Agile Software Development Joins the “Would-Be” Crowd

by Alistair Cockburn

What is the opposite of a Bengal tiger — a Siberian tiger, an elephant, a gnat, or a puddle of water? None of them, of course. The phrase “non Bengal tiger” is uninformative, just as is non-elephant, non-gnat, or non-water. Being a non-something doesn’t depict anything.

Similarly, there is no opposite to agile software development. The word “agile” depicts where people choose to center their attention. Alternatives to software development arise as soon as they center their attention elsewhere. They might focus instead on rigorous, predictable, repeatable, defect-free, traceable, or even fun software development. The important thing is to identify the center of their attention, not to create a negation of a label.

We’re still not there. There is really only would-be-agile development. Or, for that matter, would-be-rigorous, would-be-predictable, would-be-repeatable, would-be-traceable, or would-be-fun development.

Would-be indicates that the people involved in the project aspire to a certain center. Before the project they can say that they intend to work from that center, but it is only after the project that they can say that their way of working actually was agile, repeatable, predictable, defect-free, or fun.

Let’s take a fresh look at would-be-agile and would-be-something other than agile) software development.

WOULD-BE-AGILE SOFTWARE DEVELOPMENT

“Agile” refers to maneuverability, the ability to respond to changes in the environment. Would-be-agile software development means that the team decides to focus on being able to incorporate ongoing requirements changes without great trauma.

Core Elements of Would-Be-Agile

The standard mechanism for reducing the trauma of ongoing requirements changes is to break the project into subprojects, each ending with delivery of running, useful sections of the system. This early and regular delivery provides the team with feedback about both the development process and the system being developed.

Early and regular delivery helps build safety into the project. By delivering several useful releases within a short time span, the sponsors, customers, and developers gain confidence in their way of working. Pausing and reflecting after each subproject, they have opportunities to fix mistakes in their way of working and try out new ideas for becoming more effective.

Early and regular delivery also allows the people to get feedback about the system in operation and, particularly, which of their initial thoughts were mistaken. The deliveries also provide points at which they can uncover and react to new requirements, whether those come from users’ reactions or from changes in the business environment.

A common recommendation among agile methodologies, therefore, is to run small (3- to 12-week), time-boxed subprojects and to reprioritize the requirements after each time-box. This gives the projects half of the agility they need. The other half of their agility comes from replacing some of their written documents with enhanced informal communication among team members, shifting the group’s organizational memory from external to tacit knowledge.

External knowledge is the kind we can look at in paper or online documents. Examples are the project plan, the requirements
document, interface definition documents, design documents, meeting minutes, design review results, test plans, test scripts, defect reports, and many others. Tacit knowledge is the kind people retain in their persons. It includes not only what each person knows about the project plan, requirements, design, and so on, but also whom they know to talk to, when various people are available, the social and technical conventions in place.

All project teams rely on tacit knowledge, usually to a far greater extent than they suspect. Agile methodologies, though, place a deliberate reliance on tacit knowledge. Through the use of informal communication channels, such as colocated teams, pair programming, or daily stand-up meetings, they demand that the people on the team keep each other abreast of ongoing changes to the plan, the requirements, the design, the code.

When it works, it makes agile teams far more maneuverable. A short discussion at the stand-up meeting can suffice to indicate a change in corporate policy or system goal. A discussion between a few developers informs them of changed requirements or a change in the design. Not having to update the external knowledge base means they can make many changes in a short time period.

When it works.

To make it work, the team members build not only on sitting close together, but also on maintaining open communication among themselves. The tacit knowledge base doesn’t track very well if people are being secretive with each other.

In other words, group amicability, morale, and community become first-order concerns for the group. To the extent these are in place, the informal communications channels may suffice. To the extent they break down, the communications suffer, and so does the team’s maneuverability.

This, then, is the biggest difference between agile and other sorts of development processes. Agile processes put the topics of people, community, amicability, and morale front and center. Standard process descriptions don’t have a place for discussing these topics.

It is this line of thinking that brought developer Andrea Branca to write, “Other processes may look agile, but they won’t feel agile.”

Variations in Agility
As we have seen, four characteristics of agile processes are:

- Using short subprojects
- Reflecting on their practices
- Working in close location
- Attending to community

With these four characteristics, we can look at ways to vary, strengthen, or weaken the agility of the team, depending on where the organization chooses to center its attention. (Yes, Virginia, the organization might actually choose not to focus its attention exclusively on the agility theme.)

Short Subprojects
Ward Cunningham tells of using ultra-short subproject cycles while writing code for Wall Street companies. Given the fickleness of the stock market, there was no telling on one Monday what the critical business need might be the next Monday. Therefore, the team ran each project to be just one week long.

