Learning From Agile Software Development – Part Two

Alistair Cockburn
Humans and Technology

This two-part article compares agile, plan-driven, and cost-sensitive software development approaches based on a set of project organization principles, extracting from them ideas for pulling agile techniques into cost- and plan-driven projects. Part one, which appeared in October's CrossTalk, *described how the different teams make trade-offs of money for information or for flexibility, and presented the first seven of 10 principles for tuning a project to meet various prioritization of cost, correctness, predictability, speed, and agility. This month, part two presents the last three principles, then pulls the material together for actions that plan-driven and cost-sensitive project teams can use to improve their strategies and hedge against surprises.*

Being agile is a declaration of priorities, prioritizing for project maneuverability with respect to shifting requirements, shifting technology, and shifting understanding of the situation. Other priorities that might override agility include predictability, cost, schedule, process-accreditation, or use of specific tools.

Most managers run a portfolio of projects having a mix of those priorities and need to mix their strategies to suit. The question at hand is how a person can borrow from the set of agile practices to fit the plan-driven and cost-sensitive programs.

Part one of this article [1] introduced two phrases:

1. *Money-for-information* (MFI) issues are those on which the team can spend money now to obtain information that puts them in a better situation for later in the project. Work-plan breakdown structures, system performance under load, and user reaction to system design are MFI issues.
2. *Money-for-flexibility* (MFF) issues are those on which the team cannot possibly obtain information now to put them in a better situation later. The better strategy is to spend money on making the change easier later. Movements in the stock market, emerging standards, and staff continuity are MFF issues.

Many of the differences between agile methodologies and cost- and plan-driven approaches are in deciding which issues are MFF or MFI issues, and what the best allocation of resource is for each.

A plan-driven team might decide that the project plan is predictable, and a good MFI strategy is to spend energy now to make those predictions. An agile team might decide that the project plan fundamentally cannot be resolved past a very simple approximation, and therefore a MFI strategy is a waste of money. Instead, they adopt a MFF approach, which involves making many coarse-grained plans over the course of the project.

Both teams might agree that the question of system performance under load is an important MFI issue, and both might agree to spend money early to build a simple system simulator and load generator to stress test the design.

> *"The most important agile value for the cost-sensitive project leader to adopt is customer collaboration."*

Ten Principles

The following 10 principles have shown themselves useful in setting up and running projects:

1. Different projects need different methodology trade-offs.
2. A little methodology does a lot of good; after that, weight is costly.
3. Larger teams need more communication elements.
4. Projects dealing with greater potential damage need more validation elements.
5. Formality, process, and documentation are not substitutes for discipline, skill, and understanding.
6. Interactive, face-to-face communication is the cheapest and fastest channel for exchanging information.
7. Increasing feedback and communication reduces the need for intermediate work products.
8. Concurrent and serial development exchange development cost for speed and flexibility.
9. Efficiency is expendable in non-bottleneck activities.
10. Sweet spots speed development.

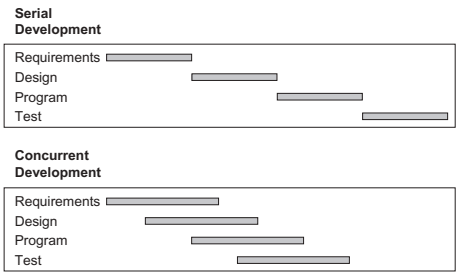
The first seven principles were addressed in the October issue of CrossTalk. I pick up the discussion here with Principle No. 8.

8. Concurrent and Serial Development Exchange Development Cost for Speed and Flexibility

On a predictable project, the project coordinator can arrange for each work specialist to show up at just the right moment, perform the needed work, and leave. Such scheduling, common in the construction and book publishing industries, minimizes salary cost in exchange for extending elapsed time (see Figure 1, Serial Development). The hazard is that a surprise might show up in an already-completed task forcing the previous task item to restart, in which case neither time nor cost is minimized.

