Agile Software Development

PURSUIT FOR AGILE IN GOVERNMENT

In 2001, about a dozen software experts came together in Snowbird, Utah, to change the future. Entirely frustrated with working 80-hour weeks against unrelenting deadlines to produce software that, in the end, was often full of flaws, over budget, and, worst of all, of little value to end users, these developers convened to hash out the principles of a new, more effective and useful way to build large software programs.

By the time they had finished their series of meetings, they had produced the Agile Manifesto, a defining document for agile development, which, simply put, is a collaborative effort among programmers, users, and business experts to work on small modules of a project in short bursts, changing and reassessing priorities based on bugs, feedback, and progress. Unlike the traditional waterfall approach, a sequential design methodology in which each large stage of the project is fully planned upfront and must be completed before flowing downstream to the next phase, the agile methodology offers flexibility in adapting to shifting user preferences and modifying the schedule or program features as development conditions warrant.

Waterfall processes are modeled on industrial systems, highly structured environments intended to eliminate costly backend changes. The Snowbird group was offering a different perspective: software development is knowledge work that requires pliability in dealing with new information, needs, constraints, and discoveries throughout the project—it’s nothing like building a car.

The Manifesto is a revolutionary document that calls for software developers to value “individuals and interactions over processes and tools,” “customer collaboration over contract negotiation,” and “responding to change over following a plan.” It says that the principles of their craft should include satisfying “the customer through early and continuous delivery of valuable software” and “welcoming changing requirements, even late in development.”

These are bold statements, a loud shot across the bow of traditional software development techniques. Some commercial software companies readily embraced the Manifesto’s concepts. However, federal government agencies have been more tentative and less enthusiastic about adopting agile practices, and, in part because of this hesitancy, they have enjoyed only mixed results when they do. Properly implemented, though, these software development principles have enabled organizations to achieve results not possible with traditional approaches. Consequently, these organizations and the contractors that supported them have become passionate advocates for agile in government.

Compelling data support the need for new ideas in designing and developing large programs. A study of nearly 4,000 public and private sector IT projects between 2002 and 2012 that cost more than $10 million found that only 6.4 percent were successful using the criteria of on time, on budget, and meeting user expectations. This study was conducted by the Standish Group, which tracks big software development efforts. In a separate study, Standish found that agile development projects had three times the success rate of waterfall efforts.

Although most IT project failures go unnoticed by the general public, one recent instance can not be overlooked: the rollout of HealthCare.gov. The debut of the
Agile Methods

Agile methods iteratively deliver a finished increment of the system to functional experts for review in weeks by taking a sliver of the overall functionality and working the analysis, design, coding, and testing concurrently rather than building the entire system sequentially. This approach provides a mechanism that encourages feedback so that functional experts are equal players in sculpting the system as it is being built rather than delaying this critical input until the system is finished. The goal of Agile methods is to focus on business value early and trawl for the system requirements throughout the development rather than relying on efforts that attempt to capture all the requirements before development begins.
technology centerpiece of the Affordable Care Act was beset with a range of troubles, from poor site availability, inability to handle excessive loads (and there were millions of Americans who needed access to it simultaneously), slowness, crashes, and incorrect data recording. There was plenty of blame to go around, and everybody involved—from President Obama on down—publicly took some of it. Most lay analysts pointed to the complexity of the system—the laws, politics, public and private sector interactions, and the multiple organizations involved, among other things—as the reason for the failure of Healthcare.gov, a $93 million initial outlay and surely much more before the dust settles.

However, difficulties in navigating intricacy are a poor excuse for a software development collapse; they imply impotency and inevitability, as though the undesirable outcome was preordained by the scope of the program and no one should have expected anything more than what they got. Indeed, the real problem with Healthcare.gov was not its size and many interdependencies but rather the development strategy chosen to design the program. The complexity of Healthcare.gov should have been a beacon not an impediment, a clear indication to the software writers that different techniques were required for this project—and that failure to properly execute those techniques would almost certainly lead to the rollout debacle that ensued.

Ironically, there is evidence that agile development was specified for the development of HealthCare.gov, as has been the case with other government projects that have failed. The problem is—it is one thing to give lip service to agile techniques and another to actually implement them. Indeed, these days, agencies often request that agile concepts be employed in their IT projects; however, without a passionate commitment to strictly follow...
agile guidelines by the business side, the user community, the developers, and the project management team, the effort easily slips back into the sequential design phases (conception, analysis, design, construction, testing, implementation, etc.) common to the waterfall approach.

For instance, an agency might believe that it has an agile project underway because there is a daily standup meeting—an agile staple in which participants discuss, in essentially real time, the progress of the design effort. Attendance may be sparse, key players frequently may be missing, and the agenda loose and unfocused, but the mere existence of this meeting may be used to claim that agile techniques are being employed. It may be agilish but adopting one small piece of this approach is not sufficient. We prefer to call it agifall—doing agile in a waterfall way.

