Using Both Incremental and Iterative Development

Dr. Alistair Cockburn, Humans and Technology

Incremental development is distinctly different from iterative development in its purpose and also from its management implications. Teams get into trouble by doing one and not the other, or by trying to manage them the same way. This article illustrates their differences and how to use them together.

Incremental and iterative development predate the Agile movement; I first ran into them while doing research for the IBM Consulting Group in 1991 [1, 2, 3]. At that time I learned how different they are in purpose and nature, and eventually how to manage them.

Those differences seem to have been forgotten in the intervening years. I now see would-be Agile teams suffering from doing only incremental development, where I used to see waterfall-type projects suffering from doing neither or only iterative development.

Both are needed. People need to learn to use them separately as well as together.

Definitions, Please!

Briefly,

- ***Incremental*** development is a staging and scheduling strategy in which various parts of the system are developed at different times or rates and integrated as they are completed.

The alternative strategy to incremental development is to develop the entire system with a *big-bang* integration at the end.

- ***Iterative*** development is a rework scheduling strategy in which time is set aside to revise and improve parts of the system.

The alternative strategy to iterative development is to plan to get everything right the first time.

It is important to notice that neither strategy presupposes, requires, or implies the other. It is possible to do either alone, both, or neither.

In practice, it is advisable to do both in different quantities. If you only increment, there tends to be an unpleasant surprise at the end when the quality is not good enough. If you iterate the entire system, ripple effects of the changes easily get out of control.

It Is Not Waterfall

First of all, we need to get past the *it looks like waterfall* trap.

In all development, whether prototype, Agile, tornado, or waterfall, we first decide what to build; we then design and program something. Only after doing some programming (however much we decide to program), we put ourselves in a position to debug the system. Only after the system is running can we validate that what we built is the right thing to build, built correctly. This sequence is shown in Figure 1.

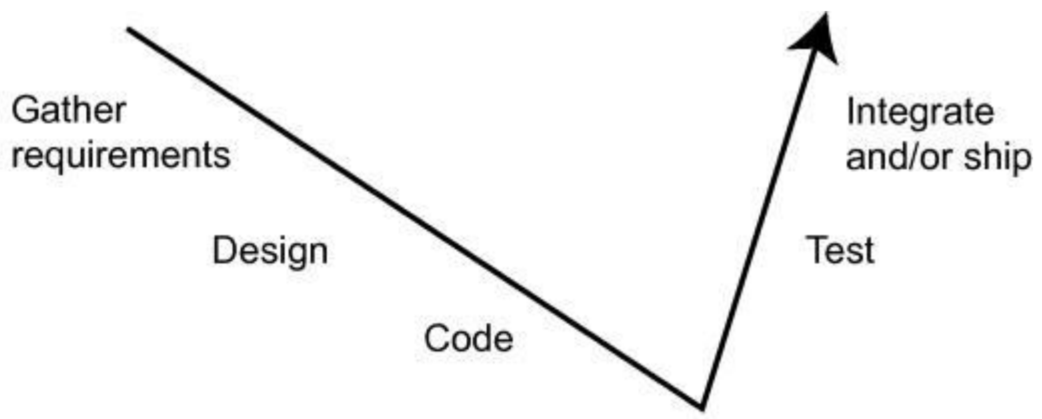

Figure 1: The Validation V Is a Fact of Life

Figure 1 should really be read backwards, as a dependency diagram: We cannot ship till we debug and validate; we cannot debug until we code; we cannot code until we design, we cannot design until we have decided what to design.

In other words, the *validation V* is a simple fact of life, and we shall have to deal with it in both incremental and iterative development.

Incremental Development

In incremental development, we break up the work into smaller pieces and schedule them to be developed over time and integrated as they are completed. Figures 2-4 illustrate this procedure.

Imagine that the top sheet of blocks represents various user interface components, the middle sheet represents middleware, and the bottom sheet represents back end or database components.

Figure 2 shows that in the first increment, a full piece of functionality is built from the user interface (UI) through to back end (and in this case, additional pieces of the UI are also built). In the second increment (Figure 3), we see that additional functionality is added across all layers of the system. This may be a sufficient point to deploy the system as it is so far to real users and start accruing business benefit. In the third increment (Figure 4), the rest of the system is completed and incremented.