On Monday, team members would meet with their sponsors and select the top issues for the week. The programmers bid what they could accomplish in the week (often not a complete system, of course, but enough to get some initiative to a functioning stopping place). At the end of the week, they completed their initiative, delivered it, and paused. The following Monday, they would find out whether the market still craved that initiative as they repeated the exercise. If the pressing demand was for some other piece of software, they worked on that instead. Since a piece of software might not get touched again after Friday, it was critical that each subproject be sized to complete and reach a stable, final state on Friday.
That was an extreme situation, of course. Still, the longest I have ever seen subprojects work well is four months. It seems that if they are longer than that, the people aren’t able to focus on their work properly. Even in four months, a lot can change about the business, and therefore longer subproject periods make the group less agile.

Setting up and tearing down subprojects is not free. There is planning and setup work to be done at the beginning, and integration, test, and delivery (and possibly training) to be done at the end. Therefore, choose a duration of time that balances the need to respond quickly with the overhead cost of setting up subprojects.

SCRUM recommends one-month subproject lengths. Extreme Programming recommends three weeks. Crystal Orange used three months. You can see that the exact duration varies by circumstance and preference, but 1-16 weeks forms the outer boundary.

Reflecting on Practices
The Crystal methodology family is the most adamant about having the members of the team get together and discuss what they are doing well and what they should change. Jim Highsmith (Adaptive Software Development) recommends a product feedback meeting, so the product requirements can be refined on an ongoing basis. Both of these meetings happen at the end of every subproject. Although reflection workshops were not cast into the 12 key XP practices, in point of practice most XP groups do insist on holding such workshops at the end of each iteration.

SCRUM and XP call for short, daily stand-up meetings, in which fast-breaking planning, requirements, design, and also team communication issues can be brought to light. (They are held standing up in order to keep them short!)

Such reflection is core to having the group’s process track its needs. To vary, weaken, or strengthen this characteristic, the team can make the reflection workshops longer or shorter, more or less frequent.

An eight-person German project team using Crystal Clear with three-month subprojects chose to hold reflection workshops after each quarter’s delivery. To have extended peace and quiet for their workshop, members of the team went to a village well outside of town for two days. They spent the first day team building and reflecting on the previous quarter’s work, and the second day jointly creating a plan for the next quarter’s work. During the second year of the project, they shortened the workshop to a single day.

In contrast, on my first Crystal Orange project, which also used three-month deliveries but had 45 people (24 programmers), we spent one to two hours in the cafeteria after each delivery. (This is perhaps a commentary both on German workers versus American workers and the insurance industry versus the retail industry.) We only gathered a subset of the team — just the analysts, say, or just the team leads and selected programmers. However, we did it twice per subproject, so that we could change our process in the middle of a subproject.

XP iterations are so short (three weeks), that once after each iteration for an hour may be sufficient time for reflection. A dot-com company using Crystal Orange Web calls for delivery every second Thursday and holds a one-hour reflection workshop with all 50 employees the (Friday) morning after.

Working in Close Location
Crystal Clear and XP simply require that people work in the same or adjacent rooms. This obviously only works for teams up to 8 or (with a stretch) 14 people. None of the other agile methodologies is so adamant on this point.

All agile methodologies are, however, sensitive to colocation, since they rely so heavily on rich, fast communication channels. Using those communications channels allows them to reduce the external documents they have to construct and maintain.

To vary the agility, vary the quality of those communication channels and vary the balance between external and tacit knowledge. It is not the case that to be agile, the
team should produce no paperwork (external knowledge). External knowledge storage has various advantages, the least being any contractual obligation for the team. Information on whiteboards and in documents has a sort of “stickiness” to it that allows people to refer back to what was decided.

The would-be-agile team, therefore, needs to adjust the amount of knowledge that should reside in conversation and personal memory versus sticky or archival form. It is extremely rare that the answer to the communication question might be “all oral.” In the other direction, due to the very real requirement for tacit knowledge and the overall weakness of our current forms of documentation, I have never yet seen a situation where the answer might be “all external.” This means that the organization and the team must decide on how much of each is appropriate. Usually, a little external knowledge goes a long way, and so less is needed than is usually believed.

For communication channels, some people are experimenting with having people sit in different areas of the same building, connected with Webcams and chat software over a high-speed intranet. While this is obviously not quite as rich as sitting in the same room, they claim fair success with it [1]. Others have experimented with pair programming over high-speed Internet lines, using NetMeeting and a telephone between them. They report that while it is not as good as being in the same room, it is much better than programming alone.

You can see that running a project across countries and time zones creates a real obstacle for the would-be-agile team. Fiddling with the issue of spontaneous, rich communication is a key part of deciding how agile to be.

Running a project across countries and time zones creates a real obstacle for the would-be-agile team.

