Disciplined Learning The Successor to Risk Management

Alistair Cockburn, Humans and Technology

Abstract. Disciplined learning, or "learn early, learn often," updates naïve agile development and traditional risk management, and safely replaces the dreaded catch phrase, "fail early fail often." Disciplined learning is a rich, creative and rewarding endeavor, already in use in small pockets of excellence.

Introduction

Naïve agile development works remarkably well, given how simple it is. It is less than optimal, however, and insufficient for many situations. Disciplined learning adds to agile.

Traditional risk-management generally addresses how to avoid failure rather than how deliver success. Disciplined learning updates risk management by incorporating some of the principles of agile development.

Disciplined learning is neither obvious nor for the faint of heart, but it is in active use by top teams in many disciplines, who manage to deliver success in difficult circumstances.

Consider, as a reference point, the still-common way of working in which a major integration or delivery occurs at the end of a long period of work without integration or delivery (see Figure 1). It is not necessary to be working in a waterfall fashion to have this moment of integration or delivery in the project, so the curve need not be ascribed to waterfall. It is a simply a common strategy.

Figure 1 shows time on the horizontal axis. The dotted line shows project costs increasing steadily over time. The solid line shows that learning progresses while the project teams work, talk, design, but not in the major way that learning (and surprises) occur immediately after the moment of integration or delivery.

Learning occurs relatively late in the project, after most of the cost has been accrued.

What we are after is how to learn earlier in the project, when

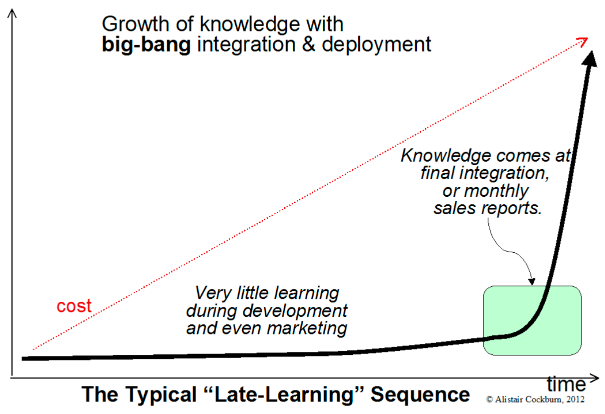

Figure 1. The typical "late-learning' strategy.

changes can be made with lower cost. This is where creativity and discipline come in.

Four Learning Topics

The team has (at least) four categories in which to learn:

- What they should really be building, never mind what they thought they should build at the start.
- Whether they have the right people on the team, and for those people, how best to work together.
- Where their technical ideas are flawed.
- How much it will cost to develop.

In the strategy shown in Figure 1, these are all learned late in the project, around the time when the parts are integrated and deployed, when the consumers finally give feedback on the result. This learning arrives too late to benefit the product.

The disciplined learning approach is to apply the same "broken" learning curve in very small doses, deliberately and often, so that each step provides information that can be used to adjust the four categories of learning. The payoff is not just reduced risk in the final delivery, but the ability of the sponsors to steer the final delivery in a fine-grained way, both in delivery time and delivered features and quality.

Figure 2 illustrates the disciplined learning approach. The following four sections describe strategies for learning in the four categories.

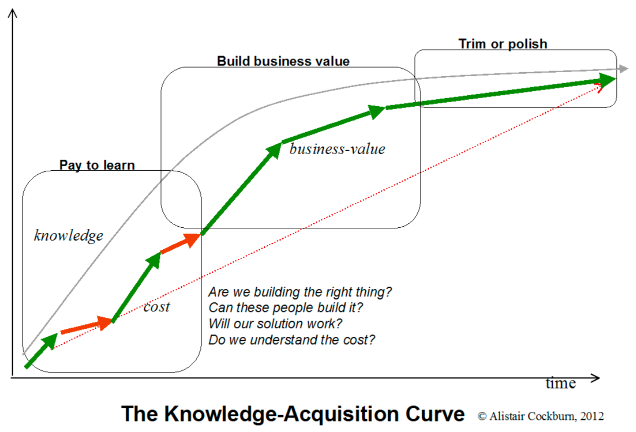

Figure 2 Applying the principle: Learn Early, Learn Often.

Learn What Should Get Built

The most important and most difficult question is: Will people like, buy and use what we're building?

Normally, this question gets answered when it is too late. Recently, however, strategies have come into usage that move this learning process forward. The strategies are fairly simple, but require discipline, patience, and a willingness to change course based on the results.

