|  |
| --- |
| Agile Development Guidelines  Aligned to EDGE Agile 2.0 |

This document contains consolidated lessons learned and guidelines for agile development teams based on past experience. All guidance should be taken in context and may not apply to your project. The document is organized in sequence to match the EDGE 2.0 Agile Development model. The brackets indicate the source of each item (references are at the bottom of the document).

Modification History

|  |  |  |  |
| --- | --- | --- | --- |
| Revision | Date | Person | Description |
| 1.0 | 17-Dec-2009 | Terry Healey | Final Draft for inclusion with EDGE R1.2 |
| 1.1 | 10-June-2010 | Prabhu Eshwarla/ Jim Hughes | * Guidelines for location considerations for testers in agile projects * Additional guidance for test automation * Separation of Black-box(behavioral) unit testing from White-box (code-based) unit testing * Guidance for tester involvement in user stories * Misc. formatting updates |

**Note: Template Revision History can be found in Hidden Text.**

Template Revision History (Hidden Text)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Change Request# | Template Version | Approval Date | Modified By | Section, Page(s) and Text Revised |
| 309233 | 1.1 | 2011-2-15 | EDGE Change Team | Change format to HP Standard |
| 212187 | 1.3 | 2009-10-02 | EDGE Change Team | Updated to reflect re-branding to HP Enterprise Services |
| 185564 | 1.2 | 2009-08-17 | EDGE Change Team | Asset updated to ensure formatting is consistent when paper size is changed between Letter & A4. |
| 145300 | 1.1 | 2009-06-05 | EDGE Change Team | Security Classification added to Title Page. Legal statement updated in footer. Corrected header/ footer formatting issues. Added ‘Project Name’ and ‘Document Status’ fields under revision history. |
| 90181 | 1.0 | 2009-04-24 | EDGE Change Team | EDGE MS Word Template created |

Contents

[**Contents 1**](#_Toc263854975)

[**Initiate Project 4**](#_Toc263854976)

[Register Potential Project 4](#_Toc263854977)

[*Choose an Appropriate Methodology and Contract Model 4*](#_Toc263854978)

[*Staff and Assign Roles With Agile in Mind 4*](#_Toc263854979)

[*Identify an Appropriate Product Owner 5*](#_Toc263854980)

[*Obtain an Agile Coach 5*](#_Toc263854981)

[*Commit to Doing Agile (Not Just ‘Trying’) 5*](#_Toc263854982)

[Define Project Charter 5](#_Toc263854983)

[*Ensure Organizational and Client Support for Agile 5*](#_Toc263854984)

[*Craft a Compelling Product Vision 6*](#_Toc263854985)

[*Identify Stakeholders and Dependencies Early 6*](#_Toc263854986)

[Define Initial Project Scope 6](#_Toc263854987)

[*Identify Business Benefits & Measures Early 6*](#_Toc263854988)

[Obtain Project Approval 6](#_Toc263854989)

[Establish Project Environment 7](#_Toc263854990)

[*Consider Agile Project Management Tooling 7*](#_Toc263854991)

[*Automate Wisely 7*](#_Toc263854992)

[*Assess Agile Training Needs 7*](#_Toc263854993)

[*Procure a Project-Owned Agile Infrastructure 7*](#_Toc263854994)

[*Ensure Adequate Communication and Collaboration Infrastructure 7*](#_Toc263854995)

[*Involve the Whole Team in Planning and Communication 8*](#_Toc263854996)

[*Perform Key Startup Activities Prior to First Iteration 8*](#_Toc263854997)

[**Perform Iteration Zero 8**](#_Toc263854998)

[Define Team Organization 8](#_Toc263854999)

[*Context is Key 8*](#_Toc263855000)

[*Implement Practices Incrementally 8*](#_Toc263855001)

[*Define “Done” 8*](#_Toc263855002)

[*Decide How Requirements Will Be Documented 9*](#_Toc263855003)

[*Prefer Simplicity (Less is More) 9*](#_Toc263855004)

[*Collaborate as “One Team” 9*](#_Toc263855005)

[*Define and Evolve Team Standards and Rules 9*](#_Toc263855006)

[Initialize Product Backlog List 10](#_Toc263855007)

[*Utilize a Single Product Backlog 10*](#_Toc263855008)

[*Compose the Product Backlog at Appropriate Detail 10*](#_Toc263855009)

[Perform Technical Launch 10](#_Toc263855010)

[*Perform Oversight Reviews in an Agile Context 10*](#_Toc263855011)

[Create Initial Release Plan 10](#_Toc263855012)

[*Employ “Two-Level” Planning 10*](#_Toc263855013)

[*Create Lightweight, Adaptable Release Plans 10*](#_Toc263855014)

[*Deliver Early and Often 11*](#_Toc263855015)

[*Create Timeboxed (Rather Than Scopeboxed) Release Plans 11*](#_Toc263855016)

[*Build Release Plans Around Minimum Marketable Features (MMF) 11*](#_Toc263855017)

[*Write Releasable User Stories 11*](#_Toc263855018)

[Develop Project Testing Strategy 11](#_Toc263855019)

[*Integrate Testing into the Process (vs. at the End) 11*](#_Toc263855020)

[*Automate Testing as Much as Possible 12*](#_Toc263855021)

[*Integrate Automated Tests With the Continuous Integration Build Environment 12*](#_Toc263855022)

[*Location Considerations for Agile testers 12*](#_Toc263855023)

[Define Data Strategy 13](#_Toc263855024)

[*Align the Data (Engineering) Strategy with an Iterative/Agile Approach 13*](#_Toc263855025)

[Define Project Architecture 13](#_Toc263855026)

[*Identify “Architecturally Significant” Stories Early 13*](#_Toc263855027)

[*Identify Architecture/Design Decisions that can be Deferred 13*](#_Toc263855028)

[*Avoid Speculation; Validate Through Delivery 13*](#_Toc263855029)

[Produce Initial Size Estimates 13](#_Toc263855030)

[*Estimate Agile Size in Story Points or Use Case Points 13*](#_Toc263855031)

[Conduct Technical Review 14](#_Toc263855032)

[*Negotiate Who/What/When for Technical Reviews 14*](#_Toc263855033)

[Validate Solution Design and Strategy 14](#_Toc263855034)

[**Plan Project Work 14**](#_Toc263855035)

[*Plan Appropriately 14*](#_Toc263855036)

[*Be Confident 14*](#_Toc263855037)

[Define Project Process – Interactive 15](#_Toc263855038)

[*Be Careful With Tailoring 15*](#_Toc263855039)

[Prepare Plan Components - Agile 15](#_Toc263855040)

[*Avoid “Two Sets of Books” 15*](#_Toc263855041)

[Estimate Project Costs – Agile 15](#_Toc263855042)

[Create Project Schedule 15](#_Toc263855043)

[*Coordinate Project Schedules with Other Projects 15*](#_Toc263855044)

