Lessons Learned Using Agile Methods on Large Defense Contracts

Paul E. McMahon
PEM Systems

Monday, 1 May 2006
Track 2: 3:55 – 4:40 p.m.
Ballroom B

While the agile movement began on small commercial projects, many contractors are employing these methods today (to varying degrees) on large defense contracts. In the process, new challenges are being faced that are not addressed by current published agile literature. Examples of questions being asked include: How do we treat firm requirements? How do we report earned value? How are systems engineering, configuration management, and our test group affected? How should we handle traditional customer deliverables? What can we do about personnel who are not motivated to work on self-directed teams? This article employs scenarios based on actual project situations occurring in 2005 to share the latest lessons learned on what is working and what isn't working when applying agile software development on large government defense projects.

In a May 2005 CROSSTALK article [1], I discussed six agile software development myths and four recommended extensions to apply agile on large distributed projects. Over the past year, I have had the opportunity to work with multiple clients applying agile – or a modified form of agile – on large U.S. defense contracts. In this article, I share what was learned through nine scenarios developed from actual project experiences, along with 22 related lessons learned.

As background for those unfamiliar with agile methods, and to set the context for the scenarios, the agile manifesto [2] provides the following four value statements agreed to by the founders of the most popular agile methods:

- We value individuals and interactions over processes and tools.
- We value working software over documentation.
- We value customer collaboration over contract negotiation.
- We value responding to change over following a plan.

Scenario 1

Agile Planning

An appealing characteristic of agile software development is its potential to help manage change. A client said to me, "We have good requirements for what we know today, but technology changes fast. I need my contractor to be ready to change direction." My client's contractor had won the job based on his proposed agile approach. I was asked by my client to assess that approach.

The contractor was planning incremental deliveries of the refined and allocated requirements along with functional capabilities. I became concerned with the approach based on responses I was getting to one particular question: "What if at the start of the third increment, your customer gives you new priorities and wants to change direction?" The most common response was, "There is no room in our schedule to change direction. We already have too much to do." I then asked, "What if some things were taken off your current list?" The response was not positive, so I asked my client a similar question. He replied, "I never take anything off the list."

Analysis

The first concern is the statement that there is "no room in our schedule to change direction." Adjusting the plan continually is a fundamental characteristic of agile methods. In Scenario 1, work was partitioned into scheduled blocks called increments, but there was no real plan to adjust the effort, degree of detail, or planned tasks during each increment's detailed planning based on new priorities or risks.

The second concern was the statement "I never take anything off the list." Collaboration means cooperation, but it also implies a willingness to honestly consider alternatives. The customer, although he wants his contractor to be ready for change, does not appear to be planning to collaborate.

Agile Planning Insight

There should always be things you can take *off the list*, but this does not mean customers on agile projects must live without all their requirements. I will explain this further later in the article.

Scenario 2

Agile Requirements

I was called in to help a large agile project that was in trouble. The program manager wanted his team to be agile; to help, he initiated a few rules. His first rule for his systems engineers was, "Don't write more than 100 requirements."

When I talked to a developer on the project, he said, "We wanted more details. There was too much ambiguity in the requirements." Another said, "There was a lack of flow-down of requirements from systems engineering."

When I shared the developer's comments with a systems engineer, he said, "We were told to pull back." Then he added, "I don't get it. How are you supposed to handle firm requirements on an agile project? If we don't write detailed requirements with agile, what are systems engineers expected to do?"

Lessons Learned

The first lessons learned address these statements: Don't write more than 100 requirements, and how are you supposed to handle firm requirements on an agile project?

Lesson 1: Write down all your must-do/firm requirements as soon as you know them, and do not plan to *collaborate* on them. For those who claim this recommendation is not agile, I say, this is *practical agility* and the following is why: Trying to *collaborate on truly firm* requirements will only frustrate your team and waste resources. I have witnessed this frustration on multiple occasions during this past year.

When I use the term *collaborate*, I mean an honest consideration of alternatives and a willingness to give. I am not saying do not talk to your customer about the requirements, but I am saying if there is no room to give, then do not pretend there is. Collaboration – in the agile con-

text – means more than talking – it implies taking action that leads to change.

