Simple Model for Agile Development

By Tony Alletag, PMI-ACP, PMP

Contributors:
Bill Faucette, PMP
Alex Hanson, Agile Software Engineer
Luke Lackrone, PMI-ACP, PMP
Joe Roberts, Agile Software Engineer

Agile is not a process but a mindset. Agile is a better way to write software and build systems focused on customer satisfaction. With leadership buy-in and proper implementation, an agile mindset will deliver quality quickly and at a reduced cost. When your team, project, or program agrees to do agile development, you are agreeing to the statements in the *Agile Manifesto*.

“Success with agile requires a team focused on rapid delivery of high-value features in an environment of constantly evolving technical and user requirements.”

The *Agile Manifesto* reads:

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.
(http://agilemanifesto.org/)

Many software development teams find themselves wanting to or directed to making the transition to agile but are not sure how to get started. The purpose of this paper is to generalize and simplify the roles, responsibilities, meetings, and iteration cycles so that an agile team can easily begin to implement their own agile process that works for their team, clients, customers, and users.

Agile Methodologies

Scrum, XP (Extreme Programming), FDD (Feature Driven Development), AUP (Agile Unified Development), and Kanban are some of the more advertised agile methodologies that have a vast amount of information available on them. The most popular and widely implemented agile methodology is Scrum. Terms such as ScrumMaster, Product Owner, Sprint Planning, and Daily Scrum are used by teams trying to implement an agile process without truly understanding or implementing Scrum. While the different agile methodologies can all have success, it is easier for teams new to agile to generalize the terms, roles, and meetings.
For the purpose of this paper, generic terms will be used for roles and meetings and to describe how they correspond to Scrum terminology.

Keep it Generic and Simple

The most successful agile teams don’t commit themselves to one specific agile methodology but tend to blend a combination of methodologies and adjust as needed.
There can be many advantages to not tying your software development project to one specific agile methodology. The

software engineers can focus on writing code and building a product and not on a process. The agile team can choose the aspects that complement the different personalities on the team, with the overall goal of producing the best product and satisfying the users and customer. Generalizing the agile process allows the agile team to build and deliver a product that is both usable and reliable to the users, without focusing on whether or not the team is unquestionably following Scrum, XP, FDD, or any other agile methodology.

A few key items for successful agile projects to follow are listed below:

1. Deliver working tested software frequently, from a couple of weeks to a couple of months
2. Welcome changing requirements, even late in development
3. Implement continuous integration
4. Have an automated unit test with a high percentage of code coverage
5. Trust team members to do their job and do it correctly
6. Have a simplified architecture and design to allow for rapid responses to change
7. Work off of a backlog and not off of function requirements documents
8. Are made up of motivated individuals responsible for meeting the deadlines they commit to and not deadlines that are forced upon them

For more information, refer to the 12 agile principles listed by the original signers of the *Agile Manifesto* at http://agilemanifesto.org/principles.html

Team Roles

Agile Team

Agile teams are self-organizing teams that include everyone involved with the design, development, test, and delivery of the finished product. The agile team's goal is to deliver a quality product that meets the needs of the users. Following are a few key roles that agile teams use; as teams mature and adapt many teams will develop unique roles, depending on their projects and the client user's needs.

Users (The most important role)

Users of the delivered products skills and responsibilities:

- Communicate what they want the delivered product to do and achieve
- Prioritize features and improvements
- Software and product acceptance
- Constantly provide feedback on features that are being developed
- Execute acceptance test
- Provide advice as the domain and subject matter experts
- Communicate the functional needs and requirements of the capability
- Being available to all of the agile team members for feedback

It is vital that a small group of primary users be identified to be the approval authority for the delivery of a capability. User contributions work best when a small group of two or three designated users are in charge of communicating needs to the agile team. These users should be leaders within the product users' community, decision makers, domain experts, and stakeholders. As new software becomes available in a development or test environment, users must be committed to viewing and providing feedback on features after every iteration, which will help the agile team set users' expectations for production deployments, thus minimizing the likelihood of the deployment being rejected by the user community.

Scrum assigns many of the user responsibilities to an internal development team member called the product owner. Product owners are charged with voicing the concerns of users of the product; however, if there are no users that are able to commit to the project by communicating with the agile team and taking ownership of the product, the agile team should revisit the business case on why this product is needed or being built.

Engineer

The engineer's skills and responsibilities are:

- Build trust and maintain user and customer relationships
- Provide system engineering support to the agile team
- Communicate with external projects and stakeholders
- Roadmap features and plan releases with the coordinator
- Maintain the product backlog
- Develop release, test, and deployment schedules
- Develop use cases for the application or new features
- System design architecture

The engineer is the project's main point of contact for new feature requests and maintaining a backlog and roadmap of when features will be implemented. The system engineer maintains and manages the project schedules, backlog, and requirements process. The engineer is also responsible for building and maintaining the user relationship. The only way the engineer can do this is to constantly communicate with the coordinator, and trust his or her estimates and technical implementations.

Scrum assigns a few of these responsibilities to the product owner; however, the product owner should not be the main voice for the users. Users need to be able to communicate feature priorities and enhancements to the developers, engineer, and the coordinator.

Coordinator

The coordinator's skills and responsibilities are:

- Project management
- Oversee all activities of the project
- Build trust and maintain customer relationship
- Manage the agile team's cycles and repetitions
- Plan iterations and meetings
- Participate in prioritization of the product backlog
- Team leader, mentor, and problem solver
- Identify and eliminate roadblocks
- Communicate and collaborate with all project resources
- Software engineering experience

The coordinator is the main point of contact for the project and is responsible for communicating with and collaborating with other projects, test teams, help desk, management, and performing status reviews. The coordinator should know everything that is going on with the project at all times and must attend all daily stand-ups to get a pulse check on what agile team members are doing. The person filling this role must have software engineering experience so that he or she can accurately understand and communicate the struggles and needs of the developers.