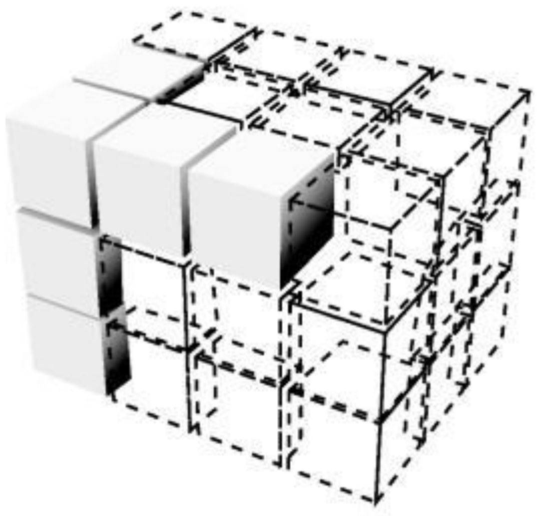

Figure 2: Incremental Development, Stage1

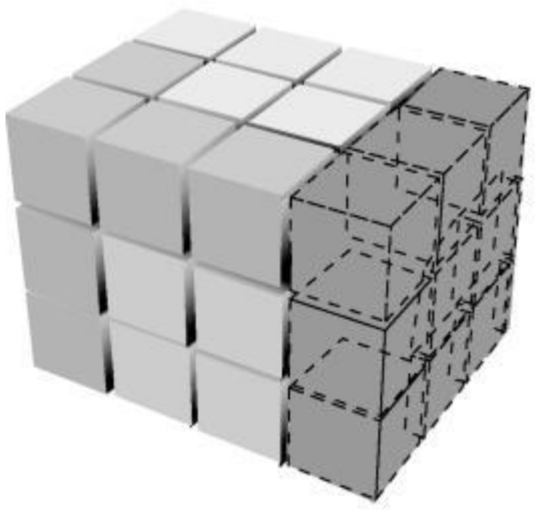

Figure 3: Incremental Development, Stage 2

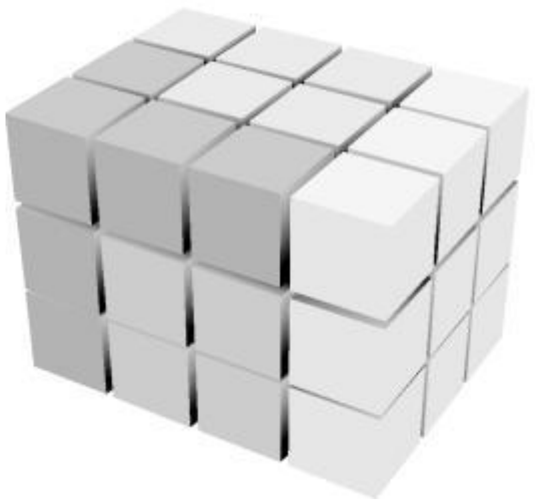
Figure 4: Incremental Development, Stage 3

The pattern just described is the one used by most modern projects, Agile or not. It is a staging strategy with a long history of success.

The mistake people make these days is they forget to iterate. They do not factor in the time to learn what they misunderstood when they decided what to build at the very beginning and what needs to be improved in the design.

This mistake results in the old failure of delivering what people do not want. I wish to highlight that even many Agile project teams make this mistake.

The correcting strategy is iterative development.

Iterative Development

In iterative development, we set aside time to improve what we have.

Requirements and user interfaces are the most notorious places where we historically have had to revise our work, but they are not the only ones. Technology, architecture, and algorithms are also likely to need inspection and revision. Performance underload is often wrongly guessed in the early stages of design, requiring a major architectural revision.

In terms of the validation V, the difference is that instead of integrating and perhaps shipping the software at the end of the cycle, we *examine* it from various standpoints: Was it the right thing to develop? Do the users like the way it works? Does it work fast enough?

Figure 5 shows the validation V for an iterative development cycle.