[Package and Secure Commitment 16](#_Toc263855045)

[**Execute Project Plan 16**](#_Toc263855046)

[Deploy Project Resources 16](#_Toc263855047)

[*Co-locate the Team or Compensate for Distribution 16*](#_Toc263855048)

[*Organize Around Features—Not Components 16*](#_Toc263855049)

[*Organize Around Small, Self-Sufficient, Co-located, Feature-Based Teams 16*](#_Toc263855050)

[*People are Not “Resources” 16*](#_Toc263855051)

[Manage Plan Execution 16](#_Toc263855052)

[*Foster an Enabling Culture 16*](#_Toc263855053)

[*Set and Expect a Sustainable Pace 17*](#_Toc263855054)

[Manage Project Results 17](#_Toc263855055)

[*Use Appropriate Metrics to Assess Agile Project Progress 17*](#_Toc263855056)

[*Be Wary of Misused Metrics 17*](#_Toc263855057)

[Manage Project Exceptions 18](#_Toc263855058)

[*Do Not Move Milestone Dates 18*](#_Toc263855059)

[*Be Transparent (Do Not Ignore “Bad News”) 18*](#_Toc263855060)

[Manage Intellectual Property 18](#_Toc263855061)

[Manage Requests for Change 18](#_Toc263855062)

[*Change Control Still Applies to Agile 18*](#_Toc263855063)

[*Embrace Change 19*](#_Toc263855064)

[**Sprint N 19**](#_Toc263855065)

[Plan Sprint 19](#_Toc263855066)

[*Plan the Iteration as a Team 19*](#_Toc263855067)

[*Choose an Appropriate Iteration Length 19*](#_Toc263855068)

[*Keep Iteration Length Consistent Throughout the Release Cycle 19*](#_Toc263855069)

[*Synchronize Sprints Across Teams 19*](#_Toc263855070)

[*Consider Utilizing Inter-Sprint Breaks 19*](#_Toc263855071)

[*Consume Lessons Learned 20*](#_Toc263855072)

[*Promote Team Learning 20*](#_Toc263855073)

[*Determine Team Velocity 20*](#_Toc263855074)

[*Define Goals and Select Features 20*](#_Toc263855075)

[*Avoid the Temptation to Over- or Under- Commit 20*](#_Toc263855076)

[*Limit Up-Front Design, but Address Technical Risk Early 20*](#_Toc263855077)

[*Decompose Selected Features into Tasks 20*](#_Toc263855078)

[*Estimate Tasks 20*](#_Toc263855079)

[*Finalize Sprint Backlog 20*](#_Toc263855080)

[*Commit to Tasks 20*](#_Toc263855081)

[Manage Sprint 21](#_Toc263855082)

[*Facilitate Daily Scrum Meeting 21*](#_Toc263855083)

[*Manage Issues 21*](#_Toc263855084)

[*Track Sprint Status 21*](#_Toc263855085)

[*Use a Burndown Chart to Track the Iteration 21*](#_Toc263855086)

[*Prepare for Future Sprints 21*](#_Toc263855087)

[*Communicate Sprint Status 21*](#_Toc263855088)

[Develop Features 21](#_Toc263855089)

[*Avoid Multi-Tasking and Limit WIP 21*](#_Toc263855090)

[*Tester Involvement in User Story Specification\*\*\* 22*](#_Toc263855091)

[*Travel Light 22*](#_Toc263855092)

[*Minimize/Eliminate “Go-Betweens” 23*](#_Toc263855093)

[*Apply appropriate Engineering Practices to maximize quality 23*](#_Toc263855094)

[*Design Feature 23*](#_Toc263855095)

[*Design for Testability and Loose Coupling 23*](#_Toc263855096)

[*Parameterize for Agility 23*](#_Toc263855097)

[*Refactor Mercilessly, Use Design Patterns Judiciously 23*](#_Toc263855098)

[*Design and Develop Test Scenario 23*](#_Toc263855099)

[*Specify by Example 23*](#_Toc263855100)

[*Create Executable Test Plans 23*](#_Toc263855101)

[*Develop the Feature 23*](#_Toc263855102)

[*Perform Agile Unit Testing 24*](#_Toc263855103)

[*Automate Unit Tests in Parallel with Development 24*](#_Toc263855104)

[*Even Legacy Code Should Have Automated Unit Tests 24*](#_Toc263855105)

[*Check-in and Integrate Application Components 24*](#_Toc263855106)

[*Fix Broken Builds Immediately 24*](#_Toc263855107)

[*Conduct Client Review 24*](#_Toc263855108)

[**Test Application 24**](#_Toc263855109)

[Perform Integration Testing 24](#_Toc263855110)

[*Automate Integration Tests Within a Continuous Build Environment 24*](#_Toc263855111)

[Perform System Level Testing 24](#_Toc263855112)

[Optimize Application Performance 24](#_Toc263855113)

[Review Application with Client 24](#_Toc263855114)

[**Commit Work 24**](#_Toc263855115)

[Facilitate Defect Causal Analysis 24](#_Toc263855116)

[Conduct Review/Demo 25](#_Toc263855117)

[*Keep Sprint Review Informal 25*](#_Toc263855118)

[Finalize the Sprint 25](#_Toc263855119)

[*Demonstrate Progress Using a Burn-Up Chart of Completed Stories 25*](#_Toc263855120)

[*Do Not Count Work Not “Done” 25*](#_Toc263855121)

[Conduct Technical Reviews 25](#_Toc263855122)

[Conduct Retrospective 25](#_Toc263855123)

[*It’s More Than Just Practices 25*](#_Toc263855124)

[*Conduct a Facilitated Full-Team Retrospective after EVERY Sprint 25*](#_Toc263855125)

[**Release System 26**](#_Toc263855126)

[**Deploy System 26**](#_Toc263855127)

[**Close Down Project 26**](#_Toc263855128)

[**Additional Guidance 26**](#_Toc263855129)

[Governance and Compliance 26](#_Toc263855130)

[*Agile Projects Will Affect Governance 26*](#_Toc263855131)

[*Agile Projects Have a Different Model of Change 27*](#_Toc263855132)

[*Agile Projects Have a Different Measure of Success 27*](#_Toc263855133)

[Distributed Teams 27](#_Toc263855134)

[Documentation 27](#_Toc263855135)

[*Agile Manifesto: Working software over comprehensive documentation 27*](#_Toc263855136)

[**References 28**](#_Toc263855137)

Guidance for EDGE Agile/Scrum Work type (draft)