Sample strategies are:

- Paper prototyping.
- Ambassador user.
- Early delivery.
- Empty or manual delivery.

Paper prototyping [1] and related strategies coming from the user-centered design community [2] involve nothing more complicated than putting a mockup of the product into the hands of the consumer, who reacts to these early design thoughts. Prepared at low cost, early in the development cycle, these prototypes allow the development team to change their minds about how to proceed.

An "ambassador user" is a friendly user to whom the team can deliver an incomplete but growing product. This user usually breaks the system within moments, and give valuable feedback from his or her (limited) perspective. The difference between the "ambassador user" and "paper prototyping" is that the ambassador user is encountering the actual system as it grows, not a mockup of the system.

"Early delivery" is a full deployment of the system with reduced capabilities. The intention is to learn, first of all, what is incorrect with the product as envisioned, but possibly more significantly, how the presence of the product changes the thoughts about what should be built in the first place. "Early delivery" recognizes that once people start using a system, their habits and needs change, often in unpredictable ways. Delivering a thin version of the system early allows the development team to gather new input and adjust the priorities on what should be developed.

The above are all standard albeit frequently ignored techniques, found in the regular and agile literature.

The most interesting strategies to emerge in the last decade are two documented and practiced in the lean startup community: Empty and Manual delivery (my terms for them).

The "Empty Delivery" [3] strategy is particularly well suited for online products. Initially, all that is detected is whether anyone clicks on a link or accesses a feature. There is no implementation behind the façade of the click. Measuring these clicks, a team can reduce or sequence the features developed to follow those drawing the most attention. The system evolves in the direction of maximum draw.

"Manual Delivery" is described in Eric Ries' book, The Lean Startup [4]. In this strategy, a team spends what may seem to be excessive money even delivering products manually, for the simple reason that manual procedures can be set up and changed for very little cost. Delivering manually, the team can change the product offering with every single purchase, evolving to what the customer base indicates is really desired.

Adjust Design Decisions

Mistakes in design come from:

- Choosing technology that doesn't work as advertised.
- Mistakes due to people not talking to each other, with resultant mistaken assumptions about each other's work.
- Inevitable omissions and mistakes in design.

These mistakes are discovered and repaired using strategies:

- Walking skeleton.
- Micro-incremental development.
- Spikes.
- Story splitting.

The "Walking Skeleton" strategy [5] calls for the team to connect a thin path through the architecture. In creating this simple but full system, they discover the first round of surprises in the technologies they are using.

Once the system is thinly connected, the infrastructure and functionality teams each adds onto their part of the system. It is not uncommon to see the infrastructure team redesigning the skeleton itself, while keeping the interfaces to the functionality running (or forcing updates). This restructuring is one of the costs of using the strategy.

Micro-incremental development is when teams integrate their work every hour, half-day, or day. The shorter the time between integrations, the faster they find mistakes, and the lower the cost of making changes. A side benefit is that they are less likely to change the same part of the design at the same time, and so they do not need to check out and branch the design, making integration easier, faster, and less error prone.

A spike [6,7] is a small, disposable piece of work created to explicitly address the question, "Is there an obvious flaw in this approach?" It is used to flush out interface mismatches as well as various performance and scaling problems.

The difference between a spike and ordinary incremental development is that ordinary incremental development is conducted using full production conventions, with the assumption that the work will be used in the final product. A spikes must absolutely not be used in the final product; it is throwaway work. Because the work is throwaway, it is always done in the most rapid and effective manner possible with the sole purpose of learning about the question at hand.

Some questions might seem impossible to move forward in the schedule, such as the final conversion of the database. With story splitting [8] a story is split into a learning (spike) piece and a production piece. The spike is placed early to learn how to address whatever difficulties might lie in it. Then the actual work can be left until the appropriate moment in the schedule.

Learn to Work Together

Failure to deliver is sometimes due not to the people being not correct for the assignment, but to them not having learned how to work together. Tom DeMarco and Tim Lister refer to a "jelled team" [9]. Three strategies help with creating a jelled team:

- Early victory.
- Walking skeleton.
- Simplest first, worst second.

The Early Victory [10] strategy is based on the work of sociologist Karl Weick [11], showing that achieving results helps people come to trust each other more, raises morale and helps them perform better.

The "walking skeleton" already described produces an early technical victory to the team and to the sponsors. The concept is sometimes adjusted to implement and deliver a thin path through the workflow of a company, with similar "early victory" and technical learning for the delivery and work flow aspects of the project.