Scrum assigns a few of these responsibilities to a ScrumMaster. The term and role of the ScrumMaster can become troublesome for teams, especially those with internal competition. In many cases, the ScrumMaster is the team's most experienced developer with the most history and knowledge of the project. The most experienced developer on the team should not spend his or her time planning and hosting meetings but should be troubleshooting technical issues and writing code, where he or she will provide more value to the project. With self-organizing teams, a ScrumMaster role is not needed and usually one person is recognized as the team's technical mentor, whether it is a formal or informal team-appointed title.

Technical Mentor

The technical mentor's skills and responsibilities are:

- Career software engineer
- Provide overall technical guidance on the product being developed
- Technical expertise in the project's technology, programming language, and product
- Knowledge and understanding of the team's limitations and capabilities
- Motivate and coach the team
- Technical problem solver
- Work with the coordinator to ensure that project goals are clearly communicated to the project team and regularly measure progress toward such goals

The team's technical mentor is the go to person when any major bug or issue arises; he or she mentors new developers, enforces coding standards, ensures unit test coverage is adequate, and ensures the technical quality/success of the product delivered. The technical mentor and coordinator need to work closely together and be in synch. There must be constant, open communication between the technical mentor and the coordinator to ensure that each has full reliance on the other when working together to resolve any project issues that may arise.

Scrum assigns a few of these responsibilities to a ScrumMaster. In Scrum, the ScrumMaster's role will typically rotate around different developers. The role and responsibilities of the technical mentor surpass those of the ScrumMaster in that they allow the team's most experienced developer to write more code and mentor the development instead of acting as a project manager. The technical mentor's role does not rotate to other team members.

Developer (Includes Technical Mentor)

The developer's skills and responsibilities are:

- Write users stories based on the feature backlog
- Estimate the level of effort to implement user stories
- Write code to develop a releasable product at the end of every iteration
- Database design and development
- Implement test cases and unit test
- Collaborate at daily stand-up meetings
- Self-motivated and self-organized
- Deploy the application to all server environments

The developers consist of the full-time software engineers. Team members should aspire to perform the full spectrum of software engineering tasks, including: front-end code and view layer markup, back-end business logic and controller code, and database design and development. Agile developers are expected to use automated unit testing as part of their development process.

Scrum refers to the developers as the Scrum team. The responsibilities of a Scrum team are nearly identical to those of the developers.

Designer

The designer's skills and responsibilities include:

- Design a product that is appealing and easy for the users to use
- Ensure users are able to find relevant information and do what they want with the product
- Get feedback from real users and make design improvements
- Develop wireframes or primitive screen mockups
- Develop site maps and user flow diagrams

The designer could be one person or a team of people responsible for the overall interface, design, graphics, font types, usability, and appeal of the product. He or she needs to work with the developers to ensure design features can be implemented and are compatible with the product's technology without affecting the product's speed performance and response time.

Tester

The tester's skills and responsibilities are:

- Participate in the development of the user stories and acceptance criteria
- Assist with writing acceptance tests for user stories
- Functional verification and validation of capabilities and user stories
- Writing test cases/scenarios for acceptance tests
- Performing acceptance and exploratory tests during each software iteration
- Collaborate during iteration planning
- Documentation of test results and reporting of bugs

The tester is responsible for the verification and validation of the capability and user stories through both software integration tests and system integration tests. The tester can also assist with writing acceptance criteria and can provide any other testing assistance required by the agile team. The tester is expected to test users' stories daily as the development team completes them and proactively learn as much about the system as a typical user.

Iteration 0

Iterations form the heartbeat of the agile team. With project initiation and planning in Iteration 0, successive development in Iteration N and periodic but frequent software release in Iteration R (Figure 1).

Iteration 0 is performed at the beginning of a project and forms the framework upon which each subsequent iteration builds. The duration for Iteration 0 will vary depending on the complexity and existing knowledge of the desired capability. Iteration 0 starts when a new capability is required and resources are allocated. Tasks included in Iteration 0 include:

- Finish the staffing profile and assemble the team
- Create and document a project charter that defines the initial rhythm and heartbeat of the agile team
- Create the initial product backlog of user stories
- Create a project roadmap
- Define infrastructure requirements and where the project will be hosted, including development, demonstration, test, and production environments.
- Configure test and/or development environments
- Perform enough architecture and system design in order to start development and get the first Iteration N started. (Additional architecture and system engineering will be performed as necessary during subsequent iterations)

Agile focuses on the highest priority features, rather than spending time on planning, designing, and discussing capabilities that never get developed. This allows the agile team to write more code and deliver more capabilities to their customers. Agile teams use a product backlog that captures high-level feature requests for the next six to eight months and uses progressive elaboration as user stories and requests get closer to development.

Three things everyone needs to agree on for agile to be successful:

1. It is impossible to gather all requirements at the beginning of a project
2. Requirements gathered up front are almost guaranteed to change

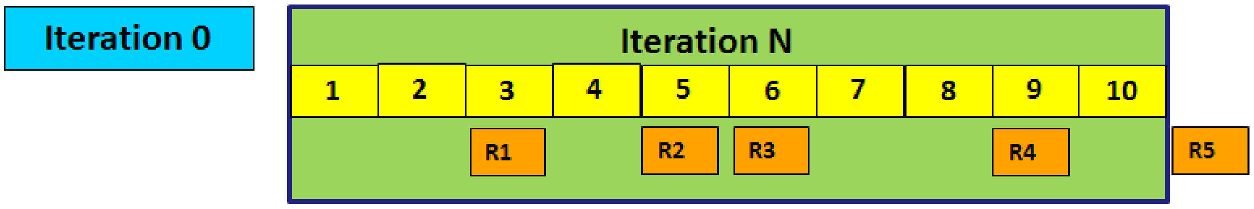


Figure 1: Iteration cycles.

