Practices for Scaling Lean & Agile Development

Large, Multisite, and Offshore Product Development with Large-Scale Scrum

Craig Larman
Bas Vodde

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

but…there is still some delay in integration and testing, and Undone Work. By retaining component teams—that create dependency and integration problems—system-level delivery of value is still slow, and there are more queues, WIP, handoff, and multitasking. Related to the prior experiment, "Try…Avoid…Early and incremental handoff of Undone Work" section on page 179, a release train could involve some handoff to an Undone Unit, such as a separate system-level testing group.

Try—If, for some reason, the group is unable to adopt feature teams with continuous integration, then a release train approach is at *least* better than a classic sequential life cycle.

ESTIMATION

Try…Estimate with Story Points

In *Agile Estimating and Planning*, Mike Cohn makes a good argument in favor of using *story points* (relative effort points) for estimation. We do not repeat the motivation, but note that this advice is even more compelling in development with many teams. Why? In this case, other—more detailed—estimation techniques (such as work-breakdown person-day estimates) are extraordinarily slow and laborious. Thus, there is disinclination to refresh the estimates—even though *updated estimates* are useful in planning, and are part of Product Backlog refinement each iteration.

But story points, especially when combined with techniques such as *planning poker*, are *relatively* quick and easy to refresh. Especially in large-scale development, that translates to an increased chance that re-estimation will occur each iteration.

Problem: Story points are relative—'5' has no absolute independent meaning. Two teams can define '5' differently, which makes it difficult to estimate overall effort or track overall progress. Therefore…

Try…Avoid…Synchronize points and range

If the product group makes a common agreement on the size of story points[14]—so that a '5' is the same for all teams—there are benefits:

- a common Release Burndown chart—better view of progress
- a product-level velocity—better prediction of future progress
- a team's estimate of items that can be done by other teams
- the time together to create a common agreement that builds more shared understanding and cross-team relationships

How to define a common point, and a common range of points? One approach is…

Cross-team Estimation Workshops for Canonical Set—All members—or team representatives—of all teams join in a common estimation workshop and identify items for which they have *common understanding*. Then they estimate these, using planning poker with story points. This *canonical set* of items is a baseline of shared understanding and is used as the baseline of future estimation workshops, including those done separately by individual teams or larger sub-groups. This cross-team estimation workshop occurs not only before the first iteration, but repeatedly in subsequent ones to resynchronize.

There are at least two ways that the group can have common understanding of the *meaning* or *effort* in a canonical set:

- Historical set of completed items is reviewed. Likely the best choice, since the items were done and so have the clearest information.
  - this gives common understanding of effort but not necessarily of meaning
- *New* items are first analyzed together by the estimators in a previous requirements workshop.

14. A common agreement also fosters overall product-level perspective among the teams, rather than isolated-team mindset.

Issues—Synchronizing points across teams works *relatively* well, but requires occasionally repeating cross-team estimation workshops,[15] since they drift out of synchronization. Synchronizing also brings with it a few potential dangers:

- *comparing*—we have seen management groups start to compare the relative 'performance' of teams that use a common synchronization, even though such comparisons are in reality meaningless—and worse, harmful.
- *converting*—one merit to story points is that they have no absolute or independent meaning. They cannot be realistically converted into person-days, for example—and the attempt to do so by management (or others) indicates that the purpose of their use in *velocity* has not been understood. We have seen teams attempt to synchronize by stating, "one point means two person-days." This is a slippery slope to a deep misunderstanding of story points and velocity.

We recommending trying cross-team synchronization on points with a canonical set. But if these dangers are manifest, there is an alternative: *Each team uses its own idiosyncratic definition of points*. However, this leads to problems in estimating overall effort and tracking overall progress across all teams. One partial solution is…

Try…Combine progress measures

If different teams (or larger requirement areas) use different point systems, then each has its own Release Burndown chart, and there is no common product-level Release Burndown chart and no common velocity. How to estimate progress? Some alternatives:

- The simplest approach we have used is to hang the separate Release Burndown charts on a wall, stacked directly underneath each other, and 'eyeball' the overall scene.
- Estimate the percentage of progress, based on points, of each group. For example, if team-1 has 100 points in their Release Backlog and have finished 20, they are "20% complete."[16] Over-

15. These workshops need not be cast as a problem; positively, they foster common understanding and product-level perspective.

all product progress can be defined with different *combination methods*: worst case, mean, or median, depending on context.[17]

Try…Avoid…Estimate velocity before iteration-1

Real *velocity* is not estimated, it is measured—it is the total of all story point estimates of the work completed in the last iteration, a historical measure.[18] Yet, sometimes a group wants to estimate it before iteration-1—before it can be measured—to help predict total release duration. For instance, the group may want to estimate release cost, and use duration as one factor. *Agile Estimating and Planning* [Cohn05] offers techniques for its estimation in the small-scale case.