Initiate Project

Register Potential Project

#### *Choose an Appropriate Methodology and Contract Model*

1. Waterfall, Iterative and Agile have their home turf. In general, *Iterative* should be the default development method (not waterfall). Use *Agile* when all/most of the “must haves” are in place (see below) and the project would benefit from Agile’s greater quality, productivity, and/or responsiveness to change. [4i]
2. There are several “must haves” that apply when considering agile (involved client, team skill mix, etc.). Go through the AWIMA Method Advisor as soon as possible to ensure that agile is a good fit for the project. Ideally, enlist an agile coach before committing to an unrealistic approach. [4i]
3. Neither Iterative nor Agile will “fix” a project that was sold fixed price with unrealistic dates, features or cost. This is why appropriate Sales and Solutioning up-front is vital. [4i]
4. Do not equate "iterative" with "agile". They are two different (but related) concepts. As a rule, all agile projects are iterative, but not all iterative projects are agile. [4i]
5. Be wary of falling into the trap of performing a series of mini-waterfalls. Agile ≠ Waterfall + Iterations. This is often referred to as Iterative Waterfall. [CC]

#### *Staff and Assign Roles With Agile in Mind*

1. It is a fallacy that agile requires only top-notch developers on the team to be successful. However, it is true that a team should have a large number (>25%) of senior people. Mix the team with skill levels and ensure you have enough senior people to mentor junior people. [4i]
2. Ensure that those being enlisted to join the agile project are aware of and supportive of (or at least open to) agile values and principles in the context of the project. In particular, specialists must be willing to suspend some of their “ownership” of project artifacts (e.g., stories, design models, test specifications, code) for the greater purpose of self-organization, collective ownership, and total team productivity. [TH, SB]
3. Minimize the number of people assigned to the project who have extra-project responsibilities. Pure Scrum demands that everyone on the project is 100% dedicated to the Sprint and cannot be pulled off to other work without the Sprint being cancelled. Some teams use a 60% rule—everyone on the team must be at least 60% allocated to the team. [TH]

#### *Identify an Appropriate Product Owner*

1. The Product Owner (PO) is a critical Scrum role and should be identified as early as possible in the project to ensure they can guide the project successfully. The Product Owner must be:
2. Available – assigned to the duration of project as his/her primary activity and responsibility
3. Experienced/Business-Savvy – Knowledgeable about the existing system and/or the desired functionality
4. Engaged/Communicative – Committed to working directly with the team daily to clarify requirements, prioritization, and product vision
5. Empowered – Authorized, responsible, and accountable to serve as the voice of the business
6. Cohesive/Decisive – Willing to make decisions, make them quickly, and stick to them.
7. Visionary – Capable of providing a clear and compelling vision of the product and its features to the team
8. Product Backlog Owner – Able and willing to assume primary responsibility for managing and prioritizing the Product Backlog, as well as providing acceptance criteria for each story.  
   [AWIMA Method Advisor, MC, SB]
9. Ideally, the Product Owner controls the budget. This allows the PO to steer the team (through prioritization and story refinement) towards the desired end state. If the Product Owner does not have influence over funding, this can be very problematic and should be addressed early. There must be clear understanding and lines-of-communication between those authorized to make scope changes (the Product Owner) and those who fund the project/team (stakeholders). We have seen cases where the business’ Program Office “pulls the rug out” from the Product Owner and eliminates all new/reprioritized features mid-project because they were not on-board with the concepts of variable-scope agile change control, leading to frustration and an inferior end-product. [TC, TH]
10. Those who are in the Product Owner role must be willing and able to be available throughout the entire iteration (rather than only the beginning and/or end). [CC]
11. Some large (multi-team) projects use a team of Product Owners (one per team). This can work, but every team must know exactly who acts as their One Product Owner, and there should be a “Chief Product Owner” who coordinates and prioritizes overall objectives for the Product Owner Team. The (Chief) Product Owner should manage their business stakeholders so there is a consistent vision for the product to the project team. [CC, MC]
12. The Product Owner is one person, not a committee. Committees may exist to advise or influence the Product Owner. Having many Product Owners (per team) is just as bad as having none or an unavailable Product Owner. [CC]

#### *Obtain an Agile Coach*

1. If the project team (particularly Project Manager, Scrum Master, or Product Owner) have not delivered an agile project before, engage an Agile Coach, particularly through the Initiate Project and early Sprint activities. [TH]

#### *Commit to* Doing *Agile (Not Just ‘Trying’)*

1. As Jedi/Scrum Master Yoda says, “Do… or do not. There is no try”. Using a different movie reference, the agile team should be formed from those willing to “take the red pill”. The team should commit to doing agile through at least the first release—not to simply trying it. This being said, agile cannot be forced onto an unwilling team. As Mike Cohn likes to say, “You cannot mandate self-organization.” The team must know that it has the freedom (and responsibility) to modify its practices within an agile context (self-organization). However, agile is good at exposing dysfunction, and new teams are easily tempted to revert back to their old habits or to blame the exposed problems on agile. It is very important for the Scrum Master and Product Owner to present a unified front and encourage the team to push forward, improvise, adapt, and overcome. [TH]

Define Project Charter

#### *Ensure Organizational and Client Support for Agile*

1. Agile will not succeed without active client (user) involvement throughout the project. [4i]
2. The project team must be empowered to make decisions about their process and practices (“self-organizing”). Ensure that the internal (HP) and client delivery organization understands this and will not interfere. Ensure that the team also understands that they are now responsible for doing their work in the most effective manner. Blaming “the process” or another organization for dysfunction is no longer allowed. [4i]
3. Not every organization is ready for agile, either because of internal (HP) factors, client factors, or both. It is much easier to find individual projects within an organization that can appropriately “go agile” rather than trying to transform the entire organization all at once. [TH]
4. Active user (client) participation is a must. At minimum, the client must be involved in story prioritization, specifying acceptance tests, and reviewing/approving completed stories at the end of each iteration (a.k.a. Sprint Review/Demo). [4i]
5. Persons performing the Product Owner role who are new to agile or this specific role should attend formal training at the beginning of the project to understand their critical responsibilities. [CC]

#### *Craft a Compelling Product Vision*

1. It is important to craft a product vision and share it with the team. Many teams use a format from Geoffrey Moore’s “Crossing the Chasm”: *FOR (customer) WHO (statement of need) THE (product name) IS A (type of product) THAT (has this compelling reason to buy/use). UNLIKE (competitive products) OUR PRODUCT (is differentiated in these ways)*. [TH]
2. The Product Vision should be compatible with the contracted Statement of Work. [TH]
3. It is important to re-visit the vision occasionally throughout the project to make sure progress is being made. Some teams like to read the Product Vision at the beginning of each Sprint’s planning session. There are several paths that an agile project may take with regard to vision:
4. (Ideal) The project stays on-track to the vision throughout the life of the project, delivering a product that meets or exceeds the original vision and producing high value to the client.
5. (Product Deviation) The project team loses track of the original vision and deviates from it, no longer showing progress towards meeting the intended vision. This is often caused when the Product Owner does not exert a strong leadership force in prioritizing the work to be done and expressing these priorities through Release and Iteration Planning.
6. (Vision Deviation) The original vision proves to be inadequate, and requires revision. This often occurs when the original vision is written too close to the original implementation design. It can also occur when business drivers change mid-project. Either of these cases is acceptable, and actually demonstrates a key benefit of agile delivery—adapting to change. However, the vision should be re-crafted once the team recognizes that it is no longer on the original course.
7. (Product and Vision Deviation) In the worst-case scenario, there is either no unifying/directing vision, or the product and vision drift aimlessly. Software is produced, but the business value is questionable. The team is unsure what to prioritize or why priorities are set the way they are. This is a very dangerous situation and points out the need for laying out a unifying vision early in the project. [TH]