To be sure, agile techniques are not hard to do per se; they are just difficult for most organizations to do. For that reason, this paper isn’t a primer for agile development—that’s already a much-discussed set of procedures. Rather we prefer to offer insight into the significant transformation that a government organization must be willing to undertake before it can hope to initiate a successful agile project. We use that discussion to demonstrate why Booz Allen Hamilton, a leading strategy and technology consulting firm, is so passionate about what an organization can achieve through a proper implementation of agile practices. The substantial benefits that agile techniques deliver in efficiency, cost, end-user satisfaction, ongoing flexibility, maintenance, and support cannot be gained unless an organization has embedded certain valuable and, in most cases, radically different, principles into its project development culture. Without that transformation, agile efforts, real or imagined, will be little more than typical waterfall projects lacking in the benefits we so much want to achieve.

So, if all the right ingredients are present—there is motivation for change and there is the desire to adopt agile techniques to improve software development performance—what does an agile-focused organization look like in practice, and, more important, how can your organization measure and ultimately benefit from the results?

Transformation

If you take only one concept critical to agile implementation away from this paper, it should be the significance of transformation across the organization. Agile is not a software development methodology that lives solely within the narrow realm of the Chief Information Officer (CIO) and his or her IT team. Implementing agile means changing how an organization engages and manages contractors, contracts, key stakeholders, the project management office, and above all, the end-user community for whom the system is being developed. That is a sharp departure from the more siloed development efforts with which most organizations have become comfortable. However, agile rewards organizations that embrace this change with success where there has previously been failure. In short, to make agile work, you must be an agent of change; otherwise, you will always get what you have always gotten.

Figure 2: Functional Innovation

The user community is interested in the sausage taste not in how the sausage is made. To create great software, the software must positively change what that user community is doing today.
But is this even realistic in the highly regulated government world, where federal contracting rules, scheduling requirements, and project management policies are relatively rigid and onerous? It may seem that it is not—but that is a misperception. Indeed, after a lengthy study, the Government Accountability Office (GAO) strongly encouraged the use of agile techniques for federal agencies and concluded that agile’s “more incremental approaches to IT development” have “the potential to improve the way in which the federal government develops and implements IT.” In this report, GAO acknowledged that there are challenges to using agile methodologies in government agencies but that the potential advantages of this approach, especially in end-user acceptance and program health assessment, more than make up for the difficulties.

The key perceived difference between agile and traditional development methodologies that is often raised as an obstacle for government agencies is that agile promotes the discovery of program requirements throughout the development process while traditional approaches prescribe the requirements before development begins. Using the agile approach, how do you write a contract to develop software against unknown requirements, how do you evaluate one contractor’s bid against another’s, and how do you measure progress and performance?

Actually, these distinctions are not as valid as they would first appear. Even with waterfall methodologies, contracts are awarded before the requirements are fully documented. What is known at the beginning of a project is the scope, the concept of operations, and perhaps nonfunctional requirements, such as performance, security, and accessibility. The analysis and design phase is relied on to produce the detailed system specifications. So the challenges of writing contracts against unknown requirements and the evaluation of competing bids exist with both agile and traditional approaches.

However, there is a big difference in how progress and performance are evaluated once the software development commences. Traditional prescriptive approaches support a linear development model in which the project is divided into well-defined phases and well-defined gates are controlled by a set of incremental deliverables. A cut-and-dried assessment of progress and quality at each gate determines whether a certain amount of money can be released. There is no doubt that this is a relatively easy and straightforward way to manage a contract.

The problem is that while this approach may be attractive to individuals managing system development, it is detrimental to people that must eventually use the system. Simply put, it is the wrong way to look at project development because it inherently dismisses the probability that a project will organically evolve and change. Given this undeniable distinction between agile and traditional techniques, organization must be willing to alter two critical contractual tenets to enjoy the benefits of agile development: the detail contained in project schedules and delivery requirements.

Looking first at project schedules, government projects commonly begin with a linear sequential approach, creating a detailed integrated master schedule (IMS). This IMS is linked to the enterprisewide IMS, which provides a blow-by-blow timeline that establishes baseline performance at each step, measures deviations from the baseline, identifies critical paths and resource allocation, and drives impact assessments from missed milestones. It is a remarkably specific document with very clear parameters defining what will happen at each step of the project. Despite this detailed roadmap, why do an alarming number of multimillion projects fail to finish anywhere close to their delivery date and, more important, why are project managers taken by surprise at missed milestones? The answer is obvious: there are too many unknowns at the start of the project.