For the multiteam case, one approach we have used is for all teams to hold a *pretend* (planned but not executed) Sprint Planning Part One and Part Two meeting, and assume that the items chosen for implementation were done—or that 75% (for example) were done. The total story points of all these items, across all teams, is the estimate of product-level velocity.

Avoid—If the next release of a product is going to be done anyway, rather than spending a week before iteration-1 *estimating* the velocity, just immediately *do iteration-1*, and measure *real* velocity.

Try…Adjust duration estimate with Monte Carlo simulation

In a tiny release with one team and four months estimated duration, plenty can go wrong to invalidate the estimate. This variability is magnified in the large scale. For some product groups we work with,

16. This is actually a fiction, because of variability and because it assumes the Release Backlog cannot be descoped.
17. As an example of context and combination method: If team-1 does COBOL features in Beijing and team-2 does Java features in São Paolo and they refuse to talk to each other (and cannot, anyway)…perhaps *worst-case* is the combination method. This is an extreme example; in very big groups, *mean* suffices.
18. Velocity is a kind of *budgeted cost of* ***work performed*** (BCWP) in earned value management [FK06]; as a name, 'BCWP' emphasizes that it is a historical measure, not an estimate.

that is not a problem—vague confidence in a fuzzy duration estimate is sufficient. In others—especially fixed-price, fixed-scope, fixed-duration outsourced work—it is vital to do the best one can to estimate duration.

Once when we were coaching in Bangalore, a lead developer's father died, and he had to leave for several weeks. Once in Budapest, a lead developer's girlfriend left him, and he was pretty useless on the team for a while—until he found a new girlfriend! These are examples of risks and sources of variability. And there are others: scope change, productivity, attrition, snowstorms, and more.

How do scientists and other estimators model stochastic (probabilistic) systems, such as development work? A standard solution is *Monte Carlo simulation* (MCS). With an MCS model for release duration, you can include the impact of many elements of variability or risk, including scope change, attrition—and girlfriends. MCS is especially useful in very large and long-duration development and in offshore, outsourced, fixed-price, fixed-duration projects.

And MCS fits well in the context of *agile estimating* because it is a simple, lightweight, fast technique to apply and adjust.

A classic book on product development risk management and MCS is *Waltzing with Bears* [DL03]; this explains the method of MCS for estimating a more realistic duration. The authors, DeMarco and Lister, also provide a free Excel spreadsheet called *Riskology* that implements an MCS model. Find *Riskology* on the web and try it, to improve the realism of your estimates.

CONCLUSION

Planning large-scale agile development is simpler than traditional approaches—or at least should be. We notice that this simplicity is disconcerting for some, because the traditional paradigm of management is that big complex work needs big complex planning and control by project managers. But there is a different way: the *emergence of order* from self-organizing Scrum feature teams. Top-down planning and control is not particularly effective in systems with variability and discovery—because the plans assume something

relatively static or deterministic—and the approach grows even less effective as these *non-linear* systems grow larger.[19] Complex systems on the boundary of chaos and order (*chaordic* systems [Hock99])—and that includes big development groups—cannot be truly planned or controlled from above.

No false dichotomy regarding "no planning" versus "top-down planning"…In Scrum, the group *does* start by creating a Release Backlog for a future goal, identifying and estimating the product features. But after that starting point, agile planning emphasizes continual learning and adapting.

How to do that within a large chaordic system? By (1) encouraging self-organization and bottom-up emergence of order, (2) increasing transparency and feedback, and (3) making it easy to frequently inspect and adapt. This is precisely what agile approaches such as Scrum offer. Consequently—in contrast to those that assumed agile development was for *small* groups—agile planning is especially useful for the *large* scale.

Recommended Readings

- For envisioning and vision workshops, two books already recommended in the *Product Management* chapter are relevant: *Innovation Games* and *Agile Product Management with Scrum*.
- For planning with small or large groups, *Agile Estimating and Planning* by Mike Cohn is an excellent, practical resource.
- *Waltzing with Bears* by DeMarco and Lister is informative and entertaining; it emphasizes iterative—rather than sequential—development as a key risk-management practice, and explains how to apply Monte Carlo simulation in estimation.

19. This was explored in *Queueing Theory* in the companion book.