#### *Identify Stakeholders and Dependencies Early*

1. Identify dependencies and stakeholders who will impact the success of the project. These people should participate in defining and implementing the Product Vision. [HR]
2. Many people will influence the course of the project. On large projects where there may be competing agendas, identify which people will be given direct influence (Scrum “pigs”) vs. having indirect interest (Scrum “chickens”). The “pigs” should be involved in training plans so that they are familiar with agile principles and practices. They will be allowed to participate in planning and daily Scrum activities. The “chickens” will be notified of plans and status, and may have access to register their priorities with the Product Owner, but they should not be given undue influence over the course or execution of the project. [CS]

Define Initial Project Scope

#### *Identify Business Benefits & Measures Early*

1. Business Benefits & Measures are very important so we know what to measure in order to ensure the expected benefits are being delivered to the client. [TC]

Obtain Project Approval

[no agile-specific guidance]

Establish Project Environment

#### *Consider Agile Project Management Tooling*

1. Large agile projects (>10 people) will likely not be able to get by with homegrown agile tools (e.g., spreadsheets). Strongly consider deploying a tool like HP Quality Center Agile Accelerator, Rally, or Version One on large agile projects, particularly if the team is not co-located. [TH]

#### *Automate Wisely*

1. Automation is very helpful for testing and continuous integration (CI), but not a necessity. Automate when it improves efficiency or velocity, but avoid the temptation to automate everything—some manual processes are ok, or even preferred. For example, with co-located teams, backlog management with index cards has several advantages over using digital tooling; however for distributed teams, a software-based solution is required. [TH]
2. Don’t scrimp on computing hardware for build cycles. This is the tendency but it will cost more in the end. [4i]
3. It is great to have a fast compile and unit test stack; but optimize for *thoroughness* in integration (compile, test, code quality). Build cycles can be reduced by running the full stack (test, quality, etc.) on a nightly basis rather than per-integration; additionally some tools offer a “pre-build” option which allows the developer to detect integrations that will break the build before actually committing. [4i]

#### *Assess Agile Training Needs*

1. Do not overlook training the entire team (including key clients) in agile/Scrum principles and practices prior to (or as a part of) the first iteration. An agile coach can be very helpful in identifying training needs and even conducting on-site workshops for the team. If feasible, prefer live training to pre-recorded, but there are many good pre-recorded agile courses and webinars on Grow@HP. [TH]
2. Train all of the team just-in-time with Agile PM and Engineering practices prior to project kick-off; in addition to the training, it “gels” the team quicker. Balance this advice with the fact that on exploratory projects, it is advisable to begin with a small "core team" to get through the initial exploratory phases before bringing-on additional technical staff once the architecture has been stabilized. [4i]
3. Training in Scrum/Agile should be conducted at a few different levels. Depending on each person’s role and previous experience, schedule the following training prior to (or during) the first iteration:
4. “Agile for Everyone” in which the entire team (including client) is trained together in agile principles, Scrum terminology, process framework (EPF Scrum or EDGE Agile), practices, and roles. This is usually a ½ day to full day course.
5. Agile Role-Based Training (one of the below for each person on the team):
6. “Agile Engineering Practices” in which the technical roles of the team (including testing) are introduced to various agile engineering practices and tools, as well as concepts such as story-point estimating and evolutionary design. This could be a combination of courses, but could take 1-2 days depending on the level of expertise required. Note that many engineering practices can be taught informally on a just-in-time basis if there are enough experienced developers on the team to mentor/coach the others.
7. “Agile Product Owner and Business Analysis” in which the Product Owner and key clients are given in-depth training in managing the Product Backlog and defining user stories. This is typically a 1-2 day course.
8. “Agile Scrum Master” which provides in-depth training to new Scrum Masters (equivalent to a Certified Scrum Master course). This is typically a 2-3 day course. [TH]

#### *Procure a Project-Owned Agile Infrastructure*

1. It is difficult for a project to be truly agile when it is at the mercy of external teams/resources for build and test environments. If at all possible, procure a unique environment (server, VM) that the project team may configure for build and test purposes. If this is not possible due to lack of funding or expertise, the team should plan to expend additional time managing/debugging their build and test infrastructure. [TH]
2. Do not ignore infrastructure or tolerate an unstable development/test environment. This will slow down progress, introduce risk, and will cost more in the long term. [CC]

#### *Ensure Adequate Communication and Collaboration Infrastructure*

1. Agile projects require much more real-time communication and collaboration than plan-driven projects. Ensure that adequate thought and planning is spent on the communication infrastructure, especially if the team is distributed. Consider travel arrangements (on-site visits), phones, instant messaging, e-mail, conference rooms (physical and virtual), speaker phones, wiki/SharePoint, whiteboard, webcam. [TH]
2. Consider using HP Sky Room to do real-time collaboration across distributed teams. [TC]

#### *Involve the Whole Team in Planning and Communication*

1. For agile, the staffing model will include more roles from the start of the project. All members of the project who are assigned during Initiation should participate in planning and communication (e.g., kickoff) activities. [TC]
2. To quickly gauge consensus and support levels during whole team planning, consider using the “Fist of Five” technique where each team member responds by showing a fist (consensus block) or a number of fingers to indicate their level of support (1=no support; 5=max support). [SB]

#### *Perform Key Startup Activities Prior to First Iteration*

1. During or prior to a Kick-off meeting that concludes project initiation, an agile project should have answers to the following questions and addressed them in planning/risk mitigation:

* Have we identified who will perform the roles of Product Owner (usually a client or internal project manager) and Scrum Master (usually a project manager or technical lead)?
* Do we need an Agile Coach?
* Is everyone on the team (particularly the client) “all in” on doing this project agile?
* Have we captured an initial list of user stories (scope) in a Product Backlog?
* Have we estimated each story in the Product Backlog?
* Have we defined “Done”?
* What aspects of the project will be challenging to agility (e.g., distributed team)? [4i]