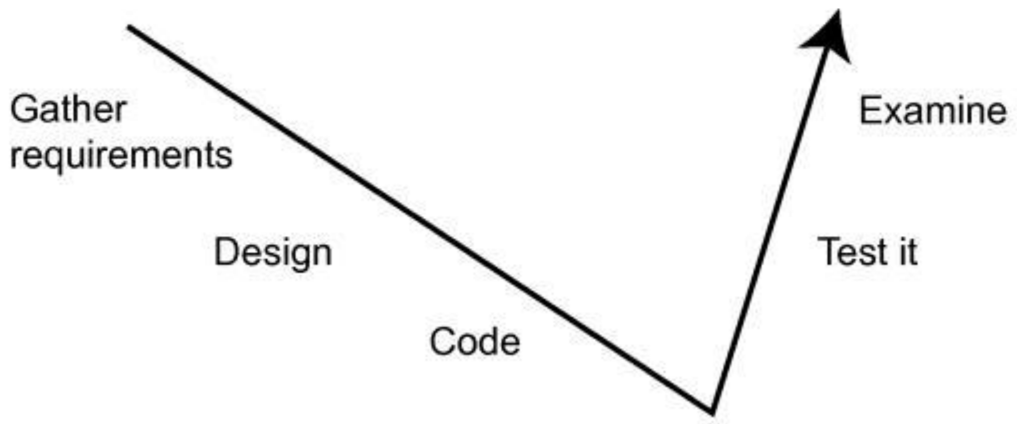

Figure 5: The Validation V for an Iterative Cycle

There are two particular, specialized rework strategies:

- Develop the system as well as possible in the thinking that if it is done sufficiently well, the changes will be relatively minor and can be incorporated quickly.
- Develop the least amount possible before sending out for evaluation, in the thinking that less work will be wasted when the new information arrives.

There are aficionados of both approaches. Indeed, both work well under certain circumstances. A project manager must learn to use both.

The following is an effective use of the first strategy: A musician and a photographer were making a DVD together. The musician recorded the four-minute track in its entirety. The photographer noticed that a small sequence of slide transitions did not match the music. The musician re-recorded just those bars, and spliced them into the music track.

To show an effective use of the second strategy, I adjust Jeff Patton's example of the painting of the Mona Lisa, imagining the discussion between Leonardo and his patron (Figures 6-8) [4].

Leonardo draws a sketch of what he intends to do (Figure 6) and goes to the patron, asking, "How's this going to work for you?"

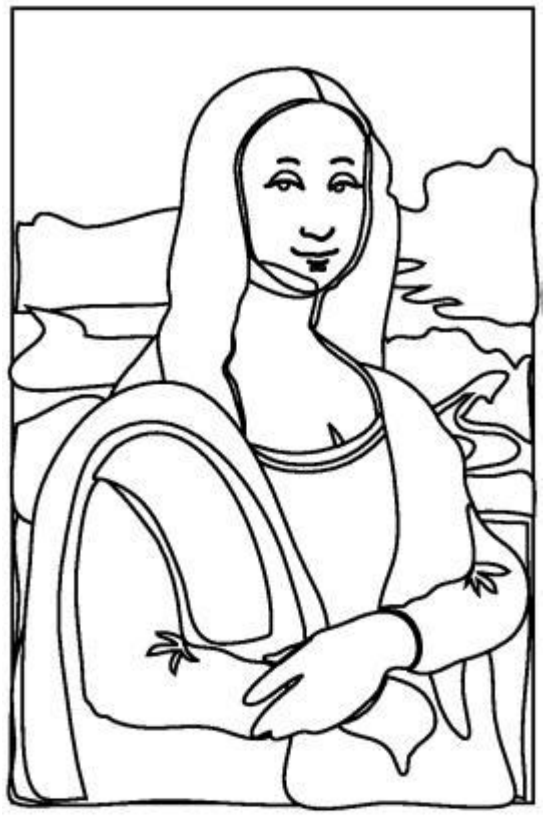
Figure 6: Iterative Development of the Mona Lisa, Stage 1

The patron says, “No, no, no. She can’t be looking right, she has to be looking left!” Fortunately, Leonardo has not done too much work yet, so this is easy to change.

Leonardo goes away, reverses the picture and does some color and detail (Figure 7). He goes back to the patron: “By cost, I’m about one-third done. What do you think now?”

