

When Software Development is Your Means Not Your End: Abstracting Agile Methodologies for End-User Development and Analytic Application

Troy Martin Hughes

ABSTRACT

Agile methodologies for software development, including the Scrum framework, have grown in use and popularity since the 2001 Manifesto for Agile Software Development. More than having obtained ubiquity, Agile demonstrably has defined software development in the 21st century with its core foci in collaboration, value-added software, and flexibility gained through incremental development and delivery. Although Agile principles can be extrapolated easily to other disciplines, Agile nomenclature, literature, and application too often remain focused on software development alone. References to Agile often depict an idealized “developer” archetype whose responsibilities seem focused narrowly on the production of releasable code, in a role that often omits operational and other non-developmental activities. In contrast, SAS practitioners, while savvy developers, often view code generation as a process not a product, and instead measure success through data analysis, data-driven decision making, and the release of analytic products. Thus, while analytic endeavors do not contravene Agile principles, their expanded responsibilities merit a unique interpretation and implementation of the Agile framework. To that end, this text describes the implementation of Scrum-based Agile methodology for multi-disciplined end-user developer and analytic teams.

INTRODUCTION

The Manifesto for Agile Software Development (aka, the Agile Manifesto) was cultivated by a group of seventeen software development gurus who met in Snowbird, Utah, in 2001 to elicit and define a body of knowledge that prescribes best practices for software development. The Agile Manifesto states:

We are uncovering better ways of developing software by doing it and helping others do it.
Through this work we have come to value:

- Individuals and Interactions over processes and tools
- Working software over comprehensive documentation
- Customer Collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.ⁱ

In addition, twelve Principles of Agile Softwareⁱⁱ augment the Agile Manifesto and are espoused by methodologies that embrace Agile, including Scrum, Lean, Extreme Programming (XP), Crystal, Scaled Agile Framework (SAFe), Kanban, and others. Agile has been instrumental in accelerating and transforming software development teams and largely has supplanted the classic Waterfall methodology in the software arena. Major corporations such as Amazonⁱⁱⁱ, Yahoo^{iv}, Marriott^v, and even SAS^{vi} have demonstrated the revolutionary impact of Agile on IT innovation and implementation. Within the federal government, the Government Accountability Office (GAO) in a multi-agency study identified Agile’s “modular development” as a best practice^{vii}, while agencies including the Department of Defense (DoD)^{viii}, Federal Bureau of Investigation (FBI)^{ix}, Department of Veterans Affairs^x, Department of Homeland Security (DHS)^{xi}, and others each has validated Agile’s role in positively transforming fledgling or failing IT projects.

Despite the widespread success and popularity of Agile in the software development arena, as well as its theoretical applicability to non-technical projects, Agile training, literature, studies, and practice tend to focus stubbornly on its software development roots alone. Moreover, Agile authors and enthusiasts often portray an unrealistic paradigm in which software developers are freed from non-development responsibilities—including operational, ad hoc, and administrative tasks—to code in unfettered bliss. In this restrictive paradigm, software typically is considered to be produced only for an external customer and, once completed, is deployed to the customer or to an operations team for use or execution. Developers diverge from their product as the project ends, moving on to subsequent

development initiatives while their software is utilized operationally by the customer or internal or external end-users. While accurate for some, this limited interpretation of Agile obfuscates the generalizability of its methods.

Throughout academia, government, and industry, end-user developers code software solutions for internal consumption that provide analytic insight, reduce costs, boost profits, produce efficiencies, and increase functional capabilities. In contrast to the restrictive developer paradigm above, these analysts directly utilize the applications they create and have responsibilities that additionally may include data analysis, decision making, publication, and delivery of other business value. To a developer who has just completed a complex data cleansing program, this delivery may represent project conclusion. To a statistician, data scientist, or other analyst, however, the fun may only be beginning when the data have been cleaned, because this represents the point at which the analyst can begin to massage, morph, and model the data. Thus, to many analysts, end-user developers, and their customers, software development represents only an efficient means to an end; releasable code, in and of itself, is not the product.

The ubiquity of personal computers in the workplace and home, coupled with endless open-source and freeware development software, ensure the “end-user developer” model will continue to thrive; yet, it is not without challenges. Because end-user developers often lack formal software engineering education and training, they are less likely than traditional software developers to exhibit established software development best practices. These best practices include core coding concepts and competencies, as well as project management and development methodologies like Agile that facilitate a successful and efficient project life cycle. This text describes a more inclusive Agile paradigm by complementing theoretical Agile concepts with analytic constructs, complexities, and challenges. It is not intended to be a compendium of Agile history or methodologies, nor a roadmap to Agile implementation. Rather, it represents a succinct introduction to Agile nomenclature, roles, methods, and artifacts, sufficient to demonstrate the abstraction of Agile methods for analytic application.

INTRODUCTION TO AGILE

No superhero ever has gained notoriety without the juxtaposition of an archvillain: Superman has Lex Luther, Batman has the Joker, and Agile has the Waterfall. Even the most concise introduction to Agile requires a brief acknowledgement of the contrasting Waterfall methodology that, while still applicable to many disciplines, largely has obsoleted in the software development community. A Waterfall project contains incremental yet sequential processes that often include planning, analysis, design, testing, deployment, evaluation, and maintenance. In this stage-gate or phase-gate model, the project initiates with extensive planning designed to chart its entire course and continues through a series of discrete phases that typically do not overlap. Documentation is extensive, flexibility both is minimized and devalued, and the customer often receives no business value until project completion.

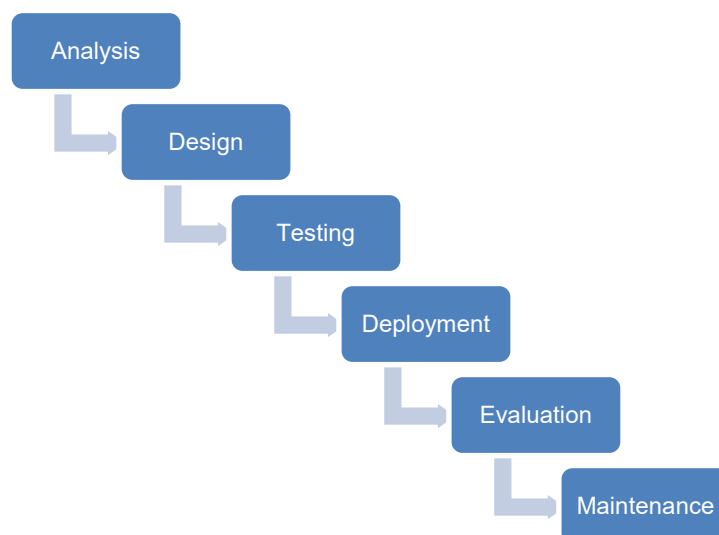


Figure 1. Waterfall Project Development Process

To be clear, the Waterfall methodology is appropriate for many industries and projects. Building a multi-lane bridge requires finite planning and specifications to ensure safety and that the ends meet in the middle. Business value—motorists' ability to traverse the bridge safely—only can be delivered upon project completion. The complex planning, extensive approval process, and sequential nature of these steps warrants a framework like Waterfall to ensure project success. However, in software development—and, arguably more so in data analysis—the final product often cannot be conceptualized fully at project outset. Complex projects that benefit from flexibility and in which the final product may be difficult or impossible to predict during initial planning benefit significantly from Agile principles.