Perform Iteration Zero

Define Team Organization

#### *Context is Key*

1. There is no silver bullet: it comes down to people, practices, communication, expertise, tooling, process…and effective execution. Agile is not prescriptive—what works for one project may not work for another, and vice-versa. When discussing practices, context is very important; when discussing principles, they should generally apply universally. Therefore, when making decisions, ensure that the team and client organization understands both the unique context of the project and agile values/principles. [4i]
2. New practitioners might be frustrated with the nature of guidance and “lessons learned” for agile development, because most of the advice will be in the form “consider…” or “the team should”. Agile coaches famously begin answers to yes/no questions with “it depends”. This is understandably frustrating, but indicative of the nature of agile, where so many of the decisions and standards depend on context and the project “ecosystem”. The team must become comfortable with ambiguity and the fact that Scrum is an *empirical* process (not a *defined* process). [TH]

#### *Implement Practices Incrementally*

1. The project does not need to implement all agile practices from the beginning. Choose the ones that will have the most benefit and continue looking at ways to improve your practices as you go. It is better to implement a few practices well than many practices poorly. [4i]
2. Balance the desire to implement practices incrementally with the fact the people will naturally resist change to their habits and thinking, even if it might address particularly pressing issues. The Scrum Master and experienced agile team members should help to encourage continuous improvement throughout the project. [TH]

#### *Define “Done”*

1. Define what “Done” will mean within the iteration. “Done” should preferably be defined before any estimate or release plan is delivered to the client, and used as an input to the estimation process. “Done” can evolve over the life of the project (which may change estimates), but should always be defined and every team member must know what the current definition is. [4i]
2. Quality is not negotiable. Do not compromise on quality and the definition of done, [CC]

#### *Decide How Requirements Will Be Documented*

1. Make some initial decisions around how and to what level requirements will be defined during the project (e.g., high-level user stories, detailed use cases, or somewhere in-between). Small co-located projects can usually get by with simply-documented user stories and face-to-face interactions with the client. Larger projects may need to develop use cases and add details iteratively through the project. *Note: under no circumstances is it recommended to use agile with a process that requires full requirements elicitation and documentation before beginning development work.* The degree of documentation is dependent on several factors, so there is no one-size-fits-all answer to this question. The general guidance is to choose the lightest-weight approach that can work. Consider:
2. The contract and customer expectations
3. The level of collaboration and trust with the customer (higher levels require less documentation, lower levels require greater documentation)
4. The degree to which the team (particularly developers and testers) will have direct access to the client
5. Support organizations (particularly Testers/QA), who might be assuming higher levels of detail
6. The tools available [4i]

#### *Prefer Simplicity (Less is More)*

1. Those who have worked on enterprise-class projects for many years have a tendency to think that “bigger is better” when it comes to plans, documents, tools, designs, processes, etc. because of the incorrect assumption that complicated solutions are required to address complex environments. This is not true—often the best way to attack complexity is through simplicity and application of lean principles (e.g., flow, eliminate waste). A corollary of this guidance is when presented with options, prefer the simpler and least-cost solution first—you can always expand/refine the solution later, usually at little additional cost. “The perfect is the enemy of the good” (Voltaire). [TH]
2. Simplicity--the art of maximizing the amount of work not done--is essential. [Agile Manifesto Principle]

#### *Collaborate as “One Team”*

1. Agile teams should be cross-functional. Avoid creating or assuming roles by technical ‘specialty’. For example, do not assume that only BA’s write use cases, Data Engineers develop data models, Testers perform tests. Avoid labeling each other into specializations (e.g., “He’s the database guy”). Each person on the team should strive to be a “generalizing specialist” who is able and willing to help the team in any way necessary to progress towards the goal iteratively and incrementally. [TH]
2. The best architectures, requirements, and designs emerge from self-organizing teams. [Agile Manifesto Principle]
3. Individuals are shaped by their background and their experiences. It is important to respect the different people who make up a team. [Scrum “Respect” value]

#### *Define and Evolve Team Standards and Rules*

1. Early in the project, the team should define standards and rules it wants to abide by (e.g., everyone should be present for a 9 AM Scrum in the team room). The rules can be minimal at the beginning of the project and should evolve over time. We usually recommend some standards around:
2. Definition of “Done”
3. Coding Standards (to promote healthy refactoring and collective ownership)
4. User stories (format, content, management)
5. Check-in, Check-out, and Build (e.g., if you break the build…)
6. Participation and execution of meetings (Spring planning, Daily Scrum, Retrospectives)
7. Working rules – quiet hours, asking for help, pair or no pair, updating hours remaining, etc.
8. Tools and configurations [TH]
9. Post team standards and rules in an easily assessable place (team room on a wall, wiki or SharePoint for distributed teams, etc.). [TH]

Initialize Product Backlog List

#### *Utilize a Single Product Backlog*

1. It is tempting to treat different types of requirements (functional, non-functional) and changes (defects, change requests) separately. It tends to work better to manage every request in a single backlog (although you can certainly type them as shown below if this is important from a measurement or status point-of-view). The Product Backlog ideally contains:

* Functional user stories or use cases
* Non-functional user stories (security, performance, logging, backup, help, internationalization, etc.)
* Newly-identified features or revisions (change requests during the project)
* Non-trivial technical/architectural spikes
* Defects discovered in testing (particularly those that are not addressed within the current iteration)
* Defects discovered post-release
* Requests for specific documents (user manuals, design documents, usability studies, etc.)
* User training  
  [4i, TH, CC]

#### *Compose the Product Backlog at Appropriate Detail*

1. The product owner is not required to describe in detail every item being tracked on the product backlog. Depending on the size of the backlog and the speed of the team it may be sufficient to describe only the high priority items, saving the discussion of lower priority items for the next sprint planning meeting. Typically, the Scrum team will provide guidance when they start to get further into the backlog list than they know could be done in the next sprint.

Perform Technical Launch

#### *Perform Oversight Reviews in an Agile Context*

1. It is typical for large and/or critical projects to conduct oversight reviews of a management and technical nature. If the reviewers are unfamiliar with agile concepts, the recommendations may conflict with known agile best practices (e.g., limiting speculative up-front architecture/design). In such a case, begin the review process with an overview of agile/lean principles and consider bringing in an experienced agile coach to facilitate. Such oversight reviews should be informed by key agile “paradigm shifts” (see “Governance and Compliance” lessons learned starting on page 25 for specifics). [TH, CC]

Create Initial Release Plan

#### *Employ “Two-Level” Planning*

1. It is vital that the team and stakeholders understand and support the notion of two levels of planning in agile: 1) coarse-grained high-level plans for the product vision, roadmap, and release plan (major milestones); and 2) fine-grained detailed plans for each iteration (Sprint) and day (Daily Scrum commitment), generated on a just-in-time basis. Detailed plans do not exist for future iterations. This notion must inform how the team utilizes planning tools like Microsoft Project (i.e., do not create 1500 line MS Project schedules for an agile project). [TC, DL]