3. There will always be more to do than time and money will allow

Successful agile projects:

1. Gather just enough user stories in Iteration 0 to build an initial capability
2. Constantly welcome change, evolve, and add new user stories to the backlog
3. Frequently reprioritize based on the highest value capabilities for users

Product Backlog

The product backlog contains the user highest priority request for the next six to eight months. A six to eight-month backlog allows agile teams to focus on near term releases without spending design cycles on features that change or are never developed. The backlog can be managed in an issue tracking system, as a top 10 list on a wiki or in a spreadsheet. Once a feature on the backlog is ready for development, a user story can be entered in an issue tracking system.

User Stories

Once a new feature request is ready to be developed, the team creates a user story that usually follows the following construct.

```
      As a <role>, I want <goal/desire> so
that <benefit>
```

Examples of User Stories:

```
      As a user, I want to search for items
      by name so that I can view the items'
      details and information.

      As a developer, I need to reduce
      my project's technical debt by
      refactoring the data ingest function
      so that the code base can continue to
      improve in quality and be easier to
      maintain.
```

User stories are capabilities with the associated acceptance criteria that the user has requested to be developed by the agile team. The agile team can internally verify this when the acceptance criteria is related to each of the user stories.

Examples of acceptance criteria:

- I can search for items by name
- I can view a list of search results
- I can click on an item to view additional details
- I cannot view items that have been removed from the inventory

Once all acceptance criteria pass, the user story is ready for formal testing. More information on user stories can be found at http://en.wikipedia.org/wiki/User_story and http://www.mountaingoatsoftware.com/topics/user-stories.

Iteration N

Iteration N commences at the conclusion of Iteration 0, and when the development team is ready to write code. Multiple Iteration Ns are grouped to form releases. One goal of agile is to have working deployable software at the end of every iteration; this does not mean that there is a production deployment after every iteration, rather the ability to deploy to a test or demo server for users to give feedback and to run through some of the acceptance criteria associated with the completed user stories.

Iteration N includes the following activities:

- Continuously reprioritize the product backlog
- Iteration planning:
  - Determine any changes in the capacity of the team due to developer availability
  - Estimate the size of the Iteration's user
  - Allow the agile team to determine who works on which user story
  - Decompose user stories to tasks. These tasks typically include: writing acceptance criteria, writing code, performing unit tests, and writing documentation
- Create the acceptance criteria for each user story
- Write the acceptance tests for each user story
- Perform an Iteration retrospective in order to continue to improve the project's processes and ensure the team repeats the successful practices

Meetings

Meetings are a necessity to communicate project needs, make informed decisions, and have technical discussions. However, the developers should not be required to attend management or status meetings unless a specific person's technical expertise is required. The developers should have as few meetings as possible, because every meeting the developers are in takes time away from writing code and producing the product. Automated reporting tools can be used to provide the status of the agile team.

Agile promotes communication and all team members are invited to attend all meetings as long as they are getting something of value out of the meeting or giving something of value to the other attendees. The required and optional attendees are expected to actively participate in the meetings listed below, whereas other attendees are expected to remain inactive participants during the meetings. The following section includes descriptions of recommended meetings, with the purpose of the meeting and the suggested attendees.

Backlog Prioritization

Expected Attendees: Users, Coordinator, Engineer
Optional Attendees: Technical Mentor, Designer
The backlog prioritization meeting's goal is to discuss current feature improvements and new features to be implemented in future iterations. These meetings should be scheduled at a minimum once a quarter but can be as regularly as once a month. Users are the main input for feature prioritization and these meetings should be scheduled at the users' convenience. In the meeting, the users must communicate the prioritization of which features they desire most and when they would like them. With coordination and input from the agile team the coordinator is responsible for giving reasonable estimates to features and a realistic timeline for when new features will be available. The coordinator must never make estimates the agile team cannot honor and should invite the team's technical mentor to discussions where more technical details are required. Roadmaps should be developed at a higher level than the feature backlog. Backlogs can be used to create a higher level roadmap for major feature release. Roadmaps are visual artifacts that can be used to communicate to executives major features that are planned to be released throughout the year.

Iteration Planning

Expected Attendees: Developers, Coordinator
Optional Attendee: Engineer
The Iteration planning meeting's goal is for the development team to take items/features from the product backlog, convert them into user stories, and commit to completing the task for the iteration. Typically, developers are not assigned tasks, but claim or commit to specific users stories. This will give them a greater sense of ownership of the task because they are not assigned, but personally committed to it. Specific iteration goals and priorities are agreed upon by the agile team with the objective of producing usable releasable software at the end of every iteration. Scrum refers to this meeting as a sprint planning meeting. As with sprints, iterations are typically two weeks in duration. Prior to the iteration planning meeting the agile team should go over the progress they made during the previous iteration. They need to take credit for any features or bugs that were completed, highlight any successes, and discuss areas for improvement.

Retrospectives

Expected Attendees: Developers, Engineer, Coordinator
Optional Attendees: Support teams (test, customer support, deployment team, designer)
Retrospectives are scheduled at the conclusion of an iteration or after a production release with all agile team members reflecting on the past iteration with the goal of continued process improvements. They are used to constantly improve the project's processes and making sure the team continues to do the things that make them successful. Retrospectives have an important role in iterative development and are needed for agile teams to learn from success and deficiencies. Retrospectives typically contain the following:

1. What should the agile team continue doing?
2. What should the agile team start doing?
3. What should the agile team stop doing?

Daily Stand-up

Expected Attendees: Developers, Coordinator
The daily stand-up occurs every work day at the same time regardless of who can and cannot attend the meeting. These meetings are intended to be 15 minutes in duration and consist of short quick overviews of each agile team's progress on the iteration. They are designed to answer three questions for each person:

1. What did I accomplish yesterday?
2. What do I plan on doing today?
3. What is stopping or slowing my progress?

Typically, a technical discussion will arise but any in-depth discussion should be held after the stand-up is completed or at another scheduled time. Short quick daily stand-ups are the key to making these meetings valuable to the entire team. If stand-ups start taking more than 30 minutes, team members become less productive and work begins to slip out of the iteration. Scrum's terminology for a daily stand-up meeting is the daily scrum.