Agile contrasts with Waterfall in several ways, the most notable being the incremental release of business value throughout the lifespan of the project, with the highest priority functionality (e.g., components or requirements) developed first. Business value (or “releasable code” in traditional Agile parlance) is released at the conclusion of each iteration, a time-boxed epoch that typically lasts between one and eight weeks. Within each iteration, a microcosm of planning, development, testing, deployment, and integration activities occur, thus Agile is often conceptualized as an sequence of short Waterfalls. While accurate, Agile principles and not merely its methods or mechanics distinguish it from and contribute to its success over Waterfall.



Figure 2. Agile Project Development Process

“Individuals and interactions,” as the first tenant of the Agile Manifesto, are central to its successful implementation. In describing the characteristics of a “generic” or traditional project life cycle, the Project Management Body of Knowledge (PMBOK) states that “Stakeholder influences are greatest at the start of the project [and] decrease over the life of the project^{xii}.” In contrast, because each Agile iteration contains a slice of every life cycle phase, Agile projects require continuous stakeholder influence throughout the project. In an ideal environment, the principle stakeholder (i.e., customer or product owner) is collocated with the development team and thus is able to monitor incremental development throughout the project and more adroitly can make modifications in scope. Developers also benefit from collocation because they are more focused on customer vision and priorities and thus better able to understand the context of the business value they provide. Moreover, mutual respect is maximized because the customer trusts developers to determine *how* to deliver business value, while developers pledge to deliver only *what* the customer has prioritized, thus inhibiting both customer micromanagement and rogue, freelance development.

Because value-driven change is embraced, Agile methodologies often are considered to be “change-driven^{xiii}” or “value-driven^{xiv},” whereas Waterfall and other traditional phase-gate methodologies are “plan-driven.” This is not to imply that plans are absent from Agile, only that deviations therefrom to increase business value or otherwise advantage the customer are applauded. Thus, an Agile customer is quick to realize and react to sunk costs and opportunity costs and can shift course to maximize business advantage. As responsible change is not perceived to be a failure, Agile projects are often said to “fail often and fail fast,” as customers redirect developers from low-value activities and requirements toward higher-value ones. This partnership reduces the mutual frustrations that occur when developers deliver a product whose utility, relevance, or value is diminished because requirements static since project outset have become outmoded.

AGILE ROLES AND RESPONSIBILITIES

Although individual methodologies may describe additional roles, Agile primarily distinguishes three team roles: the customer, the team of developers, and the facilitator. Throughout this text, “team” is utilized to specify the cooperative aggregate of these three roles. Agile inherently does not deny the existence or influence of additional stakeholders, such as project sponsors, project managers, functional managers, or a project management office (PMO.) A traditional project manager, however, who maintains a command and control authority, will find his project management responsibilities diminished as Agile embraces a facilitative, servant-leader approach. The customer or customer representative instead represents the sole voice of project vision and task prioritization, ensuring that confusion does not erupt from absent, multiple, or conflicting chains of command.

The customer (or product owner in Scrum) represents the single stakeholder who provides project vision, requirements, and a roadmap to the team and who ultimately must sign off to accept task and project completion. In the responsibility assignment (aka RACI) matrix, the Accountable party is the customer who may or may not have formulated the project vision, but who nevertheless is responsible for conveying it to the team. Where a cohort of customers exists, each of whom has project authority, a single customer representative must be designated to ensure a unified, coherent customer voice. As the business expert, the customer is best placed to define business value and is responsible for prioritizing components, functions, or development activities that will yield the highest value. Contrasted with more traditional project management approaches, the Agile customer should not instruct a team on *how* to complete a project, but rather should provide requirements for project acceptance criteria. In Agile, these criteria collectively often are referred to as the “definition of done” and, while flexible, must be agreed upon by key stakeholders at project outset and after any potential revisions in scope.

The team of developers represents the collocated individuals who are dedicated to complete the project. Although this role references a “software developer” in the original confines of the Agile Manifesto, a more generalizable usage denotes anyone who delivers developmental as opposed to operational value. In abstracting the development team to analytic projects, these roles may comprise analysts, business analysts, developers, and other multi-disciplined members directly adding business value through decision-making, functionality, increased efficiency, and other methods. Agile developers are described as self-organizing and self-managing, in that they are able to determine on a daily basis what requirements or tasks they are completing as well as how they are completing them. The team collectively is responsible for completing a committed body of work during each iteration but, being comprised of subject matter experts (SMEs), is trusted to complete its work efficiently without authoritative supervision.

The facilitator (or ScrumMaster in Scrum or Coach in XP) represents the servant-leader who upholds Agile principles and removes obstacles to ensure smooth delivery of business value from the developers to the customer. The facilitator has no management function but instead should be an expert in Agile principles and able to collaborate with all stakeholders. As an expert, the facilitator ensures all stakeholders respect Agile principles and any variations thereof established by stakeholders. As a coach, the facilitator should have the authority to mentor the customer, development team, and additional stakeholders in the healthy implementation of Agile, but ultimately has no enforcement authority. In smaller teams and contract-mandated environments in which each member is a developer and must deliver business value, a dedicated facilitator billet may be infeasible. In these circumstances, an analyst or developer may perform the facilitator functions but would be ill advised to coach or mentor the customer in Agile principles, which could represent a conflict of interest given his developer role. One of the greatest responsibilities of the facilitator is to “protect the iteration,” ensuring that once an iteration has commenced, no additional work can be assigned by the customer unless, only in rare circumstances, other requirements are removed from that iteration.

AGILE PROCESS AND SCRUM ARTIFACTS

Methods, tools, and activities that support Agile methodologies commonly are referred to as artifacts and, as Scrum is the most utilized Agile framework, Scrum artifacts and terminology have become ubiquitous to Agile. Just as the absence of specific stakeholder roles in Agile definitions does not demonstrate their exclusion, so too does the lack of an artifact appearance not indicate its absence from Agile. For example, contracts, project charters, vision, and other project management artifacts do exist but fall outside the scope of some Agile literature as well as this text. Other artifacts central to traditional project management are replaced in Agile, such as the transformation of the work breakdown structure (WBS) into the significantly curtailed and more flexible product roadmap. In this abridged introduction to Agile, the following artifacts are described below and, for reference, include:

- team agreement – specifies the team's commitment to Agile principles and local variations thereof
- product backlog – customer-prioritized list of high- and low-level functional requirements for the project
- product backlog grooming – continual improvement and prioritization of the product backlog by the customer
- sprint backlog – list of low-level functional requirements (i.e., user stories) current being developed
- iteration or sprint – a typically one to eight week cycle during which scope is frozen while developers work
- user stories – low-level functional requirements that are completed during an iteration by developers
- sprint planning session – meeting in which the team establishes and commits to complete the sprint backlog
- daily stand-up or scrum – daily meeting in which developers report current work status and impediments
- self-organizing, self-managing development – the developers do their thing without distractions or obstacles
- sprint review – a demonstration by the developers to the customer and stakeholders of product functionality
- sprint retrospective – a developer meeting to flesh out best practices and elucidate areas for improvement
- velocity – the numeric representation of the quantity of development work completed in an iteration

The most important and enduring Agile artifact is the team agreement that directs and specifies the implementation of Agile within the team. Even within a breed of Agile such as Scrum, variations in practice will exist, thus intentionally established exceptions that increase efficiency or cooperation must be distinguished from exceptions that result from apathy, ennui, a lack of knowledge, or an intentional deviation toward unhealthy behavior. As a team matures in its implementation of Agile principles, a decision to modify the team agreement (e.g., switching from two to one week iterations) should be intentional, communicated, accepted, and practiced. Juxtaposing its commitment to Agile, a team optionally may adopt and include best practices such as non-functional requirements that ensure accurate, timely, and efficient delivery of functional requirements.