Building an IT system is less like constructing a road—a project in which applied engineering principles have been developed and refined over decades—and
more like painting the Mona Lisa. The precise path from beginning to end is uncertain. Instead of following a plan, agile emphasizes planning. Rather than earmarking significant time upfront to produce a detailed IMS when almost all of the details are unsettled, agile methodology focuses on creating a product backlog, a list of general project requirements—features, capabilities, interfaces, interconnections, and the like—without specific delivery dates that the agile team can use as a blueprint for incremental development. With the product backlog, the mission’s goals are clearly stated; however, exactly what the project will look like, how it will perform, and how users will interact with it are mutable.

In turn, the product backlog serves as a set of guidelines for the project’s contract. Assumptions are made on how quickly each aspect of the backlog, including nonfunctional requirements such as security and performance, would need to be completed to meet a proposed deadline. However, these assumptions are fungible baselines; they are not cast in stone. As the project unfolds, planned progress can be measured against actual progress to validate or invalidate the initial assumptions. Based on these results, estimates can be refined and the contract timeline realigned to meet these new estimates.

Agile methodology embraces change chiefly because that is the only way to deliver the best product to the user community. And by eliminating the perception that change is anathema to the IMS, the schedule supports the project’s goals, provides room for flexibility, and reflects the current state of the project, rather than dictating an unrealistic and less useful outcome. Reflect on your own projects. How frequently (if ever) does a developer look at the IMS? How often is the IMS updated based on each developer’s input? Not routinely, right? Would you be surprised to know that those actions
occur daily in a truly agile project? In agile, every member of the development team looks at the plan, discusses its progress, and provides input virtually every day. If project managers are the only people paying any attention to the IMS—and then only updating it from time to time—as is the case with traditional methodologies, it should not come as a shock that projects unexpectedly miss their delivery dates. An agile project may take longer than hoped but the schedule slippage is transparent, known by the entire team as soon as it occurs, which enables project managers to take corrective action sooner or adjust expectations.

Establishment of delivery requirements is the second principle that must be reevaluated to be able to adopt agile development techniques. Absent a detailed IMS with clear-cut gates and scope, the business case, development, testing, launch, and reliance solely on incremental development steps, how is it possible for an agile team to gauge the project’s progress against schedule and budget? In its report, the GAO identifies the lack of both traditional project status tracking and waterfall-based milestones as major contributors to the dearth of agile adoption by government agencies. However, there are concrete, alternate controls that can be implemented in an agile project to ensure that the government is getting what it contracted for.

The most important control is in the incremental delivery of working software itself. At these points, the project team as a whole must evaluate the partial product completed during the Sprint (i.e., the individual iterative cycle that produces working deliverables) and avoid the mistake of expecting to review a complete product. Recommended changes, insights, assessment, and discoveries should be encouraged during the incremental Sprint Review. And this feedback should flow back into the product backlog to
Gaining user input throughout the project is a primary reason for this frequent interaction, but another equally important one is to drive innovation. By definition, you cannot prescribe innovation, you can only discover it.

be addressed during the next or future Sprints. In other words, the product is sculpted as it is built, rather than prescribed at the beginning and inspected at the end. These incremental deliveries offer much better and more frequent assessment periods than the waterfall method, allowing the Product Owner to monitor the quality of the project and the rate of progress against the stated goals.

Many agencies are comfortable with the big-picture, gated delivery process in the waterfall development approach because it offers large chunks of the project to analyze. The costs, time, look, and feel of the software are built-in upfront, and even if the project overruns the budget and schedule, there is some security in knowing that the software as a whole can be reevaluated at the end rather than looking at the small, individual components that may need to be fixed. The problem is that the comfort gained from the waterfall model—the relatively structured management framework this approach offers—is illusory, and as we have seen time after time, occurs at the expense of the success of the project itself.

In an agile development effort, to get the most out of each incremental delivery, in both quality and progress against goals, functional users must be continuously, on a daily basis, involved in the development effort—a notable exception to the norm for the last 20 years of government software development. Gaining user input throughout the project is a primary reason for this frequent interaction, but another equally important one is to drive innovation. By definition, you cannot prescribe innovation, you can only discover it. And who else but the functional users, people with decades of experience in the organization and the true experts on which IT capabilities could best enhance a department or agency, are in a better position to tease out innovation through their intense participation in a development effort?

There is an added benefit from functional users’ constant engagement in today’s government IT landscape. Government organizations have and will continue to address the reality of sudden reprioritization and reductions in budgets. Investments in new software acquisition are increasing difficult to manage because of these shifting resources. Government managers must keep the projects focused on delivering value by managing priorities. Inherent in agile’s flexible approach is the Product Owner’s ability to reprioritize the product backlog for maximum organizational value. Because the functional community is more actively engaged, it can make more informed decisions.