Concurrent development runs teams in parallel, even when they have dependencies between them [2]. The teams will make and change decisions as they gain information, causing the other teams some rework. With careful management of their dependencies, the teams can complete the

Figure 1: *Serial Development vs. Concurrent Development* [2]



Note: Serialized development takes longer but costs less than concurrent development.

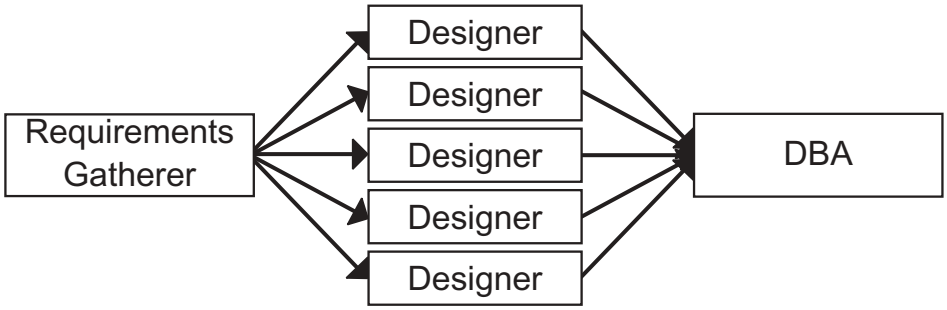


Figure 2: *The Database Analyst (DBA) Bottlenecking Five Designers* [2]

final work sooner even though their salary costs are higher (see Figure 1, Concurrent Development). Effective concurrent development demands that communication between people is fast, rich, and inexpensive (as discussed in principles 6 and 7). The hazard in concurrent development is that if work is started too early, rework costs dominate the project.

Serial and concurrent development have opposing characteristics. Cost-sensitive projects should use serial development where they can, while projects sensitive to shifting requirements benefit more from concurrent development

Agile project teams almost always use concurrent development assuming a significant number of surprises will arise during development. The close communication needed for effective concurrent development also lets them respond to late-breaking changes effectively.

9. Efficiency Is Expendable in Non-Bottleneck Activities

Effective concurrent development requires calculating the moment at which to start a downstream activity. Goldratt's process theory [3] and theory of constraints [4] provide advice here.

Suppose that one requirements gatherer feeds information to five designers, who in turn feed their results to a single database analyst (DBA, see Figure 2). It is clear that the DBA will not be able to keep up with the work (and rework). Prudence insists that the designers get their work to a complete and stable state before passing it to the DBA.

Figure 3 illustrates this idea. The vertical axis indicates how complete and stable each group's work is. Completeness refers to how much they have done, and stability refers to how unlikely they are to make changes. For simplicity, the figure illustrates them as joint: The work becomes more complete and more stable over time, shown in an S-type of curve. For each curve, the solid downward-arrow indicates at what point a dependent activity gets initiated.

If the designers take work from a single requirements gatherer as in Figure 2, they can start work on their assignments when the requirements are only slightly complete and stable and still handle the consequential rework. Figure 3 shows the trigger event (the solid vertical arrow from requirements to design) occurring close to the left, while the requirements are not yet very complete or very stable. This figure also shows information continuing to pass from the requirements gatherer to the designers as the requirements work progresses. Once requirements become complete and stable, it will not take long to finalize work. The designers can complete the extra rework because there are five of them to one requirements gatherer.

The DBA, having no excess capacity, needs to be handed work that is more complete and more stable. The solid rightmost vertical arrow in Figure 3, which shows when the DBA's work gets initiated, starts higher on the designer's completeness and stability scale.

Note that in Figure 3 each designer uses much more time than the DBA. This is appropriate, since there is only one DBA for five designers.

The principle says that rework is an expendable commodity everywhere except at the bottleneck station (the DBA, in the above example). Rework can be expended to improve a design, to investigate multiple designs, or to get a head start on a downstream activity. Applying this principle to different circumstances produces different optimal project strategies [2].

Although this is the most complicated principle presented so far, I find that most project leaders have, in fact, used this principle in responding to standard project pressures through common sense and intuition.