The product backlog represents the prioritized list of high- and low-level functional requirements created by the customer. In Agile, requirements often are decomposed into smaller user stories—succinct examples that demonstrate functionality, and whose complexity and level of effort can be estimated by the team of developers. User stories are continually added to, removed from, and prioritized by the customer in an activity known as product backlog grooming. Thus, as the project progresses, flexibility is maximized because requirements can be added or removed as priorities shift. When appropriate, the facilitator and/or developers also can assist the customer in creating user stories, especially in cases in which they have a higher level of technical ability. Thus, while top-down requirements are more common, bottom-up requirements generated by developers often highlight necessary yet overlooked components or functionality—such as back-end processing or data infrastructure—and should be suggested to the customer. Because requirements prioritized toward the top of the product backlog will be completed in the next iteration, they should be decomposed into low-level user stories through rolling wave planning or progressive elaboration before the sprint planning session. Developers should never work on user stories from the product backlog; user stories must first be transferred to the sprint backlog by the client before work can commence.

The sprint backlog includes the prioritized list of decomposed requirements or user stories that have been selected for development in the current iteration. Whereas the customer is encouraged to groom the product backlog, the sprint backlog is frozen at the beginning of the iteration and cannot be modified by the customer or the team. In strict Scrum, this rule has no exceptions. In practice and especially in fast-paced analytic environments in which the results

of one user story might influence other concurrent user stories, the customer should be allowed to remove requirements that have lost value or add emergent requirements of extremely high value. Thus, while iteration scope constancy should be respected and encouraged, it should not be considered to be an immutable object and thereby become an obstacle to agility. Rather, if a customer does need to reprioritize requirements within the current iteration, care should be exercised to ensure incomplete user stories of equal labor intensity are removed from the sprint backlog and returned to the product backlog. Sprint backlog reprioritization should be considered the exception and not the rule and, if frequent reprioritization occurs, the team should consider decreasing the standard iteration length to gain additional agility.

The iteration (or sprint in Scrum) is the time-boxed epoch that lasts “from a couple weeks to a couple months” according to the third of twelve Agile principle. In Scrum, sprints always are defined as 30-day increments; however, Scrum principles and artifacts can be applied successfully to projects that have iterations of shorter or longer durations, based on the needs of the project. Although the customer should be available throughout the iteration for collaboration and to provide guidance to developers, the scope of work in the current iteration should remain constant until its delivery during the sprint review at the close of the iteration.

The first day of each iteration begins with a sprint planning session designed to provide guidance, set expectations, and establish a scope of work for the remainder of the iteration. Planning typically is divided into two halves, with the customer first selecting the desired user stories from the product backlog, and the development team next accepting and planning the work. In the first half, the customer presents the highest prioritized user stories to the team and, after understanding the context and intent of each requirement through discussion, the developers commit to an established body of requirements that becomes the sprint backlog. User stories that are too complex or lengthy (e.g., comprising several days of work) should be further decomposed into lower-level, more manageable requirements. In the second half, the customer may or may not be present while the developers further discuss the content, complexity, challenges, and time commitment of each user story. Because the team is self-organizing and self-managing, they also collectively decide who will be responsible for completing each requirement, and formally accept the body of work by committing to the iteration. If after this more elaborate discussion the team believes it cannot complete the entire sprint backlog, specific user stories are rejected or exchanged for less time consuming stories until the developers and customer are in agreement. At the conclusion of the planning session, both the customer and developers must believe that realistically each of the committed requirements on sprint backlog will be completed by the iteration close.

Each subsequent day of the iteration begins with a daily stand-up or scrum that is time-boxed to last no more than 15 minutes, and in which the team is encouraged literally to stand to foster brevity. In traditional Scrum framework, distractions and sidebar conversations are eliminated, and each developer succinctly answers only three questions:

1. What have you done since the last scrum?
2. What will you do between now and the next scrum?
3. What got in your way of doing work?^{xv}

The questions are intended to focus and inform developers and, if necessary, demonstrate where resources should be shifted if some requirements are behind or ahead of schedule. The facilitator and customer, noting any impediments, also should spring into action to remove obstacles from the team. Because the status of each developer is known and understood, resources or personnel nimbly can be shifted to ensure bottlenecks do not slow progress. Succinct follow-up questions are encouraged during the stand-up, but conversations and collaboration should occur after and not during the meeting.

Each day of the iteration, following the stand-up, the remainder of the developers' time belongs to themselves to organize and prioritize as necessary. Some user stories will have dependencies within the iteration, prompting the team to coordinate its workflow, but no customer, project manager, or functional manager is responsible for daily tasking or monitoring workflow. Due to this mentality and the lack of micromanaging, some incorrectly perceive Agile as being a lawless frontier in which developers code or produce products in a void of guidance or supervision. On the contrary, Agile methodologies stress continuous interaction between developers and the customer to foster a “one team one fight” mentality. Many customers attend daily stand-ups to maintain awareness and regularly view developers' work to monitor progress throughout the iteration. Although the customer is unable to change the scope

of the committed user stories during the iteration, he is encouraged to interact with developers to observe work in progress, convey vision, clarify ambiguity, provide guidance, answer questions, and remove obstacles.

By the close of each iteration, all committed user stories in the sprint backlog should have been completed. If a requirement was in jeopardy of not being completed, this obstacle should have been identified previously and reported in daily stand-ups, giving the entire team the opportunity to collaborate and attack the issue. This not only informs the client but also allows the developers proactively to shift resources if possible. Because the development team collectively owns the committed work, it will either succeed or fail as a whole, thus developers are encouraged to espouse and benefit from aggressive collaboration. Regardless of what obstacles are encountered or work is delivered on time, the customer never should be surprised at iteration close that work is incomplete. And, should developers complete the sprint backlog early, new user stories can be selected from the product backlog and, in concert with the customer, added to the current iteration.

On the final day of each iteration, two meetings bring closure and prepare the team for future success: the sprint review and the sprint retrospective. The sprint review is a demonstration by the developers to the customer of functional user stories completed during the iteration. In traditional Scrum, the demonstration is always “releasable code” and the customer actually sees it run or interactively executes the product. Agile methodologies stress that rather than merely produce documentation or a PowerPoint presentation *about* a product, the customer should be able to test drive the new functionality that has been added in that iteration. In this way, Agile maintains its “fail often, fail fast” mentality because the customer immediately can provide feedback and redirection if the demonstrated work does not match the committed user stories, if project scope has changed, or if additional business value can be added through other methods.