Some have suggested that the term *negotiation* might be more appropriate in this context, but negotiation brings with it an *us* and *them* implication. Alistair Cockburn tells us that "In properly formed agile development, there is no 'us' and 'them,' there is only 'us' [2]."

However, one cautionary note: Are you sure you recognize a must-do requirement when you see one? As an example, one of my clients is modernizing a legacy system. There are lots of firm requirements. The functionality of the legacy system must be maintained, but the users do not need to achieve that functionality the same way. This has been a great point of confusion and contention on the project.

Lesson 2: We often confuse *nice-to-have* requirements with firm *must-do* requirements. Yes, this sounds like basic systems engineering, and I know some of you may be thinking this is not an agile issue. But it is, and lesson No. 3 is why.

Lesson 3: Systems engineering is still required with agile development. I find that in the name of agile, many large projects are forgetting fundamental systems engineering.

Lesson 4: We must get out of the sequential waterfall mentality – this is an outdated way of thinking and it does not work with agile methods.

Lesson 5: Agile does not require fewer written requirements. It does require collaboration to identify the needed detail to implement what the team is focusing on now in this increment. The word *flow-down* implies an ordering – something occurring before something else. Systems engineering does its job before software developers do theirs. Systems engineering does the requirements. The developers *wait* for the *handoff*. This way of thinking will not work with agile methods.

Systems engineering *pullback* is exactly the reverse of what should be happening. On large projects in particular, there are still some very important sequential activities that must happen. For example, systems engineering must do a high-level first pass of requirements and allocate them to major incremental releases *before* the developers get going. This is by no means the end of systems engineering. Today, this point is too often being missed. The critical and most intense part of systems engineering with agile is still ahead *after* the high-level requirements. This is the collaboration on the details that must happen *concurrently,* working closely with the developers in each increment.

The No. 4 and 5 lessons learned address the following statements:

- There is a lack of flow-down of requirements.
- We were told to pull back.
- What are systems engineers expected to do?

Scenario 3

Customer Collaboration and the Program Manager

A systems engineer on a large agile project told me that he had been told to keep quiet at a review concerning a specific technical topic. He said the program manager had told him, "We want to be collaborative."

> *"If you are the program manager on an agile project, expect more conflict early ... Because of this early increased conflict, it is critical to have a strong conflict management process in place and personnel trained in using that process."*

I was concerned when I heard this comment that someone had misunderstood collaboration, so I raised the issue with another team member. He explained to me that the technical topic had been thoroughly discussed and resolved at a previous meeting. The program manager apparently did not want to waste time revisiting it. This made sense to me, but do not dismiss this scenario lightly. Effective implementation of collaboration on agile projects is closely linked to requirements lists, task lists, and collaboration rules. This is explained further in the following paragraphs.

Lessons Learned

Lesson 6: The program manager should not assume the agile team knows how to collaborate. Many will need to be taught how to recognize good collaboration opportunities, and when it is time to stop collaborating. In Scenario 3, the program manager knew the technical topic had been previously discussed and resolved. He also knew that you can collaborate too much, which led to his decision.

If you are the program manager on an agile project, expect more conflict early. This is because of the shorter iterations and risk focus. Because of this early increased conflict, it is critical to have a strong conflict management process in place and personnel trained in using that process [3]. I have observed in the past year both too much and too little collaboration.

One reason people fail to collaborate is because it can be draining. Collaboration takes time and energy. This is one reason why it is important to distinguish truly *firm* requirements from *nice-to-have* requirements. This helps us pick our battles wisely.

Recognize opportunities for effective collaboration. As an example, when a developer says, "We wanted more details," as we saw in Scenario 2, this is a likely opportunity for collaboration. He is saying we need more discussion and action (e.g., updates to task lists) because the current requirements/task *list* is ambiguous, or is missing tasks.

Lesson 7: The program manager's role on an agile project is affected by how he/she interacts with the agile team, particularly with respect to requirements and task lists. Some program managers have asked, "Does agile affect my job?" If you are the program manager, I recommend that you encourage your team to resolve ambiguous requirements and add missing tasks to the appropriate *list*. This will ultimately provide more accurate visibility of the real status back to you.