Attending to Community
The daily stand-up meetings and reflection workshops give people a chance to voice their concerns or suggestions regarding the group’s amicability. I recall visiting one XP project on a Tuesday morning and listening while a programmer described a concern and a wish. He said that when he came in on Monday, he found the code vastly changed from Friday. Could people please let others know if they planned to work on the weekend, so everyone would be alert to changes on Monday and would know whom to talk to? This is the sort of attention to community that slips through the cracks in standard process descriptions.

One South African executive likes to take new teams off-site for a two-day team-building course. He’s not sure to what extent it really builds teams, but it obviously helps. At the very least, he says, the people get to meet each other socially and come to understand that the company considers team quality an issue important enough to spend money on.

Of course, the best team building comes from succeeding, which gets us back to the theme of short subprojects. Along the way, though, the members of the team have to find ways to express their fears and wishes for the community.

Varying this characteristic of the agile process is done by the project sponsor, project manager, or someone on the project with a dominant personality, who can either initiate or kill such discussions. The amount of community present, or even the amount of attention paid to community, is so dependent on the personalities of the people involved that it often is more accidental than orchestrated.

WOULD-BE-(SOMETHING OTHER THAN AGILE) DEVELOPMENT
Not all project teams aspire to being agile. If they do not, they must announce where they are actually centering their attention. They might strive to achieve predictable, repeatable, cost-optimized, rigorous, traceable, defect-free, fun, or laid-back development. Let’s look at some of these in turn.

Predictable Development
In would-be-predictable development, the team focuses on hitting a cost, time, or defect window. An example would be a fixed-time,
fixed-cost project. People bidding on such projects may abandon the advantages of the agile mechanisms in exchange for predictability.

Unfortunately, the world is not kind with respect to predictability these days. Unless the organization has already done the same kind of project with the same people in the same technology, the people will be hit by unpleasant surprises somewhere along the way.

Therefore, there is an advantage to mixing some would-be-agile in with would-be-predictable development. That way, when the inevitable surprises do hit, a team that has been attending to short subprojects, reflection, communication, and community can recover more quickly and with less cost. That is a contribution of an agile process to the other would-be processes.

Repeattable Development
Would-be-repeatable development is usually done with the intent of improving predictability, and usually with an added intent of shrinking the time, cost, and/or defect window.

Having the goal of improving predictability presupposes two things: first, that the project assignments, technology, and project teams are similar enough across projects for measurements to transfer, and second, that the metrics are measuring the correct things on the project.

To most project teams engaged in a series of projects where the aim is repeatability, it does not matter if either of those presuppositions is not met. Their asserted goal is to achieve repeatability, and so they must evolve metrics that will transfer across people, technology, and assignments.

Personally, I don’t think there are many project series that fit the presuppositions, and I don’t think the metrics are yet measuring the correct things. (For example, I have not yet seen such projects measure and report communication and amicability within the team.) One very senior developer hypothesized for me that where repeatability is being achieved, it is being achieved by making the process so slow and heavy that the time spent inventing and reinventing ideas is negligible in the cost of the entire project.

Far from being a glib attack on would-be-repeatable projects, this idea may be correct: to generate repeatability within a project series, create an overall process large and costly enough to dwarf the inevitable mistakes the developers will make. Then, the risk is greatest on the first project and diminishes after that, as long as the projects and staff are similar enough.

Would-be-agile development can still inform would-be-repeatable development. Rarely is any project as similar to a previous project as a subproject is to a previous subproject on the same overall project. The assignment is very much the same, the technology is the same, the people are the same. Repeatability improves over the subprojects, which help for the next overall project. The ideas of agile development also predict new project metrics to be tried: communication distance, frequency, and richness, as well as measures of community and amicability.

Cost-Optimized Development
Would-be-cost-optimized development calls for strategies almost opposite to would-be-agile development. Correctly executed would-be-agile processes cost more than correctly executed would-be-cost-optimized processes. That is, the former use less elapsed time but more work-hours.

The sketch of the argument goes like this: to the extent that the person coordinating the would-be-cost-optimized project can avoid being hit by unpleasant surprises during the project, he or she can arrange for people to show up only at prearranged times for prearranged durations, do their work, and leave. Doing this careful scheduling allows the project coordinator to optimize salary costs. Since some number of mistakes will certainly be made, he or she will arrange for people to show up perhaps two or three times, to reduce the defects to acceptable amounts.
You may recognize in this description several other fields besides software development where this strategy is used. I immediately think of building houses and publishing books. In both cases, there is similarity across projects and great incentive for minimizing costs. In house construction, the goal is to get things right on the first pass; publishing houses use a fixed two- or three-pass schedule.