The "simplest-first, worst second" strategy [12] is contrary to the usual recommendation in the agile development world. The usual agile advice is to build the highest business value first. That strategy makes good sense once the team is functioning well, social risks have been reduced, and the team is capable and confident of being able to deliver whatever is of the highest business value. However, many conversations need to take place before the team has reached that point. For this reason, it is sometime useful to build something real but very simple, so that they can adjust social habits in good time before the difficult parts of the project are reached.

Learn How Much It Will Cost

Two strategies help with learning the cost of a project:

- Core samples.
- Microcosm.

Tim Lister told the following story at a conference [13], "A man wanting a pool built in his back yard calls in three contractors to present estimates. The third contractor, instead of presenting an estimate, tells the homeowner he will need to drill and core sample in the ground, and will charge the man for that. The homeowner complains, saying that the first two contractors didn't charge him for core sampling. The contractor responds that he has no idea how the first two contractors could submit a bid, since they don't know what sorts of rock layer lies under the lawn, but he couldn't possibly put in a bid without having that information. The homeowner now comfortable with the third contractor, hires him for the work."

To do this with a development project, isolate parts of the system the development of which is not obvious and develop very small elements within those areas. In that development, identify what sorts of surprises lurk below the surface and understand how difficult the work will really be. Carefully selecting such "core samples" allows the team to develop a more reliable cost-, time-, and resource estimate for the project.

Core sampling is the miniature version of the more general "Microcosm" strategy [14], in which a mini-project is run for the sole purpose of establishing a sound estimate. A full Microcosm project can be set up to test the productivity of a new development team (think off-shoring, in particular), as well as to test the learning speed of staff with new technologies, to benchmark the productivity of expert versus ordinary or new developers.

Whereas a core sample effort is intended to take hours to days, a full Microcosm project may take weeks to carry out, and should therefore only be used for larger development efforts.

Creating a Plan

In the light of these strategies, the creation of a project plan is rather different than before.

Disciplined learning calls for merging learning steps from the four categories above with requests for growth of business value as is standard with incremental development. Business value and learning are artfully interleaved into a sequence of work assignments designed to reduce risk, deliver crucial information, and develop product capability in an "optimal" way.

This is where creativity enters.

The quality of the plan is sensitive to the ability of the planners to identify and merge the learning needs and the upcoming possibilities for income. As lessons are learned and new risks and opportunities spotted, the project will need to be updated.

Trimming the Tail

A product feature actually consists of three parts, not just the two:

- Learning.
- Value.
- Tail.

The "tail" is the polishing and glossing that makes a feature "wonderful." Since not every feature is of equal value to the buyers and users, many or even most features can be thinned or trimmed back without damage to the system.

Attending to the presence of a tail, a team can arrange for a minimum set of features to be at an "adequate" level of wonderfulness in plenty of time before final delivery, then spend the remaining time polishing and glossing those feature that are more important than the others [15]. Alternatively, if time is short, they can cut back on (trim) the polishing and deliver early or on time [16]. This is described in the final section.

Reaping the Benefits

Disciplined learning delivers two benefits: early income and the ability to trim the tail.

Early income from incremental development is well presented in Software by Numbers [17]. A project can become self-funding if it is delivered to paying users part-way through its development, thus lowering the load on the sponsors.

Less obvious but equally valuable is the ability to not develop less valuable aspects of the system. Here is the shortest example, to give the idea:

When you are opening a new hotel, it may not be necessary to shine the doorknobs before opening to the public. If it is necessary to have shined doorknobs for the guests, it is probably not necessary that all of the doorknobs need be shined.

You might trim any of four aspects of a system:

- Features.
- Feature details.
- Usage quality.
- Internal quality.

You drop an entire feature. A car (for example) might not need a sunroof. The first iPads did not have phone modems.

If not an entire feature, you might be able to trim an aspect of a feature: Given that your car must have all of the basics (such as brakes), it might not need brakes with antilock braking. A computer system might require searching capability, but not auto-completion or auto-correction.

Recognizing that really smooth and easy to use features take a lot of work, you might choose to skip improving usability for selected features.

Finally, you can trim internal design quality and correctness. The question is how much internal quality is needed for the delivery in question.

If development has proceeded incrementally, attending to the learning areas, then the team can deliver:

- Early, with reduced features or quality.
- On time, with either full or reduced quality, depending on where development stands at that time.
- Or later, with enriched features or quality; at the choice of the sponsors!

Under usual project circumstances, the only choices are to delay or work overtime. The "trim the tail" option is available only for those who have worked in this more disciplined fashion.