Technical Exchange

Expected Attendees: Developers, Technical Mentor
Optional Attendees: Coordinator
The technical exchange meeting should occur at the same time and place every other week. It works best when this meeting is scheduled at the same time and place as the Iteration planning meeting but on the alternate week. The goal of this meeting is to give the developers time to discuss new feature implementation, review technical design, and inspect committed code. It also gives the developers a dedicated time to discuss any technical conflicts they may have.

Demonstrations

Expected Attendees: Developers, Technical Mentor, Users, Coordinator, Designer
Optional Attendees: Engineer

When an iteration ends, the developers must be given an opportunity to present what they completed. These demonstrations should include the users of the system to ensure that everything was completed to satisfaction and to discuss possible enhancements.

Pre-Release Meetings

Expected Attendees: Coordinator, Stakeholders
Optional Attendees: Developers, Technical Mentor, Engineer, Designer

The coordinator will need to schedule pre-release meetings prior to every release with the help desk or customer support team, the test team, and the users. The coordinator will need to be prepared to demonstrate any new features being deployed with the next release. This allows the test team to ask questions to ensure they are accurately testing new features. It will allow the customer support team to prepare training materials and documentation to prepare users for the version upgrade. The designated users must have access to a test or demonstration site so that they can provide feedback and user acceptance prior to the product being released. Writing functional quality code is not a trivial task; it is time intensive and requires periods of intense focus. To minimize the agile team's distractions, demonstrations should be scheduled at the end of an iteration and not on-demand.

Task Estimation

No estimation process is ever 100% accurate. Following is one way to estimate user story and task. First, the agile team must agree on what it means for a user story to be "done." Done is typically defined as a user story that has been unit tested, code developed, code checked into source control, passed all unit testing, and has passed its acceptance test criteria.

Agile teams should use a simple estimation strategy, with the goal of allowing teams to gauge velocity and make sure they are not over or under tasked. A user story that spans multiple iterations is considered an *epic*. When a user story is estimated to take longer than one iteration, it should be broken down into manageable pieces that are estimated to take less than one iteration. This allows the agile team to have concrete deliverables at the end of each iteration for user demonstrations or software version updates.

One task estimation strategy can be done by tracking the days each developer is available during that iteration and assigning a level of effort (1–5) to each task, where a level of effort of 1 is equal to 20% and 5 being equal to 100% of the available time the developer has during that iteration. To add up the team's velocity at the end of the iteration, add up all the level of efforts for each completed task at the end of the iteration. Only user stories that fit the definition of "done" are counted in that iteration's velocity.

Another slightly more complex method of task estimation involves using Fibonacci numbers for task estimation. When used correctly, this method of task estimation can give a better gauge of the team's average velocity; however, it may be difficult for teams starting to learn agile to adopt these estimating techniques. See additional information on Fibonacci estimates at http://simula.no/publications/Simula.simula.1282/simula_pdf_file

To ensure continuous code quality improvement, agile teams must sign up for, estimate and get management and users buy-in to dedicate development cycles to code refactoring user stories.

Iteration R

Iteration R's main goal is to deploy software to production environment(s). Many projects and organizations have specific and varied control gates in place prior to giving a project the approval to update software on a production site. Because of this, Iteration R is separated out from Iteration N so that the core development team can continue to develop code for the next release while the engineer and project coordinator can usher the latest product release through their deployment and release processes.

Iteration R includes the following activities:

- Finalize documentation
- Formalized testing, which may include:
    - Security testing
    - Performance testing
    - Integration testing
    - Full regression testing
- Deploying the new software version to the production environment
- Team celebrations

Best Practices Guidelines

Software Version Control

Every project, agile or not, uses some form of source code management (SCM) tool, such as Subversion or Git, where code is checked into a central repository. Developers will check into the trunk multiple times daily, as user stories and bugs are assigned and worked. A branch is then created off the trunk, where code may change but is not in the main baseline. Some projects may extensively use branches; others may do most of their work on the trunk. Finally, a tag is a fixed snapshot of the code that does not change; if a tag needs to change, a branch is created from that tag, the code is altered in the branch, and a new tag is created.

A summary of the versioning naming conventions, using the standard notation of major.minor.patch.iteration.build is indicated as shown in Table 1.

Table 1 only describes the minimum requirements for changing software versions. Even if existing interfaces do not break, substantial changes to the software may still necessitate a major release, based on discussions and feedback from the users.

With the major.minor.patch.iteration naming convention a fifth number can be used to indicate the build number. This works great when using automated build tools. These numbers exist as tags in the project's SCM tool, allowing for additional builds of a specific software release. The best practice to follow with the build numbers is to create a branch off the trunk when a version is completed in the trunk — in most cases, at the end of every iteration; then, use the branch to tag that version of the software. For example, if at the end of iteration 6 a project is releasing version 1.1.0, the following steps would be taken:

1. Create a branch off trunk, called <project name>-<version> (-1.1.0)
2. Create a tag off that branch, with the build number appended (-1.1.0.1)
3. Increment trunk to build the next version (2.0.0, 1.2.0, 1.1.1)

If a new build needed to be created due to a minor change in the branch, the following process would then be followed:

1. Check-in any changes to the branch
2. Run a full build to ensure automated unit tests do not fail
3. Create a new tag off the branch with an incremented build number (1.1.0.2)
4. Migrate changes back into the trunk, if appropriate

Projects may optionally have a fourth iteration number to indicate the current development cycle. This number continues to increment regardless of the major/minor/patch version currently being worked by the team, for example:

- 1.1.0.5 – Iteration 5, working toward version 1.1.0
- 1.1.0.6 – Iteration 6, working toward version 1.1.0
- 1.2.0.7 – Iteration 7, now working toward version 1.2.0

The end of Iteration 6, in the above example, would have been the software deployed to production as version 1.1.0. The branches get created for each iteration (<project name>-1.1.0.5), and then each tag appends a fifth number to indicate the build (<project name>-1.1.0.5.1). While verbose, the additional iteration number allows for projects to clearly branch and tag software associated with every development cycle.