#### Create Lightweight, Adaptable Release Plans

1. Release Plans should be adaptable to change, particularly regarding releases beyond the one currently in development. Thus, they should not be especially detailed or difficult to adapt as the project proceeds. At minimum, a Release Plan should indicate:
2. key milestone dates, particularly those that affect external teams (e.g., interfaces, QA, governance, operations)
3. a name and/or unifying theme for the release
4. number of planned iterations and their start/stop dates
5. any key assumptions or constraints (staffing, velocity, etc.) the plan is based on
6. a list of features and/or stories planned for the release
7. an assessment of the value to be provided by the release  
   [TH, SW]

#### *Deliver Early and Often*

1. Agile is predicated around delivering frequently. Ensure that the team is actually delivering working, tested software (not documentation, designs, etc.) at the end of each iteration. Make sure that the code is “shippable” (whatever you define that to be), and try to deploy to production at most every 6 months. [TH]
2. The following enabling practices seem to be the key contributors towards frequent delivery of shippable code:
3. story-based planning and estimating (vs. component-based)
4. continuous integration (throughout the iteration)
5. automated build, coupled with
6. automated testing (at least unit testing, but also integration testing with something like FIT).  
   [4i]
7. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software. [Agile Manifesto Principle]

#### Create Timeboxed (Rather Than Scopeboxed) Release Plans

1. Determine the release dates first, and then make decisions about which most-valuable features can fit within those releases (based on the current estimates, velocity, and release lifecycle). It works better to develop a regular release cycle with fixed dates/variable scope than to try to determine/adjust release dates based on a fixed scope. [TH, SW]
2. It is important to defer commitment to exactly which features will be in the product at each release, because this makes it possible to reliably release the product on time. The release date is guaranteed, but the exact features are not. Timebox – don’t scopebox. [CC]

#### Build Release Plans Around Minimum Marketable Features (MMF)

1. Each release plan should be composed of a set of features that are:
2. Cohesive, meeting the product vision
3. Marketable, providing high-value to the stakeholders, product users
4. Small (minimal) both as individuals and as a group
5. Decomposed into one or more estimated user stories [TH, SW]

#### Write Releasable User Stories

1. User Stories should be written at a level in which they can be included or removed from a release without great care. For example, “Allow user profile to be stored” is a better story than “User profile screen” + “Save user profile to database” + “Retrieve user profile from database”. [TH, SW]

Develop Project Testing Strategy

#### *Integrate Testing into the Process (vs. at the End)*

1. The testing strategy for an agile project should “build quality into the product in the first place” (Deming) by ensuring that testing occurs alongside (not after) feature development. Testing is no longer a lifecycle phase—it is a continuous activity. This is usually accomplished through a combination of the following test-related strategies:
2. Tester involvement in user story refinement and specification of acceptance criteria as a necessary part of story definition
3. Automated unit, integration, and acceptance testing
4. Incorporation of automated tests into the continuous build environment
5. Small (minimal) hand-offs between programmers and testers (they work together)
6. Lightweight (fast-turnaround) defect tracking and management
7. Exploratory testing
8. Test in the same Sprint in which the feature is developed
9. Track test coverage
10. Incorporation of testing into the “Definition of Done”
11. Consideration of how various facets of testing will be performed during the Release Cycle (e.g., stronger emphasis on performance testing during ‘hardening iterations’  
    [TH]

#### *Automate Testing as Much as Possible*

1. Automated tests should be the norm—not the exception—on agile projects. Automated testing frameworks for every type of test: unit, integration, component, system, acceptance. [TH]
2. Guidelines for selection of regression test cases for automation are provided below. Consider using the assessed implementation risk for each feature to determine the candidate test cases for automation.
3. **Feature Smoke tests** to verify that the definition of 'done' is achieved, testing the basic transactions. Give these automated test suites to the developers as their 'exit' criteria. Testers run these again in the integration testing environment
4. **Integration Smoke tests** (build smoke tests) to verify that the code is ready for system testing (hardening sprint)
5. **Solution Smoke tests** to verify that the key features developed in one sprint are not broken in subsequent sprint cycles. [PE]

Separate Black-box (behavioral) unit testing from white-box (code-structure based) unit testing [PE]

1. Unit testing is typically the responsibility of developers. However, for agile projects, it is recommended that the responsibility for white-box testing be assigned to developers, while the testers can focus on black-box (feature-driven) unit testing concurrently. This provides complementary but comprehensive focus to unit testing, while also enabling early detection of defects that would typically be found in later test levels (e.g., system testing) and reduces cycle time for defect fixes and rework. [PE]

#### *Integrate Automated Tests With the Continuous Integration Build Environment*

1. Automated tests should run against the continuous build environment. Typically this is set-up either on a unique integration server or done in-memory so that these tests do not impact other manual testers. [TH]

#### *Location Considerations for Agile testers*

1. While it is ideal that the entire team is co-located, there may be situations that demand distribution of the team members geographically. Some of the possible configurations are listed below:
2. If there are multiple Scrum teams, each individual Scrum team could be co-located
3. Testers in a Scrum team may be in a location different from the developers and Business Analysts
4. Testers, Business Analysts and Developers may be co-located but not the product owner

In such cases, the following guidelines may be adopted for effective teamwork across the distributed teams:

1. Entire team can be co-located onsite for initial cycles, and after establishing rapport, some of the team members can, if required, move to distributed location in later Scrum cycles
2. Cross-pollination is an option- some onsite members go to offshore location temporarily and vice versa
3. Consider designating one of the onshore team members as a 'bridge'—this person would be responsible to bring the offshore/distributed team up to speed on an ongoing basis
4. Independent testing could be done offshore for external validation in some cases, while embedded testing (by testers who are part of the Scrum team) is done onshore
5. End-to-end business processes that cannot be tested within a single Scrum team can be performed by independent testers from a distributed location
6. Establish a mutually agreeable time so that all team members globally can participate in the daily scrum meetings
7. Apply the guidelines for *communication and collaboration* *infrastructure* for effective communications. [PE]

Define Data Strategy

#### *Align the Data (Engineering) Strategy with an Iterative/Agile Approach*

1. Resist the temptation to analyze and model a complete database design up-front. A data engineering strategy should be developed early that is compatible with iterative feature development. This is usually accomplished through a combination of the following data-related strategies:
2. Involve data engineers as an inclusive member of the Scrum team, just like testers and analysts
3. Define data persistency requirements as stories on the Product Backlog
4. Define data-related non-functional requirements (security, encryption, performance, etc.) as stories on the Product Backlog
5. Incorporate agile modeling and database refactoring techniques (Scott Ambler) and tools
6. Define a persistency approach that can be evolved and is open to refactoring (e.g., JPA)
7. Determine the degree to which developers will have authority to define/modify database entities and queries (agile prefers very open access privileges, within the context of team standards)  
   [TH]