The final piece of every sprint is the sprint retrospective. Not to be confused with the review, the retrospective is a chance for developers to discuss amongst themselves the overall impression of the iteration—essentially, what worked well and what did not work? The retrospective is especially critical for teams new to Agile. Based on professional, personal, and cultural differences, some team members easily may adopt the Agile philosophy, while others may struggle in a self-managing vacuum. Early in Agile implementation, teams also are more likely to over- or under-commit to user stories per iteration due to estimation errors. As teams mature, their velocity—i.e., the numeric representation of the cumulative amount of work they complete in an iteration—should become both increasingly stable and accurate. A number of popular Agile methods exist (e.g., planning poker, relative sizing, affinity estimation) that allow developers more accurately to estimate complexity and time commitment of user stories, but these lie outside the scope of this text. If the team observes that the customer often is shifting priorities during an iteration, they should discuss the possibility of shortening the length of each sprint to gain flexibility. Thus, the sprint retrospective is the ultimate opportunity for the development team to self-manage in maximizing its efficiency.

AN AGILE BIAS

Agile methodologies radically have revolutionized the software development industry, demonstrating widespread success and applicability to diverse projects for over a decade. Its implementation and instruction, however, seem not to be without bias, even within the software development community. The majority of Agile literature depicts a developer who endlessly codes in an object-oriented programming (OOP) language, freed from distraction and external responsibility. This endless coder paradigm, in part, is borne out of a necessity not to want to burden teams with gratuitous methods, rules, and regulations included in so-called heavy methodologies. Agile, after all, is a philosophy that embraces flexibility, not prescription. Scrum even goes so far as to reject the “methodology” label and instead heralds itself as a “light framework.” While heavy methodologies might seem onerous or rigid, too little prescription or information defined for too narrow an audience may leave incredulous professionals questioning Agile’s applicability to their specific role, team, or industry.

A 2014 study^{xvi} by the author examines trends in job postings in open-source OOP third generation languages (3GL) Java and Python—embraced by the majority of Agile literature and instruction—contrasted with trends observed in proprietary data analytic fourth generation languages (4GL) SAS and SPSS. The open-source OOP data analytic 4GL “R” also was examined because it increasingly represents a no-cost solution to organizations that have grown weary of initiation and annual license fees sought by proprietary data analytic software companies. Between June 1 and July 31, 2014, 56,948 job descriptions on Indeed.com identified at least one of these five software languages in

one of the ten most populous United States cities, including: New York City, Los Angeles, Chicago, Houston, Philadelphia, Phoenix, San Antonio, San Diego, Dallas, and San Jose. Cluster analysis demonstrated a strong cluster of data analytic languages SAS, SPSS, and R and a similar cluster pairing of more generalized 3GL languages Java and Python. While this clustering was expected, further results demonstrating the degree to which each cluster did or did not embrace Agile methodologies were shocking.

Content analysis ensured that references to Agile methodologies were validated and not simply uses of the adjective “agile.” In Java and Python job descriptions, Agile was referenced 20.6% and 17.0%, respectively. Data analytic positions, however, were significantly less likely to reference Agile, with SAS, SPSS, and R job descriptions at 3.0%, 2.3%, and 3.0%, respectively. Moreover, when a pool of more “pure” data analytic positions was isolated by removing positions that referenced either Java or Python, the incidence of job descriptions that referenced SAS, SPSS, or R using Agile dropped to 1.1%, 0.4%, and 0.8%, respectively. Thus, data analytic positions were nearly 20 times less likely to reference Agile methodologies than their 3GL counterparts!

Caution must be exercised in extrapolating these results to infer that data analytic teams in reality are 20 times less likely to espouse Agile principles than traditional software development teams. Many job descriptions are outdated or do not accurately portray the body of work to be performed, the tools or software utilized, or the preferred project management philosophy. Thus, a job description may extol the virtues of Agile and seek a lead developer role with “experience as a ScrumMaster or facilitator” yet, in practice, the work is completed in a vastly disparate authoritarian, command and control environment. A highly functional analytic development shop, conversely, may espouse Agile principles such as open communication, responsible flexibility, and self-organizing and self-managing behaviors, yet choose not to brand itself as “Agile.” Job descriptions also typically enumerate the knowledge, skills, training, certifications, education, and past experience required for a position as well as demonstrate with varying degrees of specificity how these requirements benefit job functions. Less common, however, are job descriptions that detail through what methodologies, modalities, or practices their stated requirements are applied to job responsibilities and functions. Notwithstanding these and other potential threats to validity, the results demonstrate a strong predilection for developers in data analytic environments who utilize software such as SAS not to practice Agile methodologies, and to embrace Agile far less often than their 3GL counterparts.

SO IF AGILE + DEVELOPMENT = ☺, WHY DOES AGILE + ANALYTICS = ☹ ?

The tendency for data analytic development environments not to espouse Agile principles is disappointing. This could contribute to project inefficiencies or failure for the customer who leads a team of savvy data analytic developers but who does so with a heavy-handed, Waterfall perspective. Software developers who code relentlessly in SAS, SPSS, R, or other similar software packages but who do not embrace Agile methodologies also may limit their potential to deliver the most salient, business-value to their organization. And, because Agile has become the industry standard for software development, best practices in software development such as code quality, minimization of technical debt, acceptance testing, and functional testing that are infused throughout Agile literature and practice are more likely to be minimized or omitted in data analytic development environments. Moreover, analysts who utilize data analytic software for development may do so for only a fraction of their day, while the remainder is spent developing models, writing analyses, producing reports, or conveying results to decision-makers. Each of these tasks, however, is a development activity that can benefit from Agile principles when it is abstracted from a restrictive software development realm to the more inclusive domain of generalized development.

Several differences exist in the implementation of 3GLs versus 4GL data analytic software packages, each of which may contribute to the prevalence of Agile usage in the former and paucity in the latter. These include software capability, focus, intended user base, intended audience for products of the software, and expectations or requirements for those products:

1. 4GL data analytics software is simpler but has inherent limitations in its functionality. This is intentional and, despite being robust and powerful applications, 4GLs are designed for users who will operate the software using a finite number of functional procedures that address specific requirements. A SAS user can confidently run the ANOVA procedure out of the box or can customize functionality or results through coding a limited number of parameters. The python developer, conversely, is able to write his own ANOVA procedure with endless

customization potential. Each software application has advantages and disadvantages that attract a different cadre of developers and which fulfills a different business niche.