One additional aspect of transformational change that accompanies agile development must be addressed because it is such a blind alley for government IT efforts—documentation. An all too common and costly error is to view the completion of documentation as a landmark that by itself is evidence of progress in building the software system. That is a misperception. Documentation is a roadmap of what a system looks like and how a user operates it after it is
completed; it is a reflection of the project but it is not the project itself. By including documentation in assessing the program’s timetable, you might believe that the effort is 50-percent complete at the halfway mark even if only 10 percent of that is software. The project appears on track but is actually well behind schedule. In traditional development approaches, a significant amount of documentation is produced during the upfront analysis and design phases and is incorrectly viewed as a tangible expression of progress in software acquisition.

It is easy to measure the input of effort that produced the documentation and the output in the files that were created. But ignored in this calculation is the efficacy of the software that, for example, may be perfectly documented but absolutely useless because it contains operational flaws and does not satisfy user needs. Nobody has brought up the adequacy of the documentation or the architectural diagrams attached to HealthCare.gov; only the deficiencies of the software mattered. Indeed, an overemphasis on relatively trivial documentation as a guidepost of success in traditional software development is so prevalent that one of the four tenets of the Agile Manifesto is that the software must take precedence over documentation.

Transforming an organization to view the software as predominant and, simultaneously, in flux throughout the project, is virtually impossible when specific gates are envisioned at the beginning of the development effort and the writing of the documentation, designed to chronicle the output at these gates, effectively insulates them from change. Agile development perceives documentation as a secondary artifact because the methodology is based on the idea that software design is a moving target; the program functions, features, interconnections, and interfaces that may be envisioned initially could change significantly over the course of a 6-month, 1-year, or 2-year or more development effort. Documenting the most important aspects of a program in the initial phases neglects the progressive discovery of project features and dependencies that the functional experts discover as the system unfolds. Testing and retesting, large and small alterations, new and modified requirements should always be expected in an ultimately effective development project; the documentation can and should wait until these aspects are settled.

### Engagement

Traditional development methodologies appeal to many organizations because they demand less involvement from the agency or department. Functional experts are asked to disengage from their jobs for defined, planned blocks of time to offer their input on how the software should look or operate and, perhaps from time to time, to evaluate it. However, most of the time, these users and stakeholders are divorced from the project from one gate to the next and able to return to their regular jobs until called on again.

Agile projects do not work that way. They require daily involvement from prospective Product Owners and people who can best define the business goals of the software. For many organizations, earmarking resources of that magnitude to a project is an undesirable and even unacceptable requirement. In fact, the GAO report on agile software said that the drain on staff time and cross-functional collaboration were obstacles to applying agile in the federal government environment. But you can’t make do with less. Agile projects will fail without this commitment, because they rely on constant, evolving discovery, which is impossible without the deep engagement of functional and business experts.

With such a major hurdle to overcome, it is fair to ask the question, why would you attempt agile? What do you gain by making that significant resource commitment? The answer lies in the fundamental premise of the agile methodology, captured well in the first line of the Agile Manifesto:
“We are uncovering better ways of developing software by doing it and helping others do it.”

“Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.”

“We are uncovering better ways of developing software by doing it and helping others do it.”

And in the initial principle of the Manifesto:

“Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.”

These tenets are meaningful because they clearly articulate the primary motivation for adopting agile techniques and the advantages in doing so: better, more exciting software that goes beyond the run of the mill to drive enthusiasm and improved productivity among the users. Indeed, nobody accepts the ordinary anymore, and they shouldn’t. Innovative solutions are becoming the standard. With the introduction of smartphones and tablets, the user community is increasingly accustomed to compelling and highly functional software that addresses specific needs for information, efficiency, and entertainment from the applications they download and install. With a couple of finger taps, anyone can load software for anything from the weather, to birds flying through the air, to insurance claims. In other words, software is less mysterious and not something that only the big IT departments can roll out. Consequently, fewer users will tolerate poorly performing or dysfunctional multimillion-dollar software from IT.

Which brings up the conundrum that agile highlights. The degree of user enthusiasm for a program is directly related to the level of user involvement in developing it. If the program is innovative, attractive, and targeted at user needs, it will be wholeheartedly embraced and used to its fullest extent to the benefit of the organization. So the substantial time commitment from the user community should not be viewed as a barrier, but rather as an absolutely critical aspect of software development.

Throughout an agile development effort, three fundamental mechanisms engage the user community: empowerment, prioritization, and acceptance. These mechanisms are essential to the success of the Sprint, which commonly lasts about a month and produces one segregated portion of the overall program. During that month, the first day is spent planning, the last day is spent reviewing, and in between, the team is developing the piece of software. At planning and review activities, all hands are on deck—attendance and participation from the user community is not optional, it is mandatory. These are not “grab a doughnut and hang out at a meeting” events; they are instead “grab your espresso, roll up your sleeves, and contribute” sessions.