Program managers should also let their team know they expect to hear about more issues early and that they will not *shoot the messenger*. This may sound trite, but the manner in which a program manager responds to issues raised early can set the tone for the entire project with respect to timely and accurate reporting up the chain. This is particularly important on agile projects due to the increased tendency to push work out as we will see later in Scenario 5.

Lesson 8: On large projects it is necessary to have multiple lists. The organization of most large agile projects includes many *hub-teams* that interact differently from teams in traditional hierarchical organizations. We refer to the teams as hub-teams rather than *sub-teams* because of the manner in which the teams interact [1, 4].

Large projects tend to have more complex products and sub-products. This implies multiple lists. The top list includes the end-customer requirements (e.g., product backlog list for full project/product). Lower lists are more solution (e.g., design) and task oriented and are used by individual hub-teams to remove ambiguity of higher-level requirements and clarify task responsibility.

Lesson 9: When you understand how the full family of lists works together on a large agile project, you will understand why there is always something you can *take off the list*. We remove ambiguity by collaborating and adding solution space items to lower lists (e.g., hub-team lists for specific increments/iterations). By solution space items, I mean tasks associated with design decisions (e.g., the look and feel of a user interface), and other real work that team members must do (e.g., preparation for a customer review and documentation). Because the solution space provides choice, it also provides opportunity to collaborate – to consider and be open to alternatives. This is a full team responsibility and must include systems and software engineering and customer representatives.

You can think of the lower lists as the result of *successful collaboration* as long as those lists represent the real work being done by the project teams. Watch for warning signs of *failed collaboration* such as work that is happening, but is not on any list and has not been agreed to.

Scenario 4

Customer Collaboration and User Conferences

One of my agile project clients has a very large customer community. To gain early feedback, user conferences were held to demonstrate incremental versions of the product. Developers were sent to the conferences to interact directly with the users.

One team member who attended a user conference commented, "We thought the direct interaction between the customers and our developers would lead to fewer requirements, but the users wanted more." Another said, "Many users wanted different things. Our developers did not know who to listen to." Another said, "Some users became upset because they did not see all their requirements in the demonstrated product."

Lessons Learned

I used to say, "You can involve the customer too early even on an agile project." But this does not communicate the situation accurately. I now say, "You can never bring the customer in too soon as long as you know who your customer is."

Ken Schwaber, co-developer of the Scrum process (a popular agile method), uses the term *product owner* rather than *customer*. The product owner is responsible for representing the stakeholders [5]. The product owner manages the team list. In Scrum, the team list is referred to as the *Product and Sprint Backlogs*. The product owner is responsible to keep the list in priority order, and provides clarifications to the team when needed.

Lesson 10: Each hub-team must have its own *single* product owner and its own *single* list for the work that is approved to be working on now. In Scenario 4, the developers did not know who their product owner was. User conferences are encouraged to allow developers to hear the needs of end users directly. However, approval for work by individual hub-teams must be coordinated through a single product owner. Similarly, once work is approved for a hub-team, its priority must be clear and in which increment it is approved to be worked on. There should be a single list for each hub-team for the current approved work. Large projects will have many hub-teams (e.g., a project with 500 people could have 50 hub-teams). This implies 50 product (or sub-product) owners (one for each team). This does not mean each product owner must be dedicated full-time to the product owner role.

"Thinking of a systems engineer as a product owner should not seem like a foreign idea. In many large organizations, systems engineering is viewed as the customer for software engineering."

I have heard some say that agile will not scale up because there are not enough *customer* personnel. The implication is that *customer* must be the *end-customer*. But often on large agile projects, the right customer is not the end-customer, but rather someone who represents the end customer.

Lesson 11: The right systems engineer may be the perfect candidate for a product owner role. This lesson addresses the question, "What are systems engineers expected to do?" Thinking of a systems engineer as a product owner should not seem like a foreign idea. In many large organizations, systems engineering is viewed as the customer for software engineering.