2. 4GL data analytics software focuses on data transformation. Data analytics development emphasizes transforming data from one state to another to produce some result. Often, once that result is analyzed, further development and subsequent analysis is required, some of which often could not have been predicted. This contrasts with many traditional software development projects in which an application is being built that will have a limited scope of input, as opposed to vast arrays of data. Russell Jurney, in arguing in favor of the value of Agile to data science, elucidates the complexity that “Working with real data in the wild, doing data science, and performing serious research takes time—longer than an agile cycle...Data science is stuck on the old-school software schedule known as the waterfall method.^{xvii}” Scott Ambler presents a similar concern that “Although both evolutionary and agile ways of working have been readily adopted within the development community, the same cannot be said within the data community. Most data-oriented techniques are serial in nature, requiring the creation of fairly detailed models before implementation is “allowed” to begin.^{xviii}”
3. 4GL data analytics software is intended for use by a continuum of professionals, from developers to non-developers. SAS, for example, offers users the ability to run analyses and develop complex data flows entirely through its graphical user interface (GUI.) More technologically savvy users subsequently can export, modify, and re-use the code that is produced by the GUI to gain additional functionality and customization. Over time, as they find efficiency in—or themselves tasked with—developing increasingly complex code to answer analytic challenges, their abilities grow and their focus may shift from analysis to development. At the continuum extreme, veteran developers code robust, reliable applications that benefit from the multiple SAS modules, utilizing the SAS macro language, or the ability of SAS natively to interpret PERL, SQL, and other languages. In the vast professional world that encompasses all brands of data analytics, many analysts embarking on significant software development undertakings would not consider themselves to be professional software developers—their education, training, and subject matter expertise lie elsewhere despite their significant programming prowess. Thus, end-user developers are so termed not for a lack of skill, but rather because they are the primary beneficiaries of the functional advantage or business value produced by the software they build.
4. 4GL data analytics software often is utilized to develop software products for the developer himself or his internal team. Although the ultimate product may be a non-technical solution that does have a larger, external dissemination—for example, a report that is published or a data-driven decision—the software itself does not constitute the product, but rather a necessary, intermediate step. At best, analytic code may end up in an appendix to demonstrate the validity of a model or process, but typically will not be viewed or utilized directly by consumers of the analytic solution. This contrasts sharply with software applications that are developed to be utilized directly by customers or other users outside of the development environment. The end-user developer building software for himself may develop it under the fallacy that he will be able to run, refactor, and redesign it over time, thus it may lack adequate documentation, robustness, reliability, and other attributes of quality software. For straightforward, short-term, or single-use analytic development, this bare-bones approach may be adequate or even recommended, but it can inculcate habits of poor quality development that can poison a team or persist throughout a career.
5. 4GL data analytics software often is utilized to develop analytic products or solutions that have no technical or non-functional requirements, but which require substantial software development initiatives. Say what?! This is a HUGE discrepancy from traditional software development environments and best practices. This occurs when the customer—despite the degree of technical knowledge, professionalism, or capability of the development team—designs all requirements to address aspects of the analytic product or solution, rather than quality assurance aspects of *how* it was produced. Thus, the customer who is requesting the production of a comprehensive, quantitative report from a team of statisticians and accountants may only care about the veracity of the results, the format in which they are presented, and the timeliness in which they are delivered. In delegating requirements, this customer may fail entirely to specify the technical means or quality standards through which those results are to be achieved. In these environments, even when developers attempt to code quality into their software, they may be thwarted by a customer who views these ideals as gratuitous. Later attempts to retrofit this poor software with quality will be about as successful as polishing a turd.

TURNING THE AGILE ANALYTICS FROWN UPSIDE DOWN

Because of the above deficiencies in end-user software development, a defense and demonstration of its advantages is warranted. End-user developers typically are defined as professionals who are not professional software developers, so what is a software developer? Merriam-Webster defines a developer as “a person or company that develops computer software.”^{xxix} Software, correspondingly, represents “the programs that run on a computer and perform certain functions.” In a traditional sense, the term “developer” might conjure images of someone trained and educated in software engineering or computer science who spends the majority of his day writing code. In speaking of software developers, Alistair Cockburn states that “Historically, success in our profession came from being able to sit alone for long hours without talking to anyone, staring at papers or screens. Those who didn’t like that mode of work simply left the field.”^{xx} In fact, many software shops are so specialized that developers don’t even test their own code—they have testers for that—and teams distinguish development, production, deployment, and operations roles to further focus their craft. These artisans inarguably represent professional software developers and largely comprise the audience for and about whom Agile literature and training is intended. But software development and its authorship lie on a continuum and, by Webster’s definitions, anyone authoring even snippets of code can be considered to be a developer. Over time, as these snippets beget infrastructure and their authors gain knowledge and expertise, a formidable class of end-user developers is established and must be nurtured with best practices—such as Agile methodologies—that are afforded to the contradistinguished professional software developers.

The Network of Excellence on End-User Development defines end-user development as “a set of methods, activities, techniques, and tools that allow people who are nonprofessional software developers, at some point to create or modify a software artifact.”^{xxxi} Morch et al., describe a “user developer continuum” that includes regular users, super users, local developers, and professional developers, respectively in order of increasing technical training and proficiency.^{xxii} Regular users and super users are non-developers, although the latter group is able to personalize software settings and assist regular users. Local developers (i.e., end-user developers) understand programming but are limited and must rely on professional developers. Professional developers represent the highest technical proficiency and “work in software houses and are trained as software engineers and/or programmers.” Thus, the designation of “end-user developer” is not a disparaging one, but rather meant to credit SMEs in non-developmental fields as having obtained the additional skills sufficient to deliver business value through code development.

Because end-user developers are spawned from statisticians, researchers, biochemists, physicians, accountants, intelligence analysts, and a host of other professions, they are first and foremost SMEs who understand the nature of their business and, ostensibly, its business value. Whereas in most Agile literature the stereotypical developer may be responsible for interacting with users or business analysts to understand the context of the business problem (and, thus, the solution they are tasked to implement), end-user developers inherently understand that problem because it is their problem. Thus, a methodologist who codes a SAS solution to create dynamic HTML reports often has reduced his own workload, not someone else’s. The scientist who spends 80 percent of his project time to develop a comprehensive ETL infrastructure to clean and standardize his data does so not to “produce working software,” but rather to enable effective and accurate data analysis and reporting in the remaining 20 percent. And, as SMEs of their respective fields, end-user developers are more poised to communicate with their customers, who are of course the business experts who have delegated the development work to provide business value.

Throughout Agile literature, a team of high-functioning, cross-functional developers often is referenced. Some texts explicitly mention that this development team may include software developers, software testers, database administrators (DBAs), data architects, business analysts, and other roles as necessary. However, as Agile embraces a “generalized specialist” mentality, these team members often are expected to augment each other’s functional responsibilities. For example, in a team that comprises developers and testers, over time and depending on workflow and priorities, a developer may be required to help test or a tester help develop. Thus, working in a collaborative, collocated environment, the cross-functional development team is expected through hands-on experience and some degree of osmotic learning to absorb and perform functions of each other’s roles. A tester may never embrace the full panoply of skills manifested by a developer or ever fully assume the title or responsibilities, but should be able substantially to augment development work if necessary. Abstracting these ideals to an analytic environment, end-user developers easily assume the generalized-specialist role espoused in Agile because they are already operating under this paradigm, having both obtained the required business knowledge and technical knowledge to foster and facilitate success in both realms.

ANALYTIC AND END-USER DEVELOPMENT QUALITY

The benefits of Agile methodology application to analytic and end-user development by far outweigh the challenges and complexities that will be encountered. Still, care must be exercised to ensure the abstraction demonstrates Agile values and principles. Moreover, each of the five key differences enumerated above between data analytic software and 3GLs can vex a team, while their collective aggregation can seem like an insurmountable obstacle to Agile implementation. Very real limitations in flexibility and capability do exist in data analytic software languages when compared to their 3GL counterparts. Notwithstanding, developers should strive to ensure the same high caliber, quality software is delivered despite any real or perceived limitations.