During the planning day, users, led by the Product Owner, prioritize the Sprint based on what has been completed so far and what remains to be done. This activity defines the way a program slowly evolves over the course of the project. The importance of one aspect or another of the software changes, new requirements are discovered, and implementation details are refined.

Once planning is completed, the collaboration within the Sprint is propelled by a daily Scrum, the most basic and widely

practiced agile technique. The user community, represented by the Product Owner, must be present at these sessions as well, although the Product Owner is mostly a silent participant. Rather than an open-ended session, the Scrum allows the development team, headed up by a Scrum Master, to share insights that they have gained in the past day while working on the software and to raise any impediments that they face. A Scrum is not a status meeting but rather a discussion of what has been learned and what this new information means to the project. It is also a chance for users to glean knowledge that can be an asset for planning and prioritization in this and other Sprints. If a critical development issue, outside of the Sprint’s purview, arises during a Scrum, the Project Owner and other stakeholders take note to evaluate it in a separate meeting and determine whether it should be added to the product backlog or put on the back burner.

At the end of the Sprint, the development team presents the completed incremental portion of the software for review and acceptance. Occasionally, there are leftover software issues and even program features eliminated during the daily Scrums. During this review, these leftover items are added to the backlog to be handled later if the Product Owner decides to commit time and resources to them.

The GAO study found that this approach alienated some government agencies because it was too freewheeling; they feared that the development teams would produce iterative products that were “just good enough” to pass the initial review and that the remainder of the requirements moved to the backlog would never be addressed. In addition, these organizations expressed concerns about the difficulty of defining “done” in Sprint iterations because they were accustomed to seeing only completed software at the end, when changes and fixes were hard to do and not supported by the schedule. They have lived in a “go, no-go” assessment world instead of being empowered in the middle of the project to say, “Let’s change that.”

To an agile team, these fears are not warranted, because they are merely outgrowths of the normal progression for developing software. Because the Product Owner is engaged on a daily basis, prioritization and acceptance remain in the hands of the user community. Requirements and features can be altered and shifted at a moment’s notice; constant attention to the backlog and daily output ensures that nothing that must be addressed is neglected. Certainly, tight schedules and increasingly limited funding are a reality in the federal environment, placing boundaries on the software that is ultimately produced, but these constraints should not determine whether the program energizes its users and enhances their productivity. And by having the user community deeply involved throughout the development effort, they won’t.
AGILE SUCCESS STORY; US ARMY TRAINING PROGRAM

By the mid-2000s, the program that the US Army used for training was coming apart at the seams. More than a decade old at the time, the Automated Systems Approach to Training (ASAT) relied on old technology, and despite a slew of add-ons, patches, and workarounds over the years, the program couldn’t keep up with training needs, delivered inconsistent instruction, contained redundancies, and was expensive to maintain.

To replace ASAT, the Army decided to develop a new system, the Training and Doctrine Development Capability (TDDC), which would ostensibly be state of the art. This plan didn’t quite work out as hoped. While the TDDC was designed to take advantage of the Web and of gains in hardware capabilities, the program’s builders weren’t as forward thinking in their methodologies. Structured primarily around traditional waterfall development techniques, the project continued for several years and chewed through nearly $100 million. However, the resulting program never worked to anyone’s satisfaction. It lacked the basic functionality users wanted and it couldn’t handle even a minimum number of concurrent training professionals.

Implementation delays would have meant that users had to endure ASAT for a while longer, but the Army chose to scrap the TDDC entirely and replace it. This time, Army technologists were determined to try a different approach. A requirement of the newly drawn up Request for Proposals for the new Training Development Capability (TDC) was that the contractor use agile methodology, with collaborative teams, frequent iterations, constant load testing, and deep engagement by the user community. A fully working product was completed by 2008, less than 2 years after the project’s start—and there have been no hiccups. The system has been successfully rolled out to all of the Army training schools as a replacement for the ASAT system.

“Because of the first fiasco with TDDC, I came to the initial 30-day evaluation of TDC ready to fail it quickly and take an early flight home,” says Henry Koelzer, a retired artillery NCO and early evaluator of the project. But after just a few hours, he decided, “This system, and the agile programming methodology, was going to work.”

The primary failing of ASAT was its dependence on 1990s two-tiered, fat client architecture, which resulted in a wholly decentralized program. “Every school was a system in itself,” says Dennis Baston, who is retired now but was as a Supervisory Systems Analyst at the US Army Training Support Center.

For example, the training software used at Fort Knox’s armor school, Fort Benning’s infantry school, and Fort Sill’s field artillery school had to be loaded manually on servers at each of these locations. And because the applications were stovepiped, the installations at the separate schools could not practically communicate with each other. The sheer redundancy of the course-ware and the need to dedicate as many as 50 different servers exclusively to ASAT was a huge a drain on technology and financial resources.