Schwaber, in describing the relationship of the team and the product owner, refers to "constantly collaborating, scheming together about how to get the most value for the business" [5]. This is the model of how systems engineering, software engineering, and support organizations in large companies should be operating for effective agile operations – scheming together (in a positive way) with a common goal of value for the business. Unfortunately today, many large organizations do not operate under this model, but rather with a *throw-it-over-the-wall/not-my-problem* sequential/waterfall mind-set.

Scenario 5

Risks and Priorities

On one project, a hub-team lead engineer said he learned for the first time, in a recent formal program review with the customer, that some work his hub-team was dependent upon was being shifted out to a later increment by another hub-team on the project. That other hub-team had decided they had higher priority and higher risk tasks to work on. No one from that hub-team had coordinated the change with the dependent hub-team, nor did the lead of that team realize the impact of his team's decision.

Lessons Learned

Lesson 12: Hub-teams on larger projects must not decide to move functionality out without collaborating with their product owner. In Scenario 5, the hub-team made a decision to reprioritize their work without coordinating this change with a dependent team. The product owner must approve any changes to hub-team plans.

Lesson 13: Product owners on large agile projects must meet regularly to coordinate hub-team changes with interfacing product owners. On large projects, the product owner has a larger set of responsibilities than on small agile projects. The product owner must coor-

dinate any decisions to change priorities of planned work with all dependent product owners. Individual teams may not be aware of the full project impact of a change to their plans. The coordination process described in this lesson is missing today on many large projects attempting to be agile. Refer to Figure 1 for a diagram of a large agile project team's roles, lists, and interactions.

Lesson 14: A project integration plan is a critical artifact that needs to be employed by product owners on large agile projects. I asked a developer on a large agile project where I could find the project integration plan. He replied, "We do use cases. We do not need an integration plan."

Integration occurs earlier and more frequently on an agile project. Part of the expanded responsibilities of the product owner on large agile projects includes coordination and approval of changes to the work at the hub-team level that may have an impact on interfacing teams. A project integration plan becomes more critical on agile projects due to the increased integration frequency. It is a critical artifact that should be employed by hub-team product owners when discussing potential changes to planned work.

One reason the integration plan is so important on large agile projects is because one can become lost in the details of all the individual team lists on a large project. The integration plan helps the project leaders see the big picture, which is essential when considering plan changes.

Lesson 15: Use cases are not a replacement for the integration planning. Use cases can help developers understand the project requirements. An integration plan conveys the overall project road map, including the planned sequence of activities and dependencies. One cannot replace the other.

Scenario 6

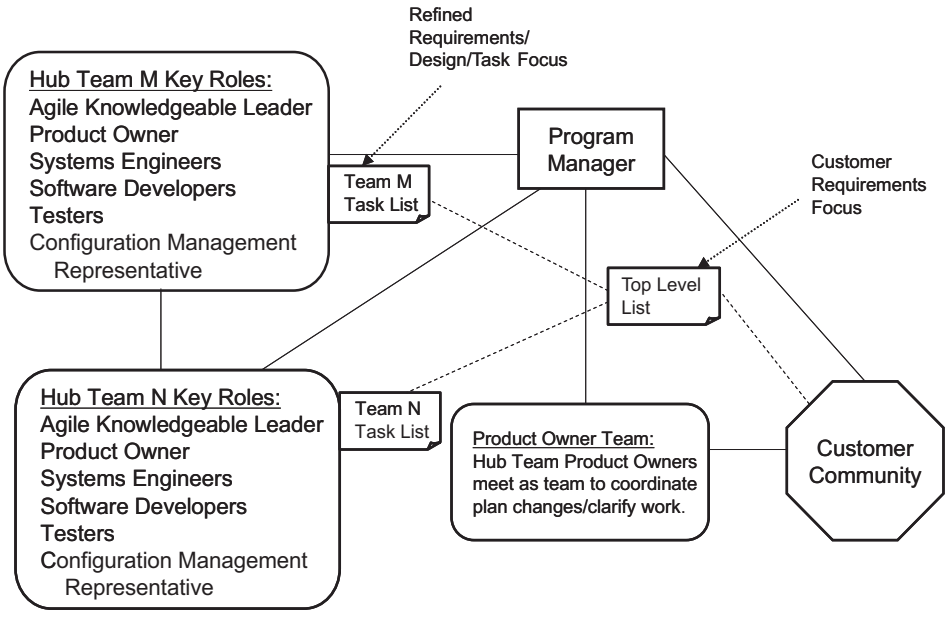
Agile Earned Value

That same hub-team that had shifted planned work out had also reported that it had completed 100 percent of its planned functionality for the same increment. When questioned about the functionality that was being moved out, the lead engineer said that his team had made the decision to move that work out based on priority and risk, so he decided not to include it in his measurement reporting for that increment.