Overall, this versioning scheme provides clear direction to the development teams as well as those performing configuration management (CM) activities as to the code associated with a particular version of the software.

Best practices for committing software includes:

- Make sure the local workspace matches the latest checked into SCM before doing a commit (most tools already do that)
- Compare changes with the latest checked in to SCM
- Keep commits small and logical
- All commits need to have comments describing what changed and the associated task
- Build and test affected modules before committing
- Ensure coding standards are followed (i.e., no commented code)
- Review other developer commits periodically

More information on software version control can be found at http://kevinpelgrims.com/blog/2012/07/05/version-control-best-practices and http://en.wikipedia.org/wiki/Trunk_%28software%29

Continuous Integration

Continuous integration (CI) is the practice of merging all distributed codes from all developers on the team into the trunk of the projects source control and building a testable

Major	Application Programmers Interface changes that are not compatible with existing interfaces
Minor	Additions to the API that do not break existing interfaces
Patch	No changes to the API – Improvements or bug fixes or performance
Iteration	Identifies the current development cycle
Build	Identifies a working software build

Table 1: Releases.

deployable product. CI should be run at a minimum once a day and, ideally, the entire baseline can be rebuilt and retested on every code commit. CI increases the quality of the software by reducing the defect escape ratio and decrease maintenance and sustainment costs. CI tools can and should be set up to notify the development team of any failed build or failed unit test. This early notification enables the development team to immediately resolve small integration issues early, instead of having a massive amount of large integration issues prior to a version release or deployment.

When complemented with an automated unit test, code quality inspection tools, and vulnerability scanning tools, CI becomes an even more powerful tool for the agile team. An automated unit test identifies problems early and prevents integration defects from piling up and become a risk to the release candidate or product delivery. Automated code quality inspections tools and vulnerability scanning tools can identify bad coding practices, code violations, code smells, and security violations without the need for another developer to review the code. CI increases code quality and reduces bugs in deployed software.

A 100% unit test is not a realistic goal. In many cases, code modules are simple enough or are even pre-generated to make unit tests excessive and not worth the benefit they provide; for example, the pre-generated Java code in Eclipse. Sixty percent automated test coverage is a good goal for agile teams to aim for, with 70% or more being exceptional. Teams and developers should use their own discretion in determining how much automated test coverage is required for a particular software module.

Guidelines for successful continuous integration:

- Automated builds should be executed at every commit or at least once daily
- Sixty percent unit test code coverage is ideal; 70% or more is exceptional
- Avoid excessive unit testing
- Automated notification to the enter development team when a build or unit test fails

Code Reviews

There are two primary benefits of code reviews. The first benefit is that any bug found during inspection is cheaper to fix (by orders of magnitude) than if it is found later in the process. The second benefit is that a team experienced in inspecting the code tends to be a team that is able to embrace the agile principle of collective code ownership. All of the code (and any bugs within) are the responsibility of the team, rather than a specific individual. This mindset helps mitigate staff turnover or conflict and results in a tighter, higher quality product.

Formalized code reviews during which a team sits in a meeting and goes line by line through code is an extremely wasteful practice. Agile teams utilize collaborative code reviewing tools such as Review Board that allow inline comments on code and provide instant feedback. Code reviews are used to identify and fix bugs and code smells overlooked by the initial developer. It also enables senior developers to mentor junior developers and teach them how to write better quality code.

Guidelines for Successful Code Reviews:

- No formalized code review meetings — Technical exchanges can be used to discuss code modules
- Work as a team — Authors and reviews need to work as a team; when a defect is found it can be used as a learning opportunity for the entire team.
- Review major features — The project's senior developers should review all new major features when code is committed.
- Request someone to review your code — Junior developers should assign code reviews to other developers if they are not confident in a commit or want a second set of eyes to review their work.
- Review small sets of code — Do not review more than 500 lines of code at one time
- Focus on high impact code — Reviews should be focused on core modules that most of the project depends on.

Frequent Deployments

Projects should deploy a build of their software after every development cycle. Two of the core tenants of agile teams are working software and rapid delivery of capabilities, and each development cycle should conclude with a working piece of software that can be deployed to support these goals.

Agile projects should, whenever possible, have four environments — Development, Software Test, Integration Test, and Production:

- Development — Development area, typically not accessible by users. The development environment should be flexible and resources can be allocated and reallocated quickly to support development efforts
- Software Test — Should be accessible by users and developers. Only stable baselines (tags) should be deployed to this environment. Used to test software deployment processes.
- Integration Test — System administrators maintain the system in this environment and the testers execute formal test procedures
- Production — Production supported with live data

Deployments to the development environment are flexible and should be changed based on the needs of the development team. Some teams may want to automate the deployment of every single build to the environment; some teams may want to have more fine-grained control; regardless, the development environment is flexible and is the responsibility of the developers to maintain.

Following each iteration a branch/tag should be made (see Software Version Control) of the working software developed during that development period. That software should be deployed to the software test environment for software-level testing and user feedback. Often, this environment will interface with other development or mock interfaces to external systems, but it will be representative of the actual deployment.

Code Refactoring

Continuous improvement in the quality of the code base is essential to allowing the agile team to develop features rapidly and continue to deliver a quality product. The technical mentor and coordinator must work together to get management and user buy-in to allow the developers to spend cycles on cleaning up the code base and refactoring code modules. Management will most likely ask why the developers would want to spend development cycles not producing any usable features to the user. An accounting system analogy can help communicate the need for refactoring.