What’s more, once a course was placed on the server, individual trainers at each school could tweak it to fit their perceived needs. As a result, there were multiple versions of each set of training materials floating around, and no way of knowing which was the most current. In fact, sometimes a course got so lost in the
system that it could only be found with an extensive search—and lots of manpower earmarked for it. At the Fort Knox armor school, for instance, after a search for the most current version of the weapons maintenance course, Army training professionals finally found it in the music school. “Who would have guessed those people were so hard core,” Baston says.

Baston adds that when congressional investigators and US prosecutors asked to see the training content related to interrogation methods used at Iraq’s Abu Ghraib prison, after military personnel were found to abuse inmates there in 2003, it was impossible to definitively decipher which version each soldier actually received.

Consequently, the Army’s goal in developing TDDC (and later TDC) was to provide an integrated and centralized repository of training products that were approved, under development, or being considered for general use. In addition, secondary benefits sought included eliminating the duplicate content and reduce the time to develop training products.

The contract to build the TDDC was awarded to what Koelzer calls “a major company; one you would immediately recognize.” Despite the vendor’s reputation and resources, the waterfall approach doomed the project from the start.

Following the typical waterfall techniques, program requirements were set in stone during the planning phase even before one line of code was written. No Army users—trainers or trainees—saw the interfaces and tested the functionality until TDDC was completed and delivered. “We gave them the use case, function points, and other major specifications, and when they were all done, they gave us the software, which was going to be a surprise, either good or bad,” recalls Baston.

The contractor tried to minimize the risk of the waterfall method by pairing it with spiral development techniques, which involves more testing and even agile-like iterations during the project, but the spiral model shares a fatal flaw with the waterfall model: the program’s requirements cannot change during development. So government evaluators were uncertain what they would see when the product was finally delivered. “So what you end up with is organized chaos,” Baston says.

But even given that the waterfall method doesn’t allow for modifications in project design, Baston says, “What was delivered didn’t meet the requirements that were specified in the first place.” He attributes this result to the fact that he and other evaluators could not see, and make corrections to, what was being produced until the very end.

For example, the system was supposed to support 6,000 training developers. But the software couldn’t handle a load anywhere close to that, perhaps fewer than 100. Baston pins the blame on the contractor’s testing process. Rather than assessing the system with real developers and in realistic numbers of concurrent users, the contractors used a few of its own coders and not in sufficient numbers to push the software to the breaking point.

The outcome couldn’t have been more of a disaster. After carefully evaluating the TDDC, Baston determined with 98-percent certainty that it could not be fixed and should be shelved. However, the project lead, a two-star general who was the deputy chief of staff for operations, was not willing to trash such an expensive effort even though Baston had given it a 2-percent chance of being fixable. Says Baston, “He wanted more certainty in our findings. So we had to go back and do more testing, more in-depth analysis, and we ended up with a 100-percent certainty that it was a complete, unrecoverable failure.”

The project was then rebid, this time as an agile development effort. Phase 1 of the TDC, which began November 1, 2006, was a 30-day demonstration phase, at the end of which the prospective contractors had to demonstrate a prototype to a packed house of about 30 government evaluators. On the basis of this session, the contract was awarded to the contractor team of Unitech, Booz Allen Hamilton, and MPRI.

One immediate advantage of agile methodology over the waterfall approach was its continuous performance testing regime even during the development of the software. For example, load measurements were conducted each month with an application that estimated total system capability based on the behavior of the program when accessed by a large number of concurrent users, as many as 30,000 by the time the software was ready to launch. In addition to merely issuing a “yea” or “nay”
on load performance, the testing software could identify choke and break points.

Assessments of agile program iterations were handled during Sprint review sessions, an agile term for intensive meetings when real-time decisionmaking about critical project issues occurs. During these meetings, potential TDC users were shown via video chat and collaborative desktop sharing software—a partially functioning piece of the system; an outline of what developers were going to work on in the next iteration; or an explanation of changing requirements based on Product Owner, development team, or stakeholder input. They were asked to experiment with prototypes and a so-called software sandbox that contained a working copy of the program, and to make suggested changes.

Baston and Koelzer recall that the unofficial leaders in each user group became pretty clear: they were people who had worked on prior failed systems and knew precisely where potential problems might be found and where this particular program could go wrong. As Baston describes it, there was a “gang of eight whose ideas counted for more than other members of the user community.”

Although each of the modules were rigorously tested and evaluated in iterative fashion throughout the project, none of the individual functions in the Army’s training program went live until the end. There was a lot of co-dependence among the modules, and all the pieces had to be completed before the system would work as a whole.

