizes demonstration of potentially-shippable product increments at regular, frequent intervals, starting with the first iteration (30 days or less). Demonstrable increments of functionality force end-to-end validation of the vision, the requirements, the implementation steps to meet them, and the schedule forecast. The Sprint Review Meeting at the end of every iteration provides an opportunity to measure *Velocity*.

- Velocity: The rate at which teams meet commitments to demonstrate potentially-shippable functionality, measured at fixed intervals and factoring in the effort estimates of the commitments.

The effort estimates can be expressed in time-based units, or abstract relative units known as "story points[4]."

When Does Short-term Velocity Translate Into Sustainable Velocity?

Extrapolations from short-term Velocity will not give useful predictions over the life of a project if the development team cuts corners to make shoddy demos. "Technical debt" drags down future Velocity by slowing the rate of delivering functionality or increasing the rate of bugs and regression failures[5].

Preventative measures that have proven successful against technical debt are the "Agile Engineering Practices": Test Driven Development (TDD), aggressive refactoring, continuous integration, and pair programming. Acceptance criteria that prevent technical debt by requiring these engineering practices every Sprint (iteration) contribute to a predictable and sustainable Velocity[6].

XP guru Ron Jeffries makes a similar point by advocating Running Tested Features (RTF) as a project metric[7]. The RTF metric is similar to Scrum Velocity when Product Backlog Items focus on features and include automated test in each item's acceptance criteria.

Note that Velocity measurements must be observed for several consecutive fixed-length Sprints before they can have any predictive value. Changes to team composition will invalidate the Velocity measurement. Even adding good people to a team may reduce Velocity.

Another caveat is that Velocity measurement works best when the team is developing end-to-end "potentially-shippable" functionality each Sprint, as demanded by the Scrum framework. It may not work for phasewise (waterfall) development such as doing database design one Sprint followed by user-interface development the next.

---

[4] Abstract relative "story points" eliminate the illusion of precision. Ironically, this results in more accurate forecasts when combined with empirical Velocity. Ref: ScrumWorks Pro *Backlog Estimation Guide* http://danube.com/sw_flash/bpg/BPG_Backlog_Estimation.html

[5] http://danube.com/blog/kanemar/technical_debt_and_the_death_of_design_part_1.html

[6] http://danube.com/blog/michaeljames/how_to_survive_technical_debt

[7] Ron Jeffries, "A Metric Leading to Agility" http://www.xprogramming.com/xpmag/jatRtsMetric.htm

Measurement of requirements change (p.k.a. "Scope Creep")

The frustration with "scope creep" is rooted in the illusion that requirements can be completely known at the start of a project. Agilists embrace the reality that our understanding of what we want to build changes as we build it. It is useful to measure the rate of requirements change, both for an expected release (which may be tied to a fixed date), and for the entire product. Comparing the rate of requirements change to the development team's Velocity can help adjust scope for a fixed release date, or predict the completion date of a "fixed-scope" project.

Earned Business Value (EBV)

In Scrum the Product Owner prioritizes requirements, generally according to anticipated Return On Investment (ROI). The Product Owner may estimate "Business Value" representing the expected return of each Product Backlog Item. Some Product Owners may prefer to think of Business Value in dollar terms, while others will prefer relative units. ROI of each item can be estimated by dividing the Business Value by the team's effort estimate. When Product Backlog Items are proved to be completed (per their acceptance criteria), they contribute to *Earned Business Value*[8].