10. Sweet Spots Speed Development

The ideal project uses dedicated, experienced people who sit within earshot of each other; use automated regression tests; have easy access to the users; and deliver running, tested systems to those users every month or two. Such a project is clearly in a better position to complete successfully than one missing those characteristics. The surprise is that sponsoring executives do not pay more attention to these important success factors.

When the team cannot hit one of those sweet spots, then they need to invent a way to get closer to it. The farther away they are, the more difficult the project becomes. Here are six sweet spots:

1. **Dedicated Developers**. There is a large emotional and mental cost to a person having to switch between multiple assignments [2, 5]. In my project reviews, I find that once people get interrupted at the rate of about three times per day, they stop even trying to focus on their main assignment and simply wait for the next interruption to happen. One senior project manager reported that he simply does not count as productive staff anyone assigned less than half-time to the project.
2. **Experienced Developers**. Experienced developers know the domain, they know the technologies or how to adopt them, and they know their computer science material. They move at multiple times the speed of their slower colleagues.
3. **Small Collocated Team** (a consequence of principles 6 and 7). Two to eight people sitting in the same room can ask each other questions without raising their voices. They are aware of when others are available to answer questions. They overhear relevant conversations without pausing in their work. They keep the design ideas and project plans on the board in ready sight and share information faster. The developers I have interviewed uniformly say that while the environment can get noisy, they have never been on a more effective project than when a small team sat in the same room.

   Technology can mitigate the situation somewhat. One project team installed cameras on every workstation to display the image of the other people on the project in their various offices [6]. This gave them a sense of each other's presence, and indicated when people were not at their workstations or not to be disturbed with a question. They used online chat boxes to fire off and get answers to the many small questions that constantly arise. They were creative in mimicking the sweet spot in an otherwise unsweet situation.
4. **Automated Regression Tests**. With

automated regression unit and acceptance tests, the developers can revise the code base and retest the entire system at the push of a button. Teams who have such tests report that they freely replace and improve awkward modules. They also report relaxing more on the weekends since they will run the tests on Monday morning and discover if someone has changed their system out from under them. These tests improve both the design quality and the programmers' quality of life.

Experienced developers spend quite some effort to minimize the amount of the system not amenable to automated regression tests.

5. **Easy Access to Users**. Having a *customer* or usage expert available at all times means that the feedback cycle from nominated solution to evaluated idea is much shorter, often in the range of minutes to a few hours. The development team gains a deeper understanding of the users' needs and habits and makes fewer mistakes nominating ideas. It also means that more ideas can be tried, allowing for a better final product.

   Missing this sweet spot lowers the likelihood of making a really useable product. Teams unable to have a usage expert available at all times have substituted weekly sessions with the users, studying the user community in depth before and during the project, using surveys, or using friendly alpha-test groups.
6. **Short Increments and Frequent Delivery to Real Users**. There is no substitute for rapid feedback, both on the development process and the product itself. Some colleagues say that even one month is an intolerably long time. However, there is also a cost to deploying a product, which makes this a MFI proposition (discussed in the previous article).

   With short increments, the process itself gets tested and can be repaired quickly, and the requirements for the product can be tested and varied quickly.

   Projects that cannot deliver to an end user every few months should integrate a full build every few months and pretend as though it were delivered. This way, they exercise every part of the development process.

Differences Between Approaches

At this point, we have listed the issues that bear on how cost- and plan-driven projects can borrow from agile approaches. Some cause intrinsically different responses; other responses are more a matter of habit.

Intrinsic Differences

Statistics vs. heuristics. Some project leaders believe software development is a statistically controllable process; others do not. Their resulting strategies are incompatible. This is one of the places where friction arises between agile and plan-driven project leaders.