Lessons Learned

Lesson 16: One of the greatest values of agile is early visibility to management of accurate status. This visibility is possible only if progress is reported relative to the baseline plan. In Scenario 6, the hub-team lead engineer made the decision not to include planned work in his measurements. As previously discussed, the hub-team should not make decisions on moving work without coordination with the product owner. But even if work is agreed to be moved out after the start of an increment, it is critical that the earned value report continue to be based on the original baseline plan. Key to agile is the reporting of true team velocity. A common, but costly, mistake on many large incremental projects is pushing planned work out and not raising the visibility. If you push work out, do not hide it. Raise it up through accurate earned value reporting.

Figure 1: *Large Agile Project Team Roles, Lists, and Interactions*



Scenario 7

Self-Directed Teams

I was explaining how self-directed teams operate to a group of senior leaders at one of my client's locations where they were initiating a new agile project. An experienced senior engineer interrupted with the statement, "It will never work on large projects because you will never find enough people with the necessary self-direction skills." My first reaction was that he might be right. I have since changed my view.

Lessons Learned

Lesson 17: Seed your hub-teams with agile-knowledgeable leaders. I used to buy into the idea that agile methods required special skills that many *average* developers could not master. An example is estimating the personal effort required to complete a task, and reporting actual personal progress accurately. Watching agile take hold in organizations has led me to change this belief. Now I believe most team players can pick up agile skills easily.

When a project has leaders who understand agile practices, and mentor others by example, a self-directed culture can take hold quickly. When new developers are exposed to an effective self-directed culture, they learn by watching peers and then *just do it*. I have witnessed this rapid behavior change. It is the leadership and team culture that leads to agile success, not some special set of individual skills.

Scenario 8

Agile Customer Deliverables

Scenarios 8 and 9 are admittedly exaggerations, but they are included to communicate issues commonly faced on large defense projects trying to become more agile.

When I use the term *customer deliverables,* I mean contractually required documentation, reviews, and products (e.g., code).

The program manager on an agile project asks one of his developers, "Can you show me your documentation?" The developer responds, "We're agile, so I am not focusing on my documentation." The program manager replies, "I thought you were writing agile documentation?" The developer replies, "I am, but you wouldn't

want to look at it because it is full of errors."

Lessons Learned

Lesson 18: Agile deliverables must be determined collaboratively with the customer early. Cockburn tells us that when it comes to determining what should be in a document, the answer is whatever the sponsor and the team decide [6].

Too often, I see a lack of discussion on customer deliverables early with the customer on large agile projects. So we should not be surprised when a customer becomes upset when the early deliverables do not meet expectations.

Agile customer deliverables should not be confused with low quality deliverables. As the exaggerated Scenario 8 points out, when we do not plan our deliverables through collaboration with the customer early – and allocate time and tasks based on all the work required – the deliverables will suffer and low quality should not be a surprise.

Lesson 19: The major difference between agile deliverables and traditional milestone deliverables is what takes place before the milestone delivery. When using a traditional waterfall model, it is not uncommon for customers to see deliverables for the first time at a major project milestone. With agile, the milestone should become a non-event because it is the culmination of an on going, close working relationship between customer and contractor. But do not be tempted to delete the *milestone event*. It is necessary on large agile projects to have checkpoints to ensure collaboration is really happening.

Scenario 9

Agile Test and Configuration Management

A manager on an agile project says, "With agile, we get more for less so let's plan on doing less testing." Another manager on the project replies, "Okay, and let's keep the testers and configuration management people in a separate building so they do not slow the agile team down."