Define Project Architecture

#### *Identify “Architecturally Significant” Stories Early*

1. There are big, one-time decisions that must be made regarding architecture. If these decisions are later changed there will be a cost / schedule impact. The biggest-impact (irreversible) decisions should be validated as early in the project as possible. Note that in the agile context, “validating decisions” means building prototypes, spike solutions, and an ‘architectural runway’—not simply modeling and reviewing ideas with ‘experts’. [4i]
2. Make sure to account for sufficient time at the front of the project in order to procure hardware and configure development and test environments – this is often scheduled as part of an "Iteration Zero". [4i]

#### *Identify Architecture/Design Decisions that can be Deferred*

1. As a corollary to the guidance to identify which architectural decisions must be made early, also identify areas in which irreversible decisions can be deferred (employing the lean principles of *deferred commitment* and *last responsible moment*). Many decisions can be deferred until later iterations which will allow the team to substitute alternatives that experience has shown to be superior. [TH]

#### *Avoid Speculation; Validate Through Delivery*

1. Do not allow the team to get bogged-down in speculative "what if?" scenarios too long, particularly during project initiation and the early iterations; if the answer is not clear, it usually becomes clearer by starting the work and re-examining the issue later (in lean, this is called the *strategy of deferred commitment*; see Poppendieck [<http://www.poppendieck.com/design.htm>] for a fuller discussion). Another technique commonly used, when there are two alternatives, is to try both (i.e., prototype, spike solution) for a short period (Iteration Zero) and then determine which is more likely to succeed. With agile, it is better to make decisions based on experience rather than speculation. This principle is true in many areas--technology, design, process, tools, techniques, etc. [4i]
2. There should be a “healthy tension” between a planned, risk-driven architecture and the concepts of agile, evolutionary design. If it feels like there is too much up-front architecture (e.g., unproven models, standards, frameworks) then there probably is. On the other hand, if it feels like there is too much ad hoc programming and competing architectural visions, then the team probably needs to step back and do more formal architecture definition. [TH]

Produce Initial Size Estimates

#### *Estimate Agile Size in Story Points or Use Case Points*

1. For agile, function points may not be appropriate, due to a level of early design decomposition that is undesirable and time-consuming. Agile projects typically estimate the size of each story on the initial Project Backlog in story points or use case points. [4i]
2. Agile estimates benefit from some form of Wideband Delphi technique such as Planning Poker. One way the team can to do this quickly and easily is to:
3. Create an index card for each story on the backlog
4. Sort the index cards into piles by relative complexity (typically you should end-up with 10 or fewer piles).
5. Use Planning Poker to estimate the story points for each stack, assign that value to each story in the stack. Do some cross-referencing to ensure that a “4” will take twice as long as a “2” and half as long as an “8”, for example.
6. Using the definition of “done”, make an initial guess at the number of hours that will be required for each story point (typically between 5 and 15 hours). Compare the total number of project development hours with any previous estimates and account for any significant discrepancies.
7. Document any assumptions that were used during the estimating process so that future estimates can utilize the same technique. [TH]
8. Use a Fibonacci sequence for assigning values to agile estimates (each number is the sum of the two prior – 1, 2, 3, 5, 8, 13, 21, epic). This reinforces the reality that the larger estimates are less precise. [4i]
9. Lightweight *Use Cases* are beneficial for iterative and waterfall estimates. It seems best to get the initial high-level requirements documented in a Product Backlog down to at least a Feature level, unless it is an optional-scope contract. This hierarchy seems to be helpful: [TH]

|  |  |  |  |
| --- | --- | --- | --- |
| Scope | Are estimated in… | Are documented in… | And decompose to…. |
| Epics | Ideal Engineering Months | Epic cards and/or High-Level Use Cases | Features |
| Features | Ideal Engineering Weeks or Use Case Points | Feature cards and/or Use Cases | User Stories |
| User Stories | Story Points and/or Ideal Engineering Days | Story cards and Use Case scenarios (if needed) | Tasks |
| Tasks | Effort Hours (max 16 hrs) | Task cards | n/a |

Conduct Technical Review

#### *Negotiate Who/What/When for Technical Reviews*

1. For agile, need to work with the CTO to determine the level of review - perhaps focus on the Initial Agile Architecture. Traditional items will not be complete, Risks need to be documented and tracked as the project moves on through the release cycle. [SB]

Validate Solution Design and Strategy

[no agile-specific guidance]

Plan Project Work

#### *Plan Appropriately*

1. People manage plans, plans should not manage people. That said, plans are commitments that once communicated become an expectation and informal (or possibly formal) contract between you and your customer.  Customers like to hear the iteration planning process described in terms of features/stories being "committed" to the [Iteration Plan](https://kcteam.emea.eds.com/sites/541/Iterative%20Wiki/Iteration%20Plan.aspx), and this term reinforces to the delivery team what they are doing.  It is NOT a good idea to "commit" features to the [Release Plan](https://kcteam.emea.eds.com/sites/541/Iterative%20Wiki/Release%20Plan.aspx), because such plans are more likely to change (a better word in that context is "allocate").  If needed, consider using risk-based commitments based on experience to-date, such as "We are 90% likely to deliver at least 80% of the requested functionality by date X". [4i]

#### *Be Confident*

1. Be confident in your project management approach, irrespective of what approach you are using. The client will quickly sense inexperience or timidity and capitalize on it in low-trust environments. [4i]

Define Project Process – Interactive

#### *Be Careful With Tailoring*

1. The legacy EDGE methods (sequential waterfall and OO iterative) are tailorable but have been typically interpreted and executed in waterfall/phased fashion. Iterative (particularly Agile) should NOT be implemented as 'mini-waterfalls'. For example, a plan that looks something like this: Iteration 1 = Analysis, Iteration 2 = Design, Iteration 3 = Build, Iteration 4 = Test; demonstrates an incorrect understanding of “iterative” development and is actually more indicative of a waterfall approach. [TH]
2. Within a method, agile practices are mutually-dependent. Thus, the team should be VERY CAREFUL about adding or removing practices without a good understanding about what they are affecting. For example, suppose a team is doing Scrum, but decides to not do Sprint Reviews ('demo'). How, then, do they know the client is satisfied with what has been built to-date? Does the team realize that human nature will now tend to drive the team towards a more lax view about which features are truly “Done” in an iteration? Suppose on the other hand, that the team decides that a formal walkthrough will be done on every piece of code that is written during the iteration. Does the team have an idea how this might affect the overall velocity? When will this review take place--before or after the code is formally integrated? How will this affect the practice of continuous integration? Will test code be reviewed? How will the team validate that all code has actually been reviewed? Are there any alternative practices (e.g., pair programming) that might be more efficient/productive? [4i]
3. The legacy EDGE methods and BPR templates are in general written for large, enterprise class projects. The EDGE custodians will gladly tell you that most process-related bureaucracy/dysfunction is SELF-INFLICTED. Assume that most of the low-level activities and high-detail templates DO NOT APPLY to agile projects. [4i]