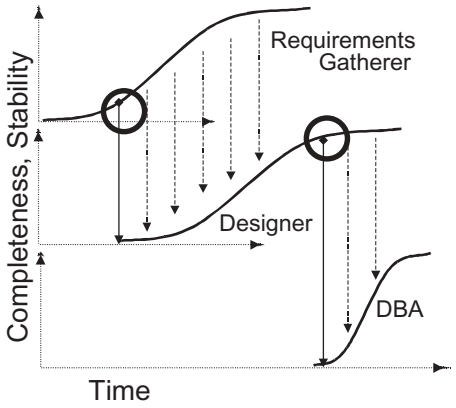
Individuals and interactions vs. processes and tools. Some leaders believe that with the right process, they can become immune to the turnover of key people. Others believe that no process can offer that immunity; the heart of good software development will always reside in the individual people on the project. As with the statistical approach, we are more likely to find process-centric leaders running plan-driven projects, and individual-centric leaders running agile projects.

Responding to change vs. following a plan. It is a fundamental difference between the two project types whether the team is encouraged to or penalized for responding to changes. Even though business needs, requirements, technologies, and people are constantly moving these days, some projects are still fixed in some combination of time, scope, and cost, and must operate in the plan-driven range.

Project plan as MFI or MFF. If the requirements or technologies are likely to change late in the game or without notice, or the team does not have experience with the technology, then it is a poor strategy to treat the project plan as a MFI issue. In those situations, the agile leader's mindset that the plan is a MFF proposition works better. The leader allocates energy to replanning coarsely but frequently.

Design as MFI or MFF. A plan-driven project team, believing that the design can be worked out in advance (MFI), expends resources early to gain that information and lock down the design. For those design elements that cannot be foreseen (MFF issues), plan-oriented design teams often design the system so that a range of future design constraints can be easily incorporated – expending extra design energy early in anticipation of having a more adaptable design.

Many agile designers find those resulting designs overly complicated. Agreeing that certain issues are MFF issues, they argue that a better MFF strategy is to make a simpler design in the first place, with less built-in flexibility. The saved money can then be allocated to change the design on an as-needed basis.



Note: The designer-programmer benefits the schedule by starting earlier and accepting more rework [2].

Figure 3: *Completeness and Stability Over Time*

Some agile designers argue that the MFI component of the design activity is negligibly small, thus little or no effort should be expended on anticipated design changes.

Serial vs. concurrent development. There is a fundamental difference in the strategies applied when agility is a priority compared with when cost is the priority. As Principle No. 8 describes, cost-sensitive projects do better with serial development when that can be successfully executed. Unfortunately, there are so many surprises in projects that it is very difficult to execute successfully.

Surmountable Differences

Working software vs. comprehensive documentation. One tends to find more initiatives for comprehensive documentation on statistics-, process- and plan-driven projects, but this is not intrinsic. Many experienced managers use prototypes, simulators, and incremental development to reduce risk and gain early information on both agile and plan-driven projects, feeding that information into the plan as quickly as possible.

Customer collaboration vs. contract negotiation. Plan-driven project leaders clearly can improve their situation by increasing the collaboration in their customer relations, even if they must write and enforce contracts. This is a case in which plan-driven project leaders can employ some of the same work practices as agile project leaders.

Project plan and design on cost-sensitive projects. A detailed plan does not, by itself, confer cost savings or safety to a project. Detailed plans and detailed designs enable an estimable base-line cost. The manager can then tell if the cost is going up or down over time. It is not the

detail of the plan, but successful application of MFF and MFI decisions that makes the difference in the result.

Borrowing From Agile

Drawing from the above, we see that the plan-driven project can streamline its development operations, improve predictability, and hedge its bets by borrowing in various ways from the agile approach. Following are examples of these.

Streamlining

A plan-driven project leader should still try to hit the six sweet spots: dedicated, experienced and collocated staff; using automated regression test suites; having easy access to knowledgeable users; and showing and delivering incrementally growing, running, tested systems to them regularly.

In addition to these, the project members can question to what extent they can lower the documentation burden through a more informal information exchange.

Improving Predictability

Good project leaders already use prototypes, simulators, and incremental development to get early information on their project. However, in my experience, many leaders of plan-driven projects do not avail themselves of these techniques, which are standard business among agile developers.