As developers continue to develop features quickly, without full design documents, and adjusting them immediately after user feedback, they tend to build up technical debt. As that debt continues to grow iteration after iteration, developers will need to dedicate cycles to reducing that technical debt so that the quality of the code and delivered project does not significantly decrease. See additional information on code refactoring and technical debt at http://en.wikipedia.org/wiki/Code_refactoring and http://en.wikipedia.org/wiki/Technical_debt

Each project team will have its own threshold for the amount of technical debt they are willing take on but, eventually, all projects will have to spend time updating APIs, modularizing code, updating code libraries to the latest version, removing deprecated functions/features, and increasing automated unit test coverage.

Testing

Human Test

One of the most common ways to test an application is to have a person sit at a keyboard and run through a set of test script; however, this is not always the most efficient or effective way to test code or an application. It is recommended to have human testing of the application when doing a major deployment, a major code refactor, or any other high risk deployment. Projects looking to follow an agile mindset should focus on automated unit and automated feature testing as a way to ensure project quality and that their product is adequately tested.

Automated Test

Most teams have resource constraints when it comes to their test teams. Automated testing can reduce the need for a multiple tester when doing minor or quick two iteration deployments. Agile projects that have a high percentage of code coverage have less dependency for a human to manually run through a full set of regression test scripts. The developers will need to fully test any new features manually but should not have to depend on a human tester to approve their deployments. Allowing agile teams with high automated test coverage to deploy on demand will bring code and new features to users as needed.

Test Scope

The automated test, like all unit tests, should have a very clear focal point. Each test should examine a single expected behavior or property and have a corresponding test that ensures the proper results and logging for unexpected or failed behavior. Even with the integration test, the automated test should walk through a very specific user story and expect a very specific outcome.

Independent Test

The test must be independent of any other test, setup or teardown. Each test should set up all data required to run and nothing more. It should also remove any data it created upon completion of the test. Many automated testing frameworks build in this capability and are relatively reliable; however, situations can occur where they don't clear out a database between tests, resulting in false results. Periodically pause your tests in the middle of a test run and see if the database has more information than the current test required.

Test Implementation

Automated testing must be considered part of the software development cycle. The tests themselves are an integral part of the software being developed because they will help minimize time spent debugging and help the team identify and resolve issues before users do.

Prior to committing code, the code should be thoroughly covered through automated tests and those tests should be committed with the code. This will help team members ensure their work is compatible with the code being committed. The entire automated test suite should be run against any and every code change before it is committed to ensure that there are no conflicts with other areas of the project.

When a bug is found in the system, the developer committed to fixing the bug should follow the following steps:

1. First write a unit test that expects the specific failing behavior to not occur.
2. Run the test, which should fail because the bug will still be in the code.
3. The developer should fix the bug
4. Run the unit test to ensure the test now passes

This practice will ensure the bug will never be reintroduced into the system without being caught by the automated test suite.

Test as Documentation

Your automated tests should describe the functionality of every piece of the code base. This allows the test to function as documentation for the code. New team members should be able to read through the test suite and gain an understanding of how the code works. Many agile teams have new team members work exclusively on the test suite and bug fixes for the first few iterations, which allows them to become more familiar with the code base while still directly contributing to the project.

User Feedback or Acceptance Test

Successful agile projects regularly put new software in front of the users for immediate feedback. This does not require a deployment to a production server, but does require a demonstration or test server that users have access to, which allows them to try out new features that are still in development. Making this environment available, increases communication between the users and the agile team. If features are completely off track, the developers can start over and only lose one iteration worth of work instead of six months or more. This has the added benefit of showing users that, although you may not have had a deployment recently, you are making progress on the highest priority request.

Deployments

The goal of an iteration is to have working software that can be deployed once the iteration is completed. This does not mean that after every iteration a working product is deployed to a production environment; however, it does mean that at the end of every iteration the users should be able to see what was developed during that iteration either through a demonstration or by accessing a test site. This puts new features or bug fixes in front of the user at a rapid and consistent pace for immediate feedback to the agile team. Regardless of implementation, a repeatable automated deployment process enables consistent deployments and minimizes the chance for human error.

Agile teams assume the risk when doing quick minor deployment. The test team should not be required to run full regression testing on two-week deployments, because the developers can have confidence in their automated test script to ensure the quality of their code and deployed application.

Recommendation for Success

Give the developers the environment and permissions they need to get the job done

Developers should request access and permissions in the development, demonstration, test, and production environments. This will allow the developers to:

- Own the project and be responsible for any issues that may occur
- Do their own automated deployments — requiring less process, less documentation, less coordination with other teams
- Make configuration changes and bug fixes immediately, without unnecessary processes

The consequences of not giving developers permission to use servers include: quality developers leave the project to go work somewhere where they are given these permissions; teams will request servers or VMs outside of the organizations control to develop and deploy code to; teams will give users development or test server URLs to use as production because they won't want to deploy to servers that they can't log into and fix and troubleshoot issues.

Let the developers write code

Developers should not be required to attend every meeting and should only attend development focused or technical exchange meetings.

- Developers enjoy writing code, not attending meetings.
- Management should not be able to require a developer to attend a meeting; they should only be able to require the attendance of a technical mentor, coordinator, or engineer.

The consequences of developers attending to many meetings include: developers not spending time writing code, features slip schedule, and user frustration. The coordinator should be able to decide if a developer needs to attend a meeting or not.

Dynamic Documentation

- Usually, formal documentation is out of date. Developers regularly document critical project information on a wiki, which may contain FAQs, a user's guide, operating instructions, or information the project team deems necessary.
- Informal dynamic documentation is more likely to be updated by the project team, which would increase the chance of the most accurate and current information being presented to the readers
- A standard read-me file included in the code is usually sufficient for deployment instructions.

The consequences of having too much formal documentation includes: documentation not updated, and developers keeping their own set of technical documentation internal to their team.