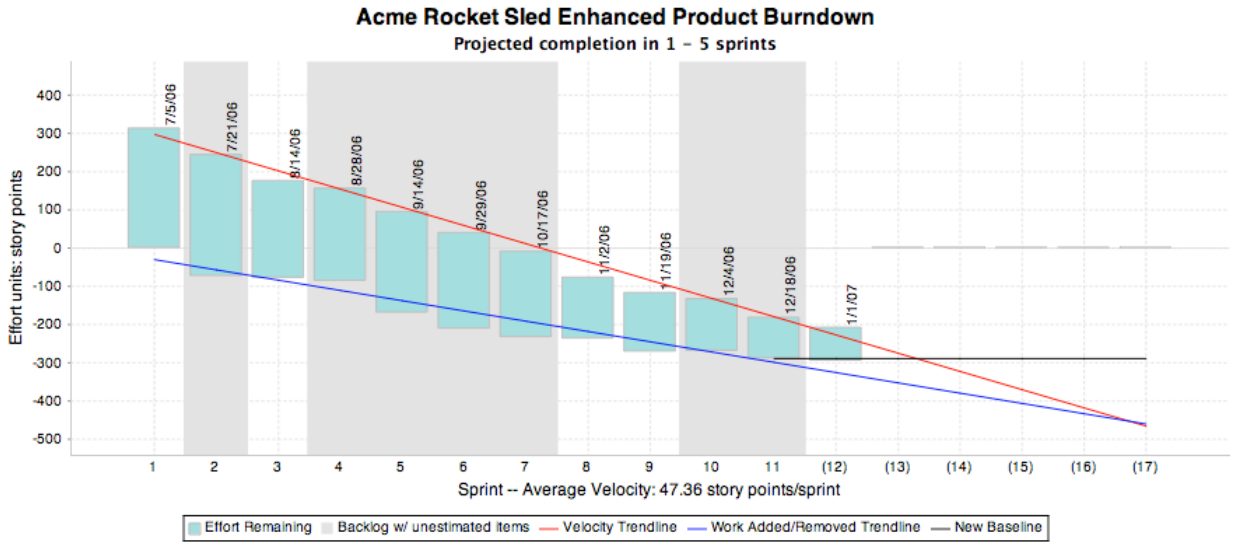
- Earned Business Value (EBV): the percentage of the known desired business value that is coded, tested, documented, and potentially shippable (or what ever "done/done/done" means on your project).

EBV is sometimes referred to as "Earned Value Metrics" (EVM).

Part III -- Representing Macromeasurements Visually

Product and Release Burndown Charts

ScrumWorks Basic™ and ScrumWorks Pro™ generate Enhanced Product Burndown Charts[9] showing Velocity and the rate of requirements change over a range of Sprints, as shown below.



[8] Dan Rawsthorne, "Calculating Earned Business Value For An Agile Project" http://www.agilejournal.com/content/view/54/

[9] Derived from Mike Cohn's "Alternative Release Burndown Chart." Ref: Cohn, M. *Agile Estimation and Planning.* Prentice Hall, 2005.

This shows progression of the product or release backlog size from one Sprint iteration to the next. The height of each bar represents the total amount of uncompleted work known at that beginning of each Sprint. As the team completes work, it's chopped off the top of the bar. Work added to the backlog from one Sprint to the next (requirements change) is tacked onto the bottom of the bar.

The red line (on top) represents Velocity as earlier described. The blue line (underneath) represents the rate of requirements change -- typically new "stories" added to the backlog. If these lines converge, one can project a completion date based on historically observed trends. Use of empirically-observed measurement has greater predictive value than traditional (i.e. speculative) approaches to scheduling. As stated above, the more history the better the predictive value. We have observed that at least four Sprints must be completed before one can have any confidence Velocity trends will continue.

Measuring the *actual* rate teams deliver working functionality can eliminate the need for accurate time-based estimates.

This graph can be produced either for an entire product development cycle, or for a release or subset of releases. A Product Owner facing a fixed-date release can use this graph to continuously tune the expected scope to hit the release date.

In the example above, the Product Owner is aggressively managing release scope to work with a fixed release date. This entails some re-prioritization *every Sprint* as new needs appear on the backlog. Re-prioritization for a fixed release date often includes painful decisions to move backlog items into future releases (or into the freezer). A faster Velocity wouldn't necessarily solve this -- product development can be a kind of Red Queen's race[10] where every added capability creates a demand for two more.

Non-convergent Velocity and Scope Creep

The example below is a new team moving through the "forming, storming, norming, performing" growth stages, and more typical of what this author sees when consulting at client sites who have recently adopted Scrum. Team Velocity is increasing and may not have stabilized yet. The rate of scope increase has slowed slightly. But the lines appear unlikely to converge any time soon. Management's options include:

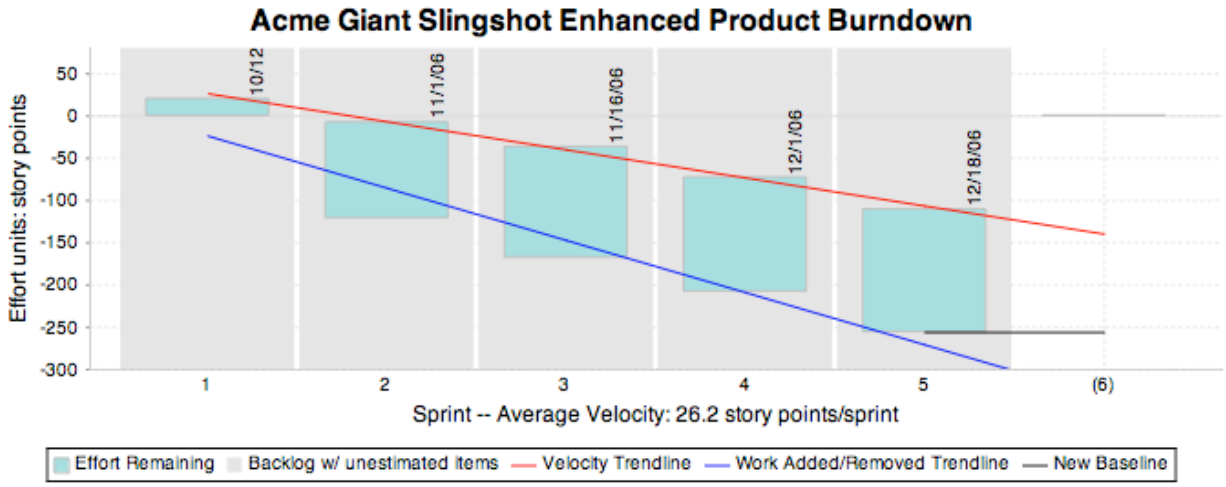
- trying to increase the rate of Velocity increase (acceleration, or Δv) by removing team-reported impediments,
- cutting scope of the pending release
- adding additional teams[11]
- cancelling development[12]

---

[10] http://en.wikipedia.org/wiki/Red_Queen's_race

[11] Increasing one team's size much beyond seven people usually reduces its effectiveness.

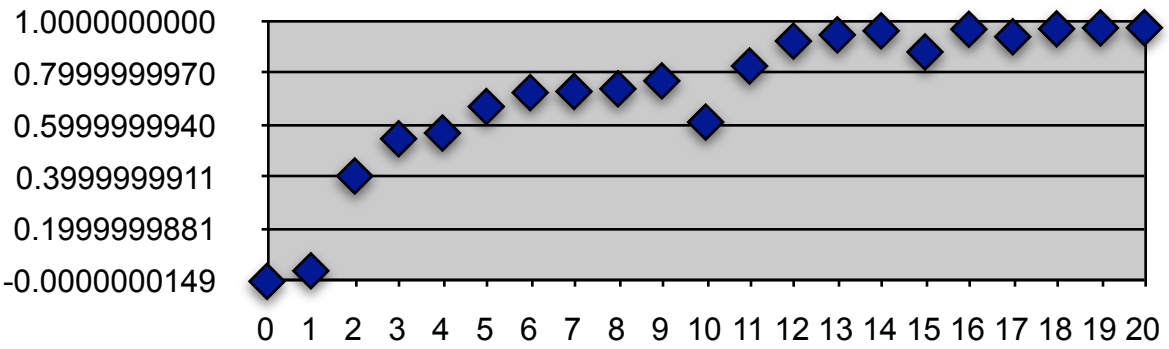
[12] Causing development to "fail fast" and get cancelled early instead of years and millions of dollars too late is a type of success.



Earned Business Value (EBV) Graph

The following graph shows Earned Business Value (as previously defined) from one Sprint to the next for a real project[13]. The X axis represents Sprint iterations. In this particular example the team, working at a nearly constant Velocity (not shown), delivered most of the business value within the first half of the project due to ROI-inspired prioritization by the Product Owner.

The downtick at Sprint 10 and quick recovery by Sprint 11 represent the discovery that important (but easy to build) functionality was missing after the product went to alpha release. The high ROI (high business value / low development effort) of the missing functionality caused the EBV spike at Sprint 11. Frequent reality checks with end users (validation) increase the Product Owner's opportunity to discover high ROI items sooner. Agile approaches are skeptical and validation-centric for this reason. One way to prevent late discovery of important requirements is to increase stakeholder attendance at each Sprint Review Meeting.



EBV graphing is planned for a future version[14] of ScrumWorks Pro™.

---

[13] Case study provided by Dan Rawsthorne, Ph.D., Danube Technologies, Inc.

[14] This article was written in January 2007, before the ScrumWorks Pro release. For the latest list of features, visit http://scrumworkspro.com

Part IV -- Conclusion

Examining macromeasurements on a frequent basis (such as once per iteration) allows the Product Owner to inspect and adapt product and release plans based on empirical data without hampering team self organization.