Disciplined learning with trim-the-tail is one of the few approaches equally available to very small and very large projects, fixed-price and floating-price projects. Here are three examples, taken from real projects:

1. Small, floating-price project: A web site development involving only the web site owner and the programmer. After several months of open-ended work, the web site owner wanted the site delivered "soon," and trimmed the tail back aggressively and repeatedly until something much smaller than expected but still suitable was deployed.

2. Small, fixed-price project: The company in question always bid small, fixed-price contracts of three- to six-months, involving three to eight people. As usual, the bids were aggressive and the teams typically ended late, missing the deadline or scope, with resulting overtime from the developers and penalties at the end of the contract. Jeff Patton [18] worked in the manner described in this article, leaving the least important features to the end, and deliberately thinning the less critical features, so that when the contract period ended, it was clear to the customers that they had gotten most of what they wanted. This produced the least overtime, the smallest penalties, the highest customer satisfaction and the greatest likelihood of receiving a follow-on contract.

3. Very large development project: A company with several thousand developers in several countries, working on a product line with multiple variations, applications and releases. Under normal circumstances, when they call for a full integration on a particular date, every team starts to work overtime and jockey for position not to be the one most behind schedule. The integration date keeps getting slipped back as team after team fails to complete their work on time. Using the trim-the-tail approach, each team would have in place the essential elements needed for the integration, with only tail elements left unfinished. For delivery, management would be in position to deliver slightly less, on time, or slightly more, a bit later.

It is exciting to find a baseline strategy that applies to projects of such different sizes and natures as just outlined.

Disciplined learning is not for the faint of heart. It requires discipline, creativity and constant correction. The payoff is the ability to get a team working together, discover what is needed in time, deliver it early in order to create a self-funding project, and finally, trim the tail at the end to meet inelastic deadlines.

ABOUT THE AUTHOR

Dr. Alistair Cockburn, one of the creators of the Manifesto for Agile Software Development, was voted one of the "The All-Time Top 150 i-Technology Heroes" in 2007 for his pioneering work in use cases and agile software development. An renowned IT strategist and author of the Jolt award-winning books "Agile Software Development" and "Writing Effective Use Cases," he is an expert on agile development, use cases, process design, project management, and object-oriented design. In 2001 he co-authored the Agile Manifesto, in 2003 he created the Agile Development Conference, in 2005 he co-founded the Agile Project Leadership Network, in 2010 he co-founded the International Consortium for Agile. Many of his articles, talks, poems and blog are online at <http://alistair.cockburn.us>.

E-mail: totheralistair@aol.com

REFERENCES

1. <http://en.wikipedia.org/wiki/Paper_prototyping>
2. <http://en.wikipedia.org/wiki/User-centered_design>
3. BBC "Searching the internet's long tail and finding parrot cages," <http://www.bbc.co.uk/news/business-11495839>
4. Reis, E., The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses, Crown Business, 2011.
5. <http://alistair.cockburn.us/Walking+skeleton>
6. <http://c2.com/xp/SpikeSolution.html>
7. <http://agiledictionary.com/209/spike/>
8. <http://alistair.cockburn.us/The+A-B+work+split>
9. Tom DeMarco and Timothy Lister, Peopleware: Productive Projects and Teams. New York: Dorset House Publishing Co., 1987.
10. <http://alistair.cockburn.us/Advancedpmstrategies1-180.ppt>
11. Karl Weick, The Social Psychology of Organizing, McGraw-Hill Humanities/Social Sciences/Languages; 2nd edition, 1979.
12. Alistair Cockburn, Crystal Clear: A Human-Powered Methodology for Small Teams, Addison-Wesley, 2005. Also online at <http://alistair.cockburn.us/ASD+book+extract%3A+%22Individuals%22>
13. Lister, Tim, keynote at Agile Development Conference 2010.
14. <http://alistair.cockburn.us/Project+risk+reduction+patterns>
15. <http://www.agileproductdesign.com/downloads/patton_embrace_uncertainty_optimized.ppt>
16. <http://alistair.cockburn.us/Trim+the+Tail>
17. Mark Denne and Jane Cleland-Huang. Software by Numbers: Low-Risk, High-Return Development. Prentice-Hall, 2003.
18. Jeff Patton, "Unfixing the Fixed Scope Project: Using Agile Methodologies to Create Flexibility in Project Scope," in Agile Development Conference 2003, Proceedings of the Conference on Agile Development, 2003, ACM Press. Available online through a Google docs search.