Contracts for agile development programs are among the more challenging aspects of these projects for the government. With so much iteration and potential modifications, contractors and agencies are often in the dark about exactly how much work will be needed to be redone and how much time that process will take during the project itself. However, Unitech, Booz Allen, and MPRI felt that TDC’s scope was well defined (based primarily on the documentation from the failed system), and the Army expressed a willingness to collaborate in minimizing costly surprises. As a result, the contractors accepted a firm fixed-price development agreement.

To protect the contractors from doing a lot of unanticipated, uncompensated work, the “reasonable person” rule was used. If it seemed reasonable for the contractors to be asked to do the extra tasks—that is, if it was a relatively minor fix—the contractors would be required to...
to do it at no cost. However, when it went beyond a small adjustment, the Army and the contractors negotiated ways to put more resources into that area of the system while streamlining other sections.

For example, the military had failed to include a critical security function in its system requirements. When that substantial shortcoming became evident, the development team and the government hammered out ways to make up for it, eventually agreeing to reuse some of the existing system accreditation documentation from the earlier programs. That freed up resources to tackle the security gap.

In the end the project came in on budget and on time. “What I saw happening was that there was an acceptance of the system from the user base as opposed to the contractor having to try to force its finished results on people,” says Baston.

The final phase of the project, including deployment, maintenance, and data conversion, lasted from July 1, 2008, through September 30, 2008, when the training system went live.

The project was successfully deployed, and now training professionals can access courseware via a Web browser and use the portions of it that they need without corrupting the original program. As new content is added to the courseware templates, the system keeps track of which version is the most recent and who is responsible for it. When an appropriate supervisor signs off on a new version, the updated training materials are marked as complete and are made available to anyone with TDC access.

TDC has already generated numerous critical improvements with tangible gains. Perhaps the single largest benefit is TDC’s impact on the preparation of course description documentation—known as Course Administrative Data (CAD) and Program of Instruction (POI)—which ultimately determine funding for training efforts. Accuracy is essential, so each CAD or POI undergoes a lengthy review by financial, training, and training development experts at Training Operations Management Authority (TOMA) before submission to the Department of the Army. With ASAT, schools submitted these documents by exporting their databases to hard drives, which were then mailed to TOMA. In turn, TOMA personnel would import the data onto their servers, indicate necessary changes, and then send the edited document back to the schools. The schools would make the required corrections, and the process would begin all over again. This system was so cumbersome that TOMA could barely meet the Army’s minimum requirements for training assessment.

In sharp contrast, under TDC, CAD and POI are sent to TOMA through the workflow architecture within the system. TOMA receives notification electronically, and its experts then make comments directly into the files and route them back to the school for changes. As a result, submission times to the Army for CAD and POI have been reduced from about 1 month under ASAT to 1 day under TDC.

In addition TDC’s security architecture permits compartmentalization of information not possible under ASAT. With ASAT, restricting which information each user had access to was a complicated process. As a result, sometimes unauthorized users would inadvertently edit or change a file that didn’t belong to them. By providing five separate domains, TDC allows supervisors to limit user access to only those programs they’re authorized to work on.

TDC also allows for consolidation of equipment, which reduces hardware, support and security costs, and complexity. ASAT ran on 78 different servers, each of which had to be housed in a restricted physical location. TDC runs on just a handful of web servers and a single database server.

Currently, TDC is used by almost 3,000 people on a daily basis and intermittently by an additional 3,000 users.
Measuring Progress

If you implement agile correctly, you will know it by your at-the-fingertips familiarity with the status of the project. During an agile effort, you are not merely briefed on the progress of the project from time to time or compelled to consult an IMS to get a 500-foot view of where things stand; instead, you actually see and feel the progress at any moment every day. Measurements are in real time, resulting in few negative surprises and a high degree of transparency when problems arise; that is critical because the sooner potentially deleterious issues are identified, the sooner they can be addressed.

Agile methodology offers a variety of techniques to visibly and accurately measure progress and promote project transparency. However, perhaps the most common-place and unique are so-called information radiators, which are the walls in the Scrum room containing detailed information, jotted down notes, and statistical graphics that provide insight into the status of the current Sprint as well as the overall project. Reviewed daily by the development team and the Product Owner, these information radiators usually include up-to-the-minute status of what is presently being worked on and what is yet to be accomplished in the current Sprint, as well as the progress made in moving toward a finished program. Often, the product backlog is clearly displayed, as are key development dates, upcoming vacations or time away, and who is responsible for which next step. Frequently, progress is illustrated by burn-down charts, which show how much work remains in the project, and burn-up charts, which indicate how much work has been completed.