You should see that this form of serialized development will be lower in cost than having the people around for longer periods of time, starting work early, and doing rework as they learn, their assumptions were incorrect, as is done in would-be-agile development. A bit of further thinking shows that the serialized development also takes longer than the concurrent development, since no team is allowed to start work until the team feeding it is done. In fact, correctly executed serialized development takes the longest time of any correctly executed strategy, at the least cost.

The reason that would-be-cost-optimized projects so rarely succeed in their goal is that surprises pop up at all stages in software development. This invalidates the presupposition for cost-optimizing (recall: “to the extent that the person coordinating the would-be-cost-optimized project can avoid being hit by unpleasant surprises...”) and is the very reason people are responding so favorably to would-be-agile development projects.

There probably are circumstances in which surprises can be minimized and plans made to optimize costs. They are likely to be the same circumstances that permit predictability and repeatability.

**Rigorous Development**

Would-be-rigorous development is a false lead. Few people try to be rigorous for the sake of being rigorous. Rigor is hard. Usually, a group works at being rigorous because they think it will help in some other way, such as predictability, repeatability, defect reduction, or fun. (XP done properly is rigorous. In an odd twist, people practicing XP tell me that the extra rigor makes programming both more defect-free and more fun!)

On occasion, a manager or sponsor will advocate more rigor in development as a catch-all prescription meaning to “get better” at something. Therefore, you should probably choose to be would-be-something-else, not merely would-be-rigorous.

**Laid-Back or Fun Development**

A university organization can’t afford to pay its programmers much. To attract and keep anyone of caliber, it has to offer other attractions. The group I am thinking of allows its programmers to work at any time of day or night that suits them, lets them choose their own projects and technologies within a very liberal set of guidelines, and generally works to keep them from feeling pressured.

**The reason that would-be-cost-optimized projects so rarely succeed in their goal is that surprises pop up at all stages in software development.**

This is would-be-laid-back development at work.

Other groups feel that the only way they can keep their developers motivated and producing excellent software is to make the development environment fun. Programmers in these organizations, who are likely to be working on aggressively state-of-the-art projects, may tease and compete with each other. Rewards include playing competitive computer games like *Doom*, even within standard working hours.

I think it should be obvious that would-be-laid-back and would-be-fun processes are fully compatible with agile development. The latter are even likely to raise morale and communication, although the intense competition in some places may affect amicability to some extent.

**SUMMARY**

The values and strategies of the “Manifesto for Agile Software Development” are only *supposed* to confer agility on the team. Agility shows up in the execution — or it doesn’t. The same is true of the values and strategies of the other sorts of processes. Conversations
about these different approaches would be a lot less heated if everyone was clear on the fact that they are \textit{would-be} processes: would-be-agile, would-be-predictable, would-be-fun, and so on.

Calling them \textit{would-be} this or that makes it clear to all where the project sponsors are centering their attention — responding to late-breaking surprises, hitting a cost window, retaining staff, or whatever they may choose — and thereby diffuses much of the hype currently surrounding would-be-agile development. It allows us to talk about why the sponsors aim for that quality, what strategies might work, what might get in the way, and how to blend the priorities.

Would-be-agile development centers around handling late-breaking surprises. That leads to the strategy of building the project from subprocesses (incremental development), enriching informal communications between people, and emphasizing the tacit rather than the external knowledge base.

Would-be-agile strategies can be blended into would-be-predictable development to reduce the cost of handling the surprises that inevitably crop up. They are antithetical to would-be-cost-optimized development, since they make the opposite tradeoffs between elapsed project time and the work-hours used. Would-be-agile strategies should work well with would-be-fun strategies.

I hope that by considering all of these approaches as \textit{wanting} to produce an effect rather than \textit{promising} to produce an effect, we can make progress in evolving and blending strategies.

**REFERENCE**


\textit{Alistair Cockburn is a Senior Consultant with Cutter Consortium’s Agile Project Management practice and a contributor to the Agile Project Management advisory. He is Consulting Fellow at Humans and Technology, where he helps clients succeed with object-oriented projects, including corporate strategy, project setup, staff mentoring, process development, technical design, and design quality. He is the originator of the Crystal family of methodologies, and he has over 20 years’ experience leading projects for companies in the US and around the world. He is the author of Surviving Object-Oriented Projects and Agile Software Development and has written papers and articles for a variety of trade journals.}

\textit{Mr. Cockburn has developed courses in project survival, methodology development, requirements writing, OO design, and project management strategies. He has been invited to teach and talk at the Software Technology Conference, Object World, Software Development, and OOPSLA. He has taught OO concepts and design to hundreds of executives, managers, and programmers worldwide.}

\textit{Mr. Cockburn can be reached at Humans and Technology, 7691 Dell Rd., Salt Lake City, UT 84121; Tel: +1 801 947 9275; Fax: +1 775 416 6457; E-mail: aлистair.cockburn@acm.org.}