Of the above techniques, the most important for the plan-driven team to adopt is incremental development. By delivering a few increments, the leader gains invaluable information about *this* team, *this* problem, and *this* technology. That data are more appropriate to the project plan than estimates from other people working on other problems in other technology.

Hedging Bets

Surprises can show up even on a plan-driven project. Based on where they estimate those surprises are, the plan-driven project leaders can incorporate some of the agile mindset into their strategy. Once again, the use of incremental development is key. The delivery, or even just integration, of each increment offers the team a chance to deal with whatever surprise showed up, whether in the requirements, the technology, or the process. The other technique to borrow is concurrent development, which offers a way to speed development and respond to late-breaking changes.

Lowering Costs

Customer collaboration over contract negotiation. The most important agile value for the cost-sensitive project leader to adopt is customer collaboration. When told that varying a requirement converts an expensive design into a simple, inexpensive one, a customer often is willing to change the requirements to allow the less expensive design. To the extent that the customer and the development team are on good terms, this happens more often.

Working software over comprehensive documentation. Tacit knowledge and informal communication are much less expensive than complete documentation. The cost-sensitive project will play a game of documentation brinkmanship, creating only minimal documents needed to keep the project from falling apart.

Responding to change vs. following a plan. Optimizing from an accurate plan is clearly a winning strategy. The only time that responding to change is advantageous to a cost-sensitive project team is when they discover a shortcut later in the project. At that point, they obviously benefit from changing the plan.

Summing Up

Agile teams put more emphasis on the ideas presented in this two-part article than do plan-driven teams. Most of the ideas are not particularly new. What is surprising is the extent to which these known, old practices are ignored. It is sobering to re-read the paper, "Disciplines Delivering Success," presented at the 1997 Software Technology Conference in Salt Lake City [7] in which Brown points out the following: "*project-saving disciplines ignored by management*: good personnel practices, planning and tracking using activity networks and earned value, incremental release build plan, formal configuration management, test planning and project stability, and metrics."

Of all the practices, the agile strategy of using concurrent development is intrinsically in opposition to cost-minimization under predictable circumstances. However, cost-sensitive project teams can benefit from all four of the agile values and all six of the project sweet spots. Customer collaboration and making good use of close, informal communications are key among those.

Of the differences between development styles, agile developers typically believe that software development is not amenable to statistical process control, and so heuristic project controls must be used.◆

References

1. Cockburn, A. "Learning From Agile Software Development – Part One." CrossTalk Oct. 2002: 10-14.
2. Cockburn, A. Agile Software Development. Boston: Addison-Wesley, 2001.
3. Goldratt, E. The Goal. Great Barrington: North River Press, 1992.
4. Goldratt, E. Theory of Constraints. Great Barrington: North River Press, 1990.
5. DeMarco, T., and T. Lister. Peopleware: Productive Projects and Teams. 2nd Ed. New York: Dorset House, 1999.
6. Herring, R., and M. Rees. Internet-Based Collaborative Software Development Using Microsoft Tools. Proceedings of the 5th World Multiconference on Systemics, Cybernetics and Informatics. Orlando, Florida. July 2001: 22-25 <http://erwin.dstc.edu.au/Herring/SoftwareEngineering0verInternetSCI2001.pdf>.
7. Brown, N. "Disciplines Delivering Success." Software Technology Conference, 1997 <http://stc-online.org/cd-rom/1997/track1.pdf>.

About the Author

Alistair Cockburn, an internationally recognized expert in object technology, methodology, and project management, is a consulting fellow at Humans and Technology with more than 20 years experience. He is one of the original authors of the Agile Software Development Manifesto and founders of the AgileAlliance, and is program director for the Agile Development Conference held in Salt Lake City.

1814 Fort Douglas Circle
Salt Lake City, UT 84103
Phone: (801) 582-3162
Fax: (775) 416-6457
E-mail: alistair.cockburn@acm.org

Call for Conference Participation

The Agile Development Conference is seeking people to give tutorials, host workshops, or submit field reports or research papers at the conference June 25-28, 2003 in Salt Lake City. More information is available at <www.agiledevelopmentconference.com>.