For example, Dave Ingram's *Design – Build – Run: Applied Practices and Principles for Production-Ready Software Development* offers a scintillating view into building quality software systems. Many data analytic software developers unfortunately, however, might not even get past his introductory yet admonitory remarks: "To really get the most out of this book, you should be familiar with software design and development practices. You should be able to read flow charts and UML diagrams. You should also be familiar with software development and the concepts of unit testing and integration testing. Knowledge of more advanced topics such as code coverage, code analysis, and performance analysis is an advantage.^{xxiii}" In fact, many of these topics would be completely outside the realm of the SAS language. The collective goal they aim to achieve—the production of robust, reliable code—is within the grasp of SAS, thus savvy analysts may need to abstract these principles for their own use in data analytic software.

In his seminal introduction to XP that predates the Agile Manifesto, Ron Jeffries enumerates inalienable "Programmer Rights" and states that "You have the right to produce quality work at all times."^{xxiv} This is not only a right but also a responsibility. Quality, however, is subjective, both in the eyes of the customer and the developer, and a commitment to quality does not ensure consensus, but often represents a compromise in quality standards between all stakeholders. After all, in a zero sum game, quality can be traded for commodities such as increased project scope, decreased project duration, or a reduction of cost. Notwithstanding, the selected quality standards should be expressed explicitly in project requirements documentation or user stories so that quality can be planned and implemented effectively rather than in an ad hoc fashion. Quality should be baked into the brownies, not sprinkled on top later.

In analytic and end-user development, the final product or solution likely may not be a piece of software, but rather an analytic report, finding, or decision for which a significant software development undertaking was required. Quality standards for non-technical products or solutions will vary by discipline and should be firmly established by the customer, and thus are not discussed here. But if an analytic report is the product, then equal or higher quality standards should apply to the code producing the report as the report itself. Too often, customers focus or fixate on vacuous panache—for example, tediously torturing an analyst with the seemingly sempiternal selection of the right hue of purple for a graph—rather than ensuring quality assurance and quality control standards were maintained to produce valid data. By stating aspects and degree of quality—such as non-functional requirements—within user stories, the customer can ensure that quality not only will be maintained, but also must be demonstrated for task and product acceptance. And, although non-functional requirements are inherently more difficult to demonstrate than functional requirements, through testing and test documentation this can be accomplished with relative ease.

Developers too can violate code quality standards, by developing too little or even too much quality. Gold-plating typically refers to developers who deliver functional capabilities that were never requested or desired by the customer. Gold-plating also can include developing and delivering non-functional "requirements" that exceed the established quality standards for the user story, task, or project. For example, an analytic customer may require a simple ETL infrastructure be developed that will clean and transform incoming clinical trials data. A developer who invests a significant amount of time ensuring that the ETL processes are portable to both Windows and UNIX environments and scalable to accommodate multiple data input streams has violated quality standards if the customer did not require these capabilities. Thus, like everything else, non-functional requirements must be appropriate for the scale and intent of the project and must be customer-directed.

In order to establish quality standards, the customer and development team should discuss the following non-functional requirements for every project that relies upon software development:

- **required (aka, customer-directed)** – Developers should only perform work that is directed by the customer.
- **sufficient** – The developer should deliver a product that sufficiently meets the established definition of done, thus all required user stories must be both delivered and accepted.
- **timely** – Agile typically does not embrace purely time-boxed projects (i.e., those that freeze project duration) because it instead embraces a flexible project scope. Notwithstanding, where the customer has prioritized specific user stories within an iteration to meet obligations that may occur on the project roadmap, the development team should strive to deliver these requirements on time, lest they become obsolete or valueless.
- **reliable** – Reliability encompasses all aspects of functionality to describe software that simply does not fail.
- **modular** – Modularity is critical in Agile, because higher-level requirements must be decomposed into bite-sized user stories that can be distributed simultaneously across a team or temporally across several iterations.
- **unique** – Duplication should be avoided because it is inefficient and introduces unnecessary opportunities for faults in the code. The SAS macro language allows the user to develop dynamic code that can be re-used. Macro development inherently adds a level of complexity and, at times, opaqueness, thus its use must be balanced with the requirement for timely code and other priorities.
- **scalable** – A scalable solution often references its flexibility in accommodating a large number of users or a large amount of data. In data analytics, solutions often seem like good ideas until faced with the infamous big data challenge, at which point refactoring to discover and implement efficiencies becomes a priority to developers.
- **extensible** – The ability to add additional functionality or components to software describes its extensibility. Developers often want to code longer-term infrastructure to support “future” endeavors, while a customer may be interested only in current priorities. When push comes to shove and the customer wins, extensibility, while admirable, should be reduced to accommodate a more timely delivery of the immediate, requested solution.
- **portable** – Developers who code in a single operating system (OS) may think they do not have to worry about portability—but they’re wrong. Portability also reflects backwards compatibility of code and its ability to be interpreted accurately by future software releases. For example, understanding what SAS features are being deprecated in v9.4—or Python features in v3.5—will facilitate future code reliability.
- **re-usable** – In software development parlance, this typically refers to the preference to develop code that is intuitive, dynamic, and documented sufficiently that it can be re-used by other instances, applications, and developers. In development for data analytics, while this principle is also true, additional benefits are realized when underlying data structures also can be re-used. Thus, a single ETL process that ferries data until the last possible point of divergence is preferred over multiple processes that perform overlapping or redundant ETL functions and thus unnecessarily introduce the possibility of error.
- **robust** – Software should resist failure, especially failure that can be anticipated, such as memory errors or network unavailability. Exception handling is a primary mechanism for building robustness into code, which enables a program to re-route around difficulties or fail gracefully.
- **resilient** – Software utilized by customers or processes making real-time transactions cannot afford a break in availability, and thus requires a system that immediately or within a specified time period will be operable again.
- **maintainable** – In many traditional software development environments, the developers initially writing the code may not be the same developers who must maintain the code once it is operational. In the end-user developer model, because developers often assume development, test, and operational roles, they directly benefit from designing perspicuous code that can be understood and modified with ease.
- **testable** – Customers should establish the degree of testing required on software and hopefully confirm that no software should be released for production without having been fully tested. Where automated testing is uncommon or impossible, as is largely the case in data analytic software such as SAS, manual tests with representative data as well as code peer review should be implemented.
- **secure** – Many software initiatives store and transform sensitive data that must be afforded a level of protection from intentional intrusion or accidental disclosure.
- **documented** – No one enjoys documentation and those fleeing the Waterfall in hopes of finding solace in a document-free Agile environment will be sorely disappointed. The second Agile value does embrace “working

software [aka, business value] over comprehensive documentation.” However, only through some modicum of documentation are user stories prioritized and recorded in both the product and sprint backlogs. Like Agile, documentation should be incremental, with a preference to document functionality that has been completed that is less likely to continue evolving in the flexible environment.