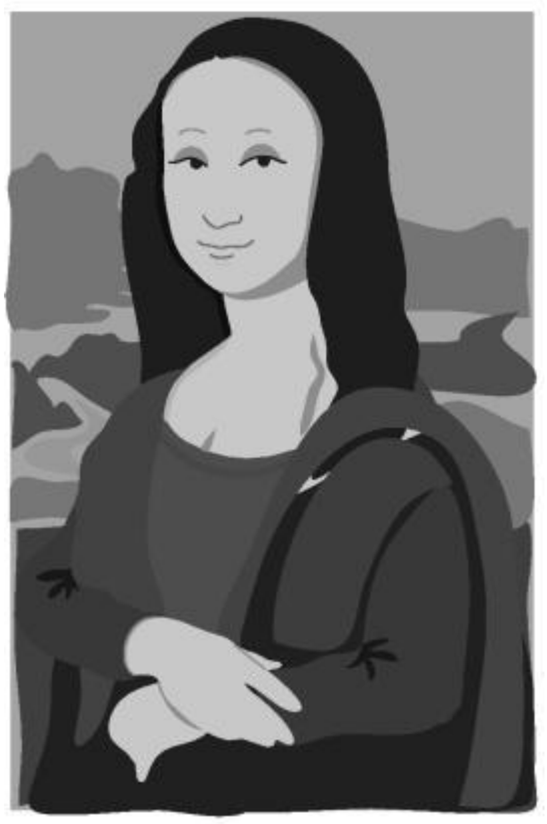

Figure 7: Iterative Development of the Mona Lisa, Stage 2

The patron says, “No, you can’t make her head look that big! Make it look more balanced with her body size.” (Yes, they had the equivalent of Photoshop and airbrushing back then – he was called *Leonardo*).

Leonardo goes away and finishes the painting (Figure 8) and turns in his bill.

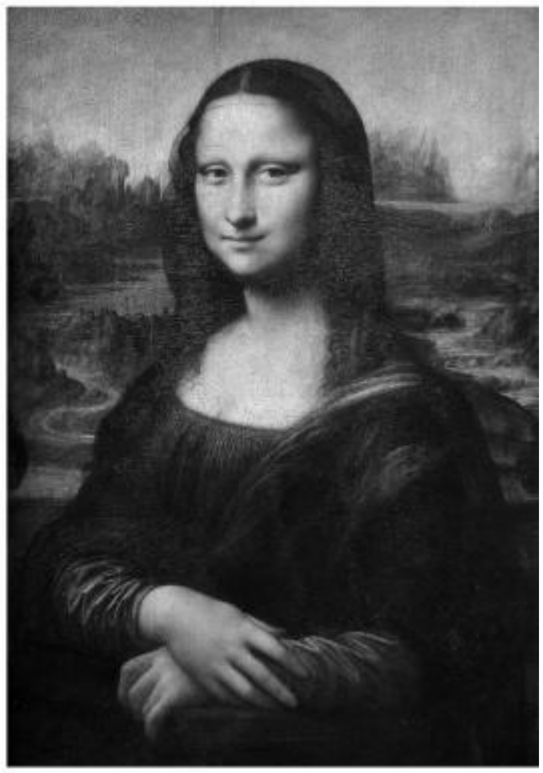

Figure 8: Iterative Development of the Mona Lisa, Stage 3

The patron says, “Really, I’d rather have her eyes bigger, but okay, for the money I’ve paid, let’s call it done.”

What I wish to highlight is that both strategies are valid and both fit the *iterative* tag. In both cases, rework was done on an existing part of the system.

Blending the Two

Incremental and iterative development fit well with each other. Taking advantage of the validation V, we can arrange to alternate *incremental* Vs with *iterative* Vs in various ways to get any number of composite iterative/incremental strategies as Figure 9 illustrates.