Conclusion

An agile mindset will evolve differently in every agile team. Agile development is based on team agreements, not set processes. When starting out with agile development, teams should try to generalize the agile mindset, team roles, and meetings. Agile teams should start out slow and iterative, and should not try to implement every meeting or have a lead role for everyone on the team just to say they are doing Scrum or XP. The coordinator and technical mentor should decide what they think is needed and see how the agile team evolves and adapts. If communication does not flow between developers, a solution may be to implement a bi-weekly technical discussion. If unit tests are not being completed, remember to stress the importance of unit test in daily stand-ups and feature levels of effort estimates. The focus should be on deliverables, not process, tools, and meetings. This will allow the team to continuously adjust according to the different personalities, client demands, and user priorities.

Success with agile requires a team focused on rapid delivery of high-value features in an environment of constantly evolving technical and user requirements. These suggestions allow the team to successfully deploy proactive user-ready software features in dynamic user environments. Remember: when your team, project, or program agrees to do agile development, you are agreeing to the statements in the *Agile Manifesto*.

The *Agile Manifesto* reads:

> 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.
> (http://agilemanifesto.org/)

Agile is not a process but a mindset. Agile is a better way to write software and build systems focused on customer satisfaction. With leadership buy-in and proper implementation, an agile mindset will deliver quality quickly at a reduced cost.

About the Author

Tony Alletag's professional experiences include a diversified portfolio of agile software development projects spanning across three government intelligence agencies and consisting of cyber security solutions, intelligence dissemination, web portals, financial reporting systems, and satellite modeling and simulation.

At the time of this writing, Tony enjoys supporting Booz Allen Hamilton's clients, where he is the Agile Software Development Manager for a program with over 300 staff members and 40 software development projects. He enjoys mentoring, coaching, and training the program's software engineers in best practices and agile project management, while successfully transitioning legacy projects from waterfall to an agile mindset.

Prior to working at Booz Allen, Tony was an Agile Project Lead at Lockheed Martin for over 20 software engineers tasked with developing, deploying, and supporting enterprise IT solutions for government clients.

Appendix

Your First Iterations

Below is a sample of what a team should be accomplishing in the first few iterations.

Iteration 0 — Two weeks to a month

- Identify the users and stakeholders
- Get any necessary papers signed, such as team agreements and access permission
- Have a team chartering session; discuss expectations; define "Done"
- Start talking about an overall architecture approach
- Identify some likely technologies; perform analysis of alternatives
- Prepare the team space – make sure the physical space is conducive to collaboration and productivity; set up the information radiators
- Get your environment set up (see the Tools section); does anyone need training on version control or the build system?
- Get a few user stories into the backlog
- Communicate the output of Iteration 0 to the teams and stakeholders

Iteration 1 — Two weeks

- Hold daily standups
- Be sure the backlog is estimated
- Work with the users, coordinator, and engineer to establish an iteration backlog
- Focus on identifying end-to-end functionality that can offer new (atomic) value when delivered
- Write code; test; get user stories "Done"
- Hold a demo — at this stage, you may just have some wireframes and/or screen mockups to obtain client feedback
- Hold a retrospective and make changes to improve

Iteration 2 — Two weeks

- Hold daily standups
- Estimate new backlog items
- Hold iteration planning to establish the iteration backlog
- Write code; test; get user stories "Done"
- Hold a demo; obtain client feedback
- Measure your velocity; how do you feel about the team's productivity?
- Hold a retrospective; is there anything to improve?

Iteration 3 and beyond

- Keep going – inspect and adapt!

Tools

Following is a list of common tools (Table 2) that can be used to assist with automation, communication, and work tracking. Although tools are great, tools don't get the work done, people do. Agile teams pick tools that make sense for their projects and that their development team is familiar with and comfortable using.

Tool	Free	Additional Information
Agile Ticket Tracking		
Trac	X	http://trac.edgewall.org/
Redmine	X	http://www.redmine.org/
JIRA		https://jira.atlassian.com/secure/Dashboard.jspa
GreenHopper		http://www.atlassian.com/software/greenhopper/overview
FogBugz		http://www.fogcreek.com/fogbugz/
VersionOne		http://www.versionone.com/
Serena Agile Planner		http://www.serena.com/index.php/en/products/agile-planner/
Continuous Integration and Deployment		
Jenkins	X	http://jenkins-ci.org/
Hudson	X	http://hudson-ci.org/
CruiseControl	X	http://cruisecontrol.sourceforge.net/
Continuum	X	http://continuum.apache.org/
Bamboo		http://www.atlassian.com/software/bamboo/overview
RPM's	X	http://en.wikipedia.org/wiki/RPM_Package_Manager
Capistrano	X	https://github.com/capistrano
Maven	X	http://maven.apache.org/
Code Quality and Vulnerability Inspection		
Sonar	X	http://www.sonarsource.org/
FindBugs	X	http://findbugs.sourceforge.net/
IBM Appscan		http://www-01.ibm.com/software/awdtools/appscan/
CAST		http://www.castsoftware.com/discover-cast
Retina		http://www.beyondtrust.com/Products/ RetinaNetworkSecurityScanner/
Automated Testing		
Selenium	X	http://docs.seleniumhq.org/
Robotframework	X	http://code.google.com/p/robotframework/
Robot		http://www.ibm.com/developerworks/rational/products/robot/
Cucumber	X	http://cukes.info/
Collaborative Code Reviews		
Review Board	X	http://www.reviewboard.org/
Collaborator		http://smartbear.com/products/software-development/code-review
FishEye/Crucible		http://www.atlassian.com/software/crucible/overview
Gerrit	X	http://code.google.com/p/gerrit/
Kiln		http://www.fogcreek.com/kiln/
All In One		
Team Foundation Server		http://msdn.microsoft.com/en-us/vstudio/ff637362.aspx
Atlassian		http://www.atlassian.com/
Rally		http://www.rallydev.com/
Rational Jazz		http://www-01.ibm.com/software/rational/jazz/

Table 2: Agile tools.