This does not represent an exhaustive list of non-functional requirements, but rather a starting point to facilitate discussion. Because the benefits of non-functional requirements often are observed only under unfavorable conditions, substantial testing should accompany their development to demonstrate their viability. The customer who receives an operable unit of software is pleased until he realizes that it fails when a critical data set is in use, produces invalid results when a file naming convention is violated, allows garbage input to enter the data store, and is documented so poorly that only the original developer is able to modify or attempt to rehabilitate the code. Thus, while functional requirements will remain the focus for both technical and non-technical product user stories, non-functional requirements should be addressed. To address, however, does not confer inclusion, and the customer must establish a standard of quality. In some situations, poor code will suffice, especially in straightforward, short-term, or single-use analytic development. For the vast majority of projects, however, non-functional requirements are critical to maintaining functionality and, just as Agile remains flexible, should flex to conform to the evolving scope of the project.

IMPLEMENTING AGILE

The path to Agile implementation has been eulogized in thousands of books, blogs, studies, articles, and symposia for over a decade. Developers (including end-user developers) who facsimile the Agile “developer” paradigm will find no problem applying this substantial body of knowledge and literature to their benefit, and likely already espouse many Agile values and principles. End-user developers and analysts whose work involve delivering business value *through* code, but who themselves do not deliver software as a product, will have less in common with this paradigm and will need to abstract Agile philosophy and principles into their environments. To the analyst who develops software during even a fraction of his day, however, Agile provides a viable methodology to reliably deliver value to the customer. Moreover, those same methods can be extrapolated without discomfort beyond the realm of software to facilitate and transform the development and delivery of analytic products. The following does not represent an implementation roadmap to Agile, but rather a few challenges and complexities to consider at the outset:

- In implementing and excelling with Agile, all is lost if all are not on board. While the facilitator will help coach and guide a customer and team of developers, his role does not empower him to force Agile on anyone. Thus, just as the customer and developers must believe *before the iteration begins* that all user stories in the sprint backlog will be delivered, all stakeholders also must commit firmly to Agile *before* embarking on the journey.
- Take Agile as a wife, not a lover. Development teams looking to try something new often have already tried something new—and failed at it. A pessimistic attitude or resistance to change may persist among developers or customers who believe that Agile is just the next ex-girlfriend who won’t be around by Christmas. These teams must demonstrate a commitment to Agile and proactively work throughout the sprint, especially collaborating during the retrospective to discover ways they can hone their implementation of Agile.
- Anticipate the environment to establish sprint duration. If fly-by tasks are the norm and no assignment or project ever lasts more than a month, then a 30-day sprint is going to lead only to frustration and failure. If projects endure for months, however, a month-long sprint may be appropriate. In principle, an iteration should be the length of time for which the scope of work being performed can be anticipated to be reasonably stable.
- The “make or buy decision” applies to personnel, not only software and products. Agile’s self-managing mentality and open communication quickly can elucidate developers ill-aligned with project and customer priorities. Developers who are not generalist-specialists and who want to be very selective in their personal scope of work (i.e., choice of user stories during each sprint) will cause friction and devalue themselves if they are not willing to learn additional skills to maintain relevancy to project priorities. In some environments, the team must make the tough decision between camaraderie and comparative advantage—they either invest time to train the teammates they know and like, or replace inflexible or inefficient developers with those more aligned to customer objectives.
- While pure Scrum stresses that working software (aka, business value) should be delivered during the first sprint, other methodologies and implementations of Agile have adopted a zero sprint that is reserved for planning and

project initialization activities. Teams should undertake this strategy with caution but, because of the amount of data modeling, literature review, research, and other preparatory activities required in analytic fields, this may be a necessity. If a zero sprint is required for planning and other non-developmental activities, teams must ensure they continue to plan in remaining iterations, embracing value-driven change and flexibility.

DOING AGILE

Doing Agile is significantly easier than implementing it, principally because all stakeholders have committed to the methodology. The customer is not (or only infrequently) making modifications to the sprint backlog, analysts are enjoying the freedom of organizing their days while producing quality work, and the facilitator increasingly is becoming comfortable in his new servant-leader role. Hurdles still will arise, and a few that are particularly common among analytic application of Agile include:

- Each developer should be able to anticipate how many pure development hours he will be able to devote during the current iteration. Thus, if a team of developers is responsible for building, operating, and maintaining an ETL infrastructure, the time dedicated toward operations and maintenance should be excluded from pure development hours. Anticipated leave, administrative requirements, and other non-developmental activities also should be excluded to ensure a reasonably accurate estimate. Developers thereafter can utilize this estimate during the sprint planning session to estimate the amount of work they will be able to accept and complete.
- Each developer should be able reasonably to assess how long it will take to complete each user story for which he is the lead. The developer who bemoaned having to estimate his development hours during the iteration—and there's always at least one—truly will balk at this requirement. In some cases, the user story needs to be decomposed further. In other cases, the developer should “spike” the user story, prompting a time-boxed period (e.g., two hours) to be devoted for the sole purpose of determining a realistic time estimate for user story completion. In prioritizing a user story for completion, the customer ultimately is buying a product, thus he must understand what he is purchasing and for what price (i.e., the number of hours) to be able to assess opportunity costs.
- Don't abandon daily stand-ups. While daily scrums are the lifeblood of the Scrum framework, not all Agile methodologies require their use. On analytic teams in which multiple simultaneous projects are assigned among members, a tendency to stovepipe can exist, especially along lines of job function or subject matter expertise. Thus team members largely may work on relatively individualized projects in serial fashion. Because Agile is self-organizing and self-managing, one analyst may begin a project, only to be assisted or replaced by a teammate midstream. Thus, for collaboration and in order to ensure project awareness by teammates, brief daily stand-ups should be required.
- Be prepared to be able to demonstrate your code or product to your customer. Although the big dog and pony show is during the user demonstration within the sprint review, communicative, hands-on customers are encouraged in Agile. Whether using formal version control methods or simply saving versions of work, developers should ensure they have something quick to demonstrate at a moment's notice. Telling the customer “The code was working, but now I'm debugging so it won't run.” is frustrating for everyone. Developers are advised to save the latest “good” copies of their code or product so, when the customer does approach, the developer can provide a successful yet succinct product or user-story demonstration.
- Never leave a customer with less functionality or capability than he had yesterday. Especially on data analytic projects, many user stories will be intertwined with or dependent upon each other. Developers should ensure that the completion of the current user story does not break a previously completed component or otherwise reduce business value in some other area. This is especially important at the end of an iteration, as a developer may attempt to demonstrate a previously completed component, only to find his team has broken it with a more recent modification. Because the iterative nature of Agile requires that all development phases occur within each iteration, sufficient testing and integration must occur before a user story meets the definition of done.
- Manage technical debt responsibly. Technical debt results when functional components are “completed” and accepted by a customer, but later are discovered and acknowledged to have a fault or be lacking functional or non-functional requirements. For example, code already may have been delivered by a developer but, in a subsequent iteration, a user realizes the code fails to produce valid data every Monday. The defect in the code—

which causes the failure—represents technical debt the team now has incurred. A natural tendency of many developers is immediately to “pay back the debt” and fix the code but, because all work must be customer-directed in Agile, a user story about the defect should be written and the customer must prioritize this work to be completed in a later iteration. Technical debt is a natural and unavoidable by-product of software development and data analysis but must be managed to avoid accumulation. In cases of poorly managed technical debt, a development team may find itself facing an entire sprint in which all or the majority of user stories represent technical debt from previously delivered work. At this point, the development team is no longer practicing Agile because they are only fixing past defects and are failing to produce new functionality during the sprint.