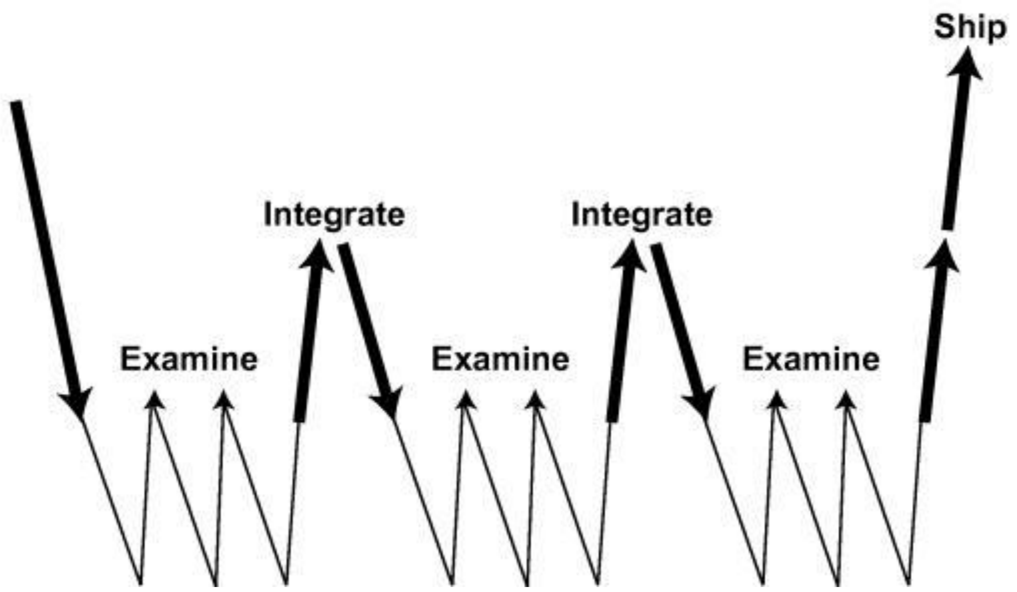

Figure 9: Putting Iterative and Incremental Development Together

Figure 9 shows a strategy in which each incremental section of the system is given two examination/rework periods before being integrated and staged for delivery. The figure shows three incremental development periods, each increment staged as it gets completed, the whole then shipped as a package.

This is one – but only one – of the possible ways to blend the two. As long as they are clearly marked *examine* and *stage* (or even better, *ship*), the Vs can be mixed in almost any combination.

Figure 9 shows one added benefit of describing increments and iterations with Vs: The resulting diagram maps easily to a calendar. Each examine, integrate, or ship marker is a milestone in the project manager's project plan. This allows the project manager to preview and monitor the time spent revising. In this way, we put the validation V *fact of life* to good use in showing our incremental-iterative development strategy.

Managing Them

Superficially, the two look very similar. However, they need to be managed differently. Increments are easy to spot, easy to separate, relatively easy to estimate, and easy to schedule. The entire strategy can be summarized in two steps:

- Divide the system into complete, useful slices of functionality or according to some other useful decomposition you may choose.
- Do them one after the other.

Iterations are considerably more difficult. They are hard to separate, hard to estimate, and hard to schedule. Of course, being difficult does not mean you do not have to do all those things. You still have to do them, including answering the following three questions:

- Which items need to get rework periods scheduled?
- How many rework periods does each need?
- How long should each rework period be?

Although there is no simple, reliable answer to these questions, there is a simple starting strategy from which you can derive a version that fits your project [5].

- Plan to revise the user interface for sure, plan time to add or revise requirements as the end users start to use the system, and suspect that the performance-under-load architecture will need to be revised.
- Allocate two revision periods for the user interface design, and one each for requirements drift and performance re-architecture.
- Allocate for the first revision one-third of the initial development time, and for the second revision one half of that.

You will need to develop and test your own numbers, but those might not be bad starting numbers for the first estimate.

If you are doing Agile development with Scrum or eXtreme Programming, make sure every user story or backlog card passes through the Scrum sprint backlog three times with those multipliers on them as work size estimates for the subsequent passes.

Three Stories

Finally, I offer three stories of incremental/iterative development done poorly and done well. The first one, project *Baker*, shows iterative confused with incremental – they iterated when they should have incremented. The second one, project *Laddie*, shows the problem of modern Agile projects incrementing without iterating. The final one, project *Winifred*, shows them done well.

Project Baker was a fixed price, fixed-scope project with 200 people. They worked in monthlong cycles (an incremental development strategy).

The teams were separated and worked in pipeline fashion so that the requirements writers wrote requirements for some features for a month then passed them along at the start of the next month to the designers. A month later, the designers passed the designs along to the programmers who programmed for a month. At the end, the testers were given pieces of code to test and integrate.

Misunderstanding the term *iterative development*, they then gave everyone instructions that requirements and design could be changed at any time (this was their iterative development strategy).

The pandemonium that ensued is just what you might expect. Each month the requirements writers revised any part of the requirements document, which changed any amount of design and programming. By the third month, it was obvious to the programmers that they were programming a system that had already been changed by the designers who were simultaneously aware that they were designing a system that had already been changed by the requirements writers. The testers never got anything that fit together.

Project Baker was in trouble from the start, partly due to the pipelining strategy but even more to the uncontrolled iteration.

Let us look at an Agile failure mode.