Common Terminology

A common lexicon is import for proper communication between all team members. A sample list of common terminology that an agile team may use is listed below.

Bug — An error, flaw, mistake, failure, or fault in a computer program or system that produces an incorrect or unexpected result or causes it to behave in unintended ways.
Capability — Agile development teams build toward the highest priority request from users; they do not work of off requirements documents, such as functional requirements documents or systems requirement specifications.
Daily Stand-up — Daily team meeting held to provide a status update to the team members.
Done — A user story is complete when it is successfully unit tested, code developed, code checked in to source control, and has passed its acceptance test criteria.
Epic — Large user stories too big to implement in a single iteration, which need to be broken down into smaller user stories at some point.
Iteration — A development cycle, typically two weeks in duration, during which development occurs on a set of backlog items that the team has committed to. One or more iterations make up a production software release.
Iteration Planning — A meeting that occurs at the beginning of every iteration cycle, during which the team signs up for and commits to completing user stories
Product Backlog — A list of features and request for work on a project containing short descriptions; these features and requests are ordered in priority with more time and details added to the features at the top of the list.
Release — Production software version upgrade; releases contain one or more iterations.
Retrospective — At the end of every iteration, a retrospective is held to look for ways to improve the process for the next iteration.
Roadmap — Visual artifacts that can be used to communicate planned major capabilities to senior management.
User Story — An agile requirements approach that helps shift the focus from writing about requirements to talking about them. All agile user stories include a written sentence or two and, more importantly, a series of conversations about the desired functionality.
Velocity — Number of user story points completed in an iteration.
Version Code Freeze – The trunk of a repository should not typically freeze, a version freeze occurs when a tag is made for a version to be tested or deployed to production.

Agile Maturity Checklist

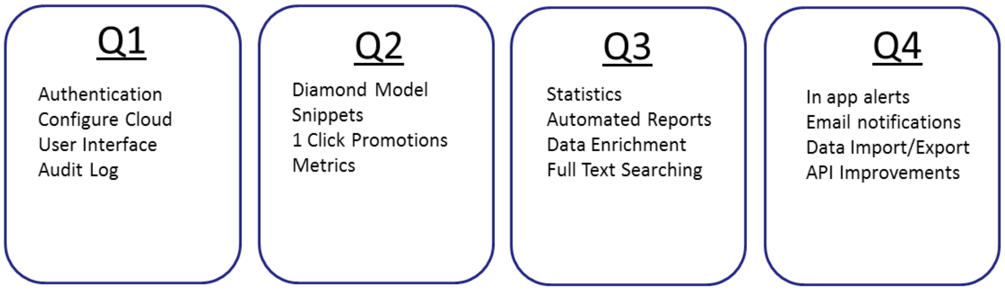
This checklist can be used to see if you team is embracing agile tools and techniques and help identify areas for improvement. The Santeon Group provides an excellent online agile assessment with Dr. Agile at http://www.dragile.com/

	Best Practices		**Best Practices**
	Unit Testing		Pair Programming
	Continuous Integration		User Stories with Acceptance Criteria
	Defined Coding Standards		Dedicated Users
	Definition of Done		
	Automated Builds		
	Ability to Reduce Technical Debt		**Meetings**
	Velocity Measures		Daily Standup
	Working software after an Iteration		Iteration Planning
	Use of Burndown/Burnup Charts		Backlog Prioritization with Users
	Work Committed to not Assigned		Roadmapping Sessions
	Those Doing the Work Provide Estimates		Release Planning
	Use of a Prioritized Backlog		Retrospectives
	Feature request maintained in a Backlog not in System Requirements Documents		Demonstrations
	Collective Code ownership		Technical Exchanges

Table 3: Agile checklist.

Sample Roadmap

Below is an example of a simple roadmap that visually conveys what the project is expecting to complete over the next year by quarter.



Sample Schedule

Development Iterations should be scheduled for two weeks as per the sample schedule above; for example, starting on a Wednesday and ending on the second Tuesday. Friday and Monday are typically avoided for iteration planning meetings, because extended weekends and holidays tend to disrupt the agile team's rhythm and development cycle. Releases should be packaged with specific features so that test cycles and deployments can be separated from development iterations. Decoupling development from test cycles and deployment will facilitate schedule adjustments when features take longer than estimated, by allowing development iterations to be added to that release; it will also allow for more flexibility in the test cycles and deployment schedules. If there are more complex features, or a complex refactoring in a release, the team should plan on having a few iterations prior to scheduling test cycles and the deployment of that release. The testing period can be lengthened for major version updates or extensive refactoring of software. Deployment cycles can be adjusted based on internal control gates. This allows the developers to continue to focus on writing code and developing new features, while the coordinator can push the version update through the processes external to development. A repetitive schedule encourages the entire team to develop good habits to keep everyone in synch.

Release 0.1 Development (Feature X,Y,Z)			
	Iteration 1	2 Weeks	Start on Wed end on the next Tues
	Iteration 2		Start on Wed end on the next Tues
	Iteration 3		Start on Wed end on the next Tues
Release 0.1 Deployment (Feature X,Y,Z)			
	Test	1 week	Starts Wed after Iteration 3 ends
	Deploy		After testing and approval for deployment
Release 0.2 Development (Feature A,B,C)			
	Iteration 4		Major bugs, or an immediate request for a capability, modify the release to contain 1 iteration
Release 0.2 Deployment (Feature A,B,C)			
	Test	1-2 day	Shortened test period for small release
	Deploy		Deploy as soon as testing is completed
Release 1.0 Development (Feature E,F,G,H)	Iteration 5		
	Iteration 6		
	Iteration 7		
	Iteration 8		
Release 1.0 deployment (Features E,F,G,H)			
	Test	2 weeks	Major refactoring or software version upgrades require additional test time
	Deploy		

Table 4: Sample schedule.