For information radiators to be useful, it is essential that the Product Owner establish a rigorous definition of “done” so that the

Figure 3: Burn Up Chart
This chart shows progress in terms of Story Points (vertical axis) completed over time (horizontal axis).
gap between the “To-Do” and the “Done” is accurately measured and everyone involved in the project understands its parameters. (For example, is the project or an iteration considered finished only after complete documentation and training materials are produced? Or will the documentation be provided in the form of Help screens, which are written into the software during development?)

There is a misperception that supporters of agile methodologies claim it will help you develop software faster. That is not actually part of the argument for choosing agile. Rather, agile techniques enable developers to build higher quality software more in tune with the users’ requirements in an efficient and timely manner. With either agile or traditional approaches, initial estimates for cost or schedule may be askew, but agile methodologies alert you more quickly to any shortcomings, giving you ample opportunity to remove features from the release if the timeline or budget is inflexible. Put another way, you can make deliberate and thoughtful choices during development, when things are relatively calm, rather than after the deadline is reached and, as with HealthCare.gov, the whole country, including the President, is paying close attention to every misstep.

Progress in an agile project must be measured in rhythm with the incremental software deliveries unique to this methodology. Each Sprint should be viewed as a discrete effort—with the leftover issues, such as defects or missing functionality, placed in the backlog; the significance of these “leftovers” to the project helps determine whether the original schedule can be met or must be adjusted. As the most important of these items are pulled from the backlog to be addressed in a Sprint, the information radiators will indicate whether and how these decisions will affect the software’s intended release date.
Compare this process with traditional methodologies, in which all testing takes place at the end of the project. With traditional approaches, as the launch nears, the number of defects increases, although it will be impossible to say how many flaws exist in the program until it is tested. If the number of defects is greater than what you can address in the time available, then the only option is to deploy the software with known problems, which in itself is an undesirable option, or delay the launch, which is equally undesirable for different reasons.

In an agile project, however, because measuring progress is completely transparent, there are actually two acceptable choices: address systemic issues that are causing the defects during the development effort and perhaps avoid creating other flaws later on, or minimize program content to account for the defect rate. Which situation would you rather be in, 100-percent functionality with shaky performance and/or delayed launch, or less functionality with excellent performance delivered on time?

**Conclusion**

If you believe you adopted agile development approaches but your organization hasn’t changed in the process, then you didn’t. The change will be obvious, manifested in greater user involvement in development, more accurate and transparent assessments of a project’s health, contracts based on incremental success, user enthusiasm for the new software, fewer defects upon launch, and more internal innovation involving IT efforts.

One of the chief reasons that government agencies fail to enjoy the gains from agile techniques is that they have become less immersed in project development in the past few decades, when agile demands the opposite. Ironically, in an effort to minimize inefficiency in government, agencies have, in effect, distanced themselves from day-to-day software building efforts and outsourced project management to contractors. The attitude has been that if organizational management participates in the upfront part of the project when, in waterfall methodology, the requirements are delineated, then the technologists can build the system to those specifications.

Unfortunately, that is just another example of shortsightedness that has grown out of the best intentions. And now many federal organizations are finding themselves playing catch-up or, worse yet, stuck in IT quicksand—still incapable of making sense of the Agile Manifesto, more than a decade after it was created.

The questions that you should ask yourself are: Is your organization experiencing the same mistakes made over and over in critical development projects? Have overruns, flaws, unenthusiastic users, unanticipated delays, and less than satisfactory results become the norm? Agile methodologies can indeed pull you out of the quicksand, and better yet, help you avoid stepping into it in the first place. However, the overall effect of adopting agile cannot be stressed enough: agile is transformational—your organization and its culture will almost certainly need to be retooled and redesigned to drive the individual engagement and new ways of looking at software development that will lead to agile’s more positive outcomes. And transformation always needs a champion. Are you that person?
About Booz Allen
Booz Allen Hamilton has been at the forefront of strategy and technology consulting for 100 years. Today, the firm provides services primarily to the US government in defense, intelligence, and civil markets, and to major corporations and not-for-profit organizations. Booz Allen helps clients achieve success today and address future needs by applying functional expertise spanning consulting, analytics, mission operations, technology, systems development, cybersecurity, engineering, and innovation to design, develop, and implement solutions.

Booz Allen is headquartered in McLean, Virginia, employs more than 22,000 people, and had revenue of $5.48 billion for the 12 months ended March 31, 2014. In 2014, Booz Allen celebrates its 100th anniversary year. To learn more, visit www.boozallen.com. (NYSE: BAH)

© 2014 Booz Allen Hamilton, Inc.
C12.018.14

FOR MORE INFORMATION

Shawn M. Faunce
Senior Lead Technologist
faunce_shawn@bah.com

Dan Tucker
Principal
tucker_dan@bah.com

Haluk Saker
Chief Technologist
saker_haluk@bah.com

Wyatt Chaffee
Senior Lead Technologist
chaffee_wyatt@bah.com