Project Laddie was using an Agile approach with two-week iterations. All user stories were put into a long list and developed to completion each iteration (their incremental development strategy). At the end of each iteration, the customer was shown what had been built during those two weeks. Of course, since two weeks is a very short time, there was never time to show the customer what was being designed so there generally were corrections to be made.

The customer had to choose whether to delay work on new user stories in order to correct the mistakes made, or to push the corrections to the back of the work queue. (This was their iterative development strategy – not a very nice one from the customer's perspective.)

The customer complained after a while that he felt he had to get things right the first time since the choices given to him about how and when to fix mistakes were not very pleasant. This, he correctly felt, violated the very spirit of Agile development [6].

Let us end with a happy story.

Project Winifred was a fixed price, fixed-scope, fixed-time project of 18 months, using about 45 people at its peak. The basic development cycle was three months, resulting in deployment after each cycle [7]. (This was the incremental strategy.)

There was no particular incremental strategy required within each development cycle – the teams got to develop features in any sequence they wanted. However, every team had to show their ongoing work to real users at least twice within each cycle, so that the users could change or correct what was being built (their iterative strategy). It had to be real software, from user interface to database, not just screen mock-ups.

Typically, each team showed the users what they were building after six weeks of work and again after eight weeks of work. In the first user viewing, perhaps 60-80 percent of the functionality was complete. The users were given the right to change anything they did not like about what they saw, including *I know that's what I said I wanted, but now that I see it, it is actually not what I want at all*.

By the second viewing, perhaps 90-95 percent of the functionality was complete, and the users were only allowed to correct egregious mistakes and make fine-tuning corrections. This permitted both requirements and user interface correction while still making sense for a fixed-price contract.

Project Winifred deployed successfully and the users got more-or-less what they wanted. The system is still in use and being maintained a decade later, which is a fair indicator of success.

Note that project Winifred's incremental-iterative strategy combination followed the style of the Mona Lisa story earlier.

Summary

The word *increment* fundamentally means *add onto*.

The word *iterate* fundamentally means *re-do*.

Sadly, *iterative development* has come to mean either incremental or iterative, indiscriminately. That was an unfortunate turn for our industry since each serves a different purpose and needs to be managed differently.

Incremental development gives you opportunities to improve your development process, as well as adjust the requirements to the changing world.

Iterative development helps you improve your product quality. Yes, it is rework, and yes, you probably need to do some rework to make your product shine.

The development process, feature set, and product quality all need constant improvement. Use an *incremental* strategy, with reflection, to improve the first two. Use an *iterative* strategy, with reflection, to improve the third.

References

1. Cockburn, A. "The Impact of Object Orientation on Application Development." *IBM Systems Journal* Nov. 1993.
2. Cockburn, A. "Unraveling Incremental Development." *Alistair Cockburn* <http://alistair.cockburn. us/index.php/Unraveling_incre mental_development>.

3. Cockburn, A. “Using VW Staging to Clarify Spiral Development.” *Alistair Cockburn* <http://alistair.cockburn. us/index.php/Using _VW_staging _to_clarify_spiral_development>.
4. Patton, J. “The Neglected Practice of Iteration.” *StickyMinds* <www.sticky minds.com/sitewide.asp?Function= edetail&ObjectType=COL&ObjectId=13178>.
5. Cockburn, A. “Three Cards for User Rights.” *Alistair Cockburn* <http:// alistair.cockburn.us/index.php/Three_cards_for_user_rights>.
6. Cockburn, A. “Are Iterations Hazardous to Your Project?” *Alistair Cockburn* <http://alistair.cockburn. us/index.php/Are_iterations_ hazardous_to_your_project?>.
7. Cockburn, A. *Surviving Object-Oriented Projects*. Addison-Wesley, 1998.

About the Author

Alistair Cockburn, Ph.D., is an expert on object-oriented (OO) design, software development methodologies, use cases, and project management. He is the author of *Agile Software Development*, *Writing Effective Use Cases*, and *Surviving OO Projects* and was one of the authors of the Agile Development Manifesto. Cockburn defined an early agile methodology for the IBM Consulting Group, served as special advisor to the Central Bank of Norway, and has worked for companies in several countries. Many of his materials are available online at <http://alistair.cockburn.us>.

1814 East Fort Douglas CIR
Salt Lake City, UT 84103
Phone: (801) 582-3162
E-mail: acockburn@aol.com