Lessons Learned

Lesson 20: Agile does not always mean less. For example, do not plan on less testing. With agile, we do less of certain activities because other activities compensate. For example, we may do less formal written detailed requirements partly because the detailed test cases can compensate [7]. With agile, we test continuously to ensure previous iterations function properly along with new functionality. With agile, it is flawed thinking to believe you can do less testing.

Lesson 21: Testers must be part of the hub-teams. Agile developers must do their own low-level testing due to the tight coupling of the test-code-design cycle. Distinct testers on large projects writing higher level tests must work closely with developers to ensure complete test coverage. In our exaggerated Scenario 9, the testers were placed in a different building partly to keep from slowing the agile team down and partly to provide a level of test independence. On large agile projects, you can still have an independent group run tests, but this does not mean they should be physically separated from the team.

Lesson 22: Configuration management must be integrated into the hub-teams. Cockburn tells us that the configuration management system is steadily cited by teams as their most critical non-compiler tool [6]. This is partly because of the systems support for individual check-in, check-out, and continuous integration. On large agile projects, I have found another reason why configuration management must be integrated into each hub-team.

Schwaber uses the term *shippable* [5] in describing the quality that each iteration's product must have. With agile, we must never demonstrate to the customer a product that has not been fully tested and is *ready to ship*, even if we do not plan on deploying it today.

The reason is visibility – accurate reporting. What we demonstrate must be *done*. If it is not done, we do not report it as done, and we do not demonstrate it. This is an essential practice of agile methods.

Done means ready to ship, which means fully tested, documented, and supportable. If it is not done, do not pretend it is. Configuration management, especially on large agile projects, can provide an important checkpoint to keep the team from caving in on their definition of done when external pressures mount.

Conclusion

Many of the lessons discussed in this article are not new, and some may appear to have little – if anything – to do with agile. Examples include the following: distinguishing *must-do* require-

ments from *nice-to-have*; providing more details for ambiguous requirements; providing single focal points (product owners) for work and work change approval; coordinating schedule changes across the project; developing and using an integration plan; reporting earned value relative to your baseline plan; determining how collaborating teams resolve conflict; planning the work and scheduling the time for customer deliverables; and controlling your baseline releases through your configuration management system.

While it may appear that these are not agile issues, they are very real agile issues. This is because today – in the name of agility – we are witnessing a breakdown of fundamental systems engineering.

Agile is not a short-cut around systems engineering. It is not about systems engineers stepping back and letting developers go. It is about systems engineering stepping forward and working more effectively with all project stakeholders. It is about implementing more effective ways to manage our work *lists* and communicating the results more effectively.

Ultimately, agile is about value achieved through managed change. In particular, make small changes early and often so we do not get surprised by the big ones later.◆

References

1. McMahon, Paul. E. "Extending Agile Methods: A Distributed Project and Organizational Improvement Perspective." CrossTalk May 2005 <www.stsc.hill.af.mil/crosstalk/2005/05/0505mcmahon.html>.
2. Cockburn, Alistair. Agile Software Development. Addison-Wesley, 2002: 215-218.
3. McMahon, Paul. E. Virtual Project Management: Software Solutions for Today and the Future. St. Lucie Press, 2001: Chapter 5.
4. Highsmith, Jim. Agile Project Management. Addison-Wesley, 2004: 239-240.
5. Schwaber, Ken. Agile Project Management with Scrum. Microsoft Press, 2004.
6. Cockburn, Alistair. Crystal Clear: A Human-Powered Methodology for Small Teams. Addison-Wesley, 2004: 178, 37.
7. Ambler, Scott. Agile Modeling. John Wiley & Sons, 2002: 217.

About the Author

Paul E. McMahon, principal of PEM Systems, helps large and small organizations as they move toward increased agility. He has taught software engineering, conducted workshops on engineering process and management, published articles on agile software development, and is author of "Virtual Project Management: Software Solutions for Today and the Future." McMahon is a frequent speaker at industry conferences including the Systems and Software Technology Conference, and is a certified ScrumMaster. He has more than 25 years of engineering and management experience working for companies including Hughes and Lockheed Martin.

PEM Systems
118 Matthews ST
Binghamton, NY 13905
Phone: (607) 798-7740
E-mail: pemcmahon@acm.org