- Practice CYA—cover your analyst. Customers may be slow to prioritize the completion of work that involves technical debt or non-functional requirements because neither delivers new functionality. Non-functional components and patches critical to complex infrastructures have a way of rearing themselves in the most inopportune moments when their accumulation causes a cascading meltdown. Nevertheless, a constant clash may exist between developers clamoring to build infrastructure and customers prioritizing only new functional requirements. When open communication and facilitator imploring have failed, a document that clearly and succinctly illuminates each technical debt—and includes the most likely and worst case scenarios if left uncorrected, as well a succinct, proposed solution—not only facilitates further discussion, but also raises awareness that there is no easy-button in analytics. It is also always preferred for a development team to highlight defects in their own code or product, rather than wait for someone else to find them first.
- Developers, say “no” to your customer. Because Agile teams are self-managing, when a customer tasks a development team with new requirements during an iteration, he effectively is telling them he has reprioritized this new work over other prioritized requirements already being developed. In strict Scrum, this always is verboten and only by killing a sprint (not discussed in this text) can a customer prematurely terminate a sprint in order to enable an immediate reprioritization of new requirements. In reality, and often in more tactical data analytic environments, emergent requirements do appear requiring the customer to assign these immediately to the development team; in these cases the iteration should not be aborted. Instead, the development team must trade the new requirement for unworked requirements that have not been completed for that iteration. The customer can assist in deprioritizing the user stories that should be rejected and returned to the product backlog. In similar fashion, a customer who learns that a current user story is no longer necessary immediately should instruct the team to cease that work, at which point the team may be able to accept one or more additional user stories from the product backlog for completion in the current iteration.

CONCLUSION – BEING AGILE

A team implements Agile, then it does Agile but, most importantly, it needs to be agile. Yes, little agile. Methodologies, even those that are tried and true, can evolve from flexible philosophies into hardened, prescribed ceremonies. Some implementations of Agile within analytic and end-user development teams will mirror Agile or Scrum literature flawlessly, but most will not. In the introduction to her primer *Coaching Agile Teams*, Lyssa Adkins remarks “I realize that my agile may not be your agile, but I’m betting that the core concepts of all agile frameworks shine through in the way I talk about agile in this book.^{xxv}” For a team whose development work only occupies a fraction of its workflow, Agile still can be achieved on that fraction. And, for the majority of analytic teams whose development is only partially consumed with software production, but who are more steadfastly focused on research, analysis, data-driven decision making, and dissemination of analytic products, Agile methodologies successfully can be abstracted to these development processes as well.

REFERENCES

- ⁱ The Manifesto for Agile Software Development. The Agile Alliance. Retrieved from <http://www.agilealliance.org/the-alliance/the-agile-manifesto/>.
- ⁱⁱ The Twelve Principles of Agile Software. The Agile Alliance. Retrieved from <http://www.agilealliance.org/the-alliance/the-agile-manifesto/the-twelve-principles-of-agile-software/>.
- ⁱⁱⁱ Atlas, A. *Accidental Adoption: The Story of Scrum at Amazon.com*. 2009 Agile Conference. IEEE. August 2009, pp. 135-140.
- ^{iv} Chung, M. *Agile at Yahoo! From the Trenches*. 2009 Agile Conference. IEEE August 2009, pp. 113-118.
- ^v Fewell, J. *Marriott's Agile Turnaround*. 2009 Agile Conference. IEEE August 2009, pp. 219-222.
- ^{vi} Arthur, T. *Agile Adoption: Measuring its Worth*. SAS Institute, Inc. 2013. Retrieved from <http://support.sas.com/rnd/papers/2013/AgileAdoptionPaper.pdf>
- ^{vii} US Government Accountability Office. Report to the Subcommittee on Federal Financial Management, Government Information, Federal Services, and International Security, Committee on Homeland Security and Governmental Affairs United States Senate. *Software Development: Effective Practices and Federal Challenges in Applying Agile Methods*. GAO-12-681. July, 2012. Retrieved from <http://www.gao.gov/products/GAO-12-681>
- ^{viii} Broadus, W. *The Challenge of Being Agile in DoD*. Defense Acquisition, Technology, and Logistics January-February 2013.
- ^{ix} Fulgham, C., Johnson, J., Crandall, M., Jackson, L., Burrows, N. *The FBI Gets Agile*. Federal Bureau of Investigation. IT Pro, September/October 2011.
- ^x Thibodeau, P. *Budget Woes Force Feds to Embrace Agile*. Computer World. February 11, 2013.
- ^{xi} Thibodeau, P. *DHS Shifts to Cloud, Agile Development*. Computer World. April 8, 2013
- ^{xii} Project Management Institute. *A Guide to the Project Management Body of Knowledge (PMBOK® Guide) Fourth Edition*. PMI, Inc. 2008. p17.
- ^{xiii} International Institute of Business Analysis. *A Guide to the Business Analysis Body of Knowledge (BABOK® Guide) Version 2.0*. 2009. pp18-19.
- ^{xiv} Michele Sliger, Stacia Broderick. *The Software Project Manager's Bridge to Agility*. Addison-Wesley Professional. 2008. p38.
- ^{xv} Schwaber, Ken and Beedle, Mike. *Agile Software Development with Scrum*. Prentice Hall. 2002. p43.
- ^{xvi} Hughes, Troy Martin. 2014. *The Diverse End-User SAS Practitioner: Lessons Learned in Best Practices or a Recipe for Disaster?* South Central SAS Users Group (SCSUG) Educational Forum.
- ^{xvii} Russell Journey. *Agile Data Science*. 2013
- ^{xviii} Scott W. Ambler, Pramod J. Sadalage. *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional. 2006. p2.
- ^{xix} Merriam-Webster's Collegiate® Dictionary, Eleventh Edition.
- ^{xx} Cockburn, Alistair. *Agile Software Development: The Cooperative Game*. 2006. p27. Addison-Wesley Professional.

^{xxi} Retrieved from the Network of Excellence on End-User Development.
<http://giove.isti.cnr.it/projects/eud-net.htm>

^{xxii} Anders I. Mørch, Hege-Rene Hansen Asand, Sten R. Ludvigsen. *The Organization of End User Development in an Accounting Company*. pp108. in End User Computing Challenges and Technologies: Emerging Tools and Applications. Editor Steve Clark. 2007

^{xxiii} Dave Ingram. *Design – Build – Run: Applied Practices and Principles for Production-Ready Software Development*. Wrox. 2009.

^{xxiv} Ron Jeffries. *Extreme Programming Installed*. 2000. p4

^{xxv} Lyssa Adkins. *Coaching Agile Teams: A Companion for ScrumMasters, Agile Coaches, and Project Managers in Transition*. Addison-Wesley. 2010. pxx.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Name: Troy Martin Hughes
E-mail: troymartinhughes@gmail.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.