Prepare Plan Components - Agile

#### *Avoid “Two Sets of Books”*

1. Project Managers typically start agile projects with external expectations about the use of certain templates and documents (aligned with PMBOK and CMMI requirements). In order to avoid conflict, agile projects have been known to create “shadow” documents (plans, schedules, etc.) for the purpose of complying with organizationally-mandated standards. Such documents rarely resemble what is actually happening on the ground. This practice is wasteful and can contribute to negative perceptions of agile as a “rogue” activity. It is generally better to try to engage oversight groups early—Mike Cohn recommends the following actions:

* Negotiate and set expectations up front. If necessary, pitch the project as a one-time experiment.
* Fit your reporting to current expectations. Don’t fight the expectations, but do try to shift them over time.
* Invite them into your process (e.g, Sprint Planning, Daily Scrum [as chickens!], Sprint Review, Retrospective)
* Reference a success. “Nothing convinces like success.  
  [TH, MC]

Estimate Project Costs – Agile

[no agile-specific guidance]

Create Project Schedule

#### *Coordinate Project Schedules with Other Projects*

1. When multiple projects need to coordinate their (testing/deployment) schedules, the Project Manager and Product Owner need to ensure that adequate coordination takes place. There are several proven strategies that apply:
2. Ensure the other projects are represented during Product Roadmap and Release Planning. The Product Owner must take into account commitments to integrate with other applications when developing long-range plans and see to it that those priorities are reflected in Sprint Planning.
3. Utilize milestones in the long-term schedules that commit to dates that the various components must integrate with each other for build, testing, and deployment. This allows each team to develop using their preferred methodology while still meeting required commitments.
4. If other project teams require intermediate components or documents from each other in order to progress (e.g., design documents, WSDL, test cases), add those items to the Product Backlog and prioritize appropriately.
5. Invite the other teams to participate in the Scrum meetings (as observers)
6. If they are large, encourage the other projects to decompose into smaller projects and teams which can be more easily coordinated and prioritized

Package and Secure Commitment

[no agile-specific guidance]

Execute Project Plan

Deploy Project Resources

#### *Co-locate the Team or Compensate for Distribution*

1. Collocation with the business rep and the rest of the development team is a critical success factor with agile. It is possible to succeed without this, but the project team must compensate with additional overhead and practices (more written documentation, online planning tool, customer-written acceptance tests, prototypes, mid-iteration demos, longer iterations, 2-phase acceptance testing, etc.). [4i]

#### *Organize Around Features—Not Components*

1. It is preferable to organize the technical teams around features (stories) rather than functions (Architecture, Analysis, QA, Data Engineering, etc.) or technical components (online, batch, database, etc.). For example, a typical feature team might consist of 1 BA, 3 Developers, 2 Testers, and a Data Engineer. Such teams are often called “feature teams” or “story teams” and are responsible to define/build/test entire stories and features. With the advent of Service Oriented Architecture (SOA), stories can written/organized at the service level, and developed completely by the feature team (roundtrip request-to-response). Component teams, if used, should be the exception. [TH]

#### *Organize Around Small, Self-Sufficient, Co-located, Feature-Based Teams*

1. Studies show that team productivity (average per programmer) is inversely-proportional to team size. In fact, Scrum has a rule that teams should have no more than 7 members (link). A more memorable way to think about this is to strive for "Two Pizza Teams"--those which require no more than two pizzas to be fed (roughly 5-10 people). Additionally, consider breaking large projects into co-located feature-based subteams which can completely control some aspect of the work (Service-Oriented Architecture is well-suited to this approach--team-per-service or team-per-feature). If there are multiple specialties involved in the project (analysts, data engineers, testers, etc.), it is much better (from a productivity and quality perspective) to create integrated end-to-end FEATURE teams rather than segregating by components or technical functions (which tends to lead to an inefficient, error-prone "over-the-wall" processes). [4i]

#### *People are Not “Resources”*

1. People on an agile project team should not be referred to as “resources”, because it is a) incorrect; and b) dehumanizing and contradictory to the spirit of the Agile Manifesto (*“value people and interactions over processes and tools”*). Resources have capacity—people have capability. For example, a laptop is a resource. Two laptops of the same model have equal capacity. However, it is impossible to assess people this way; studies have shown that top developers can be up to 50x more productive than their junior counterparts, but surely this is only on some types of work. Treating knowledge workers like plug-replaceable parts is not correct or helpful towards promoting self-organization. Prefer words like “talent”, “knowledge workers”, or simply “people”. [TH]

Manage Plan Execution

#### *Foster an Enabling Culture*

1. Don’t underestimate the importance of communication, collaboration and team-building, irrespective of methodology. [4i]
2. Trust, understanding and cooperation will overcome lots of potential “show-stoppers.” [4i]
3. Agile is not for everyone; even the best people may not adopt the philosophy. Move negative influences off the team if they refuse to adapt--monitor this closely during the first several weeks of the project. If the negative influence is a key stakeholder, ensure the person understands the purpose of agile delivery methods--the best argument is results. If the team is not producing visible good results after the first release cycle, consider "throttling back" to a more traditional iterative approach and re-visit the Method Advisor. [4i]
4. Try to do everything together as a team: co-location, story collection, estimation, design, development, testing, status meetings, demos, etc. This is a key aspect of "self-organization". While some members will have more skill in certain areas than others, agile works better when every member shares ownership and develops mutual trust and empathy with each other. Stove piped roles, responsibilities, and an 'ownership' mentality will almost certainly cause lower productivity, less-than-honest feedback, and poor morale ("code monkey syndrome"). [4i]
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done. [Agile Manifesto Principle]
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation. [Agile Manifesto Principle]
7. Be willing to commit to a goal. Scrum provides people all the authority they need to meet their commitments. [Scrum “Commitment” value]

#### *Set and Expect a Sustainable Pace*

1. If executing properly, an agile team will be mentally exhausted at the end of an 8-hour day. That is a good sign. Expect no more and no less. [TH]
2. Level the workload by evening out the arrival of work. Manage the schedule by limiting work to the capacity of the team. Let the team pull though the work and avoid pushing work. Ask not – How long will this take? Ask instead – What can be done by this date? Allow the team to establish a regular heartbeat and cadence. [CC]
3. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely. [Agile Manifesto Principle]

Manage Project Results

#### *Use Appropriate Metrics to Assess Agile Project Progress*

1. Completed stories are the primary measure of progress on an agile project. Stories released are the primary measure of earned value (an undeployed feature has no value). Do not measure progress or earned value by effort hours (“navigating by the gas tank”, as Ross Pettit calls it). The agile rule of thumb on metrics is: “Measure outcomes, not activity.” [TH]
2. When asked to provide metrics to governance organizations, ask “What decision do you need to make?” and then “What information do you need to make that decision?” It is likely that the data collected from an agile project will be different from traditional projects to determine 1) am I going to be late? Or 2) am I going to run out of money? [TC]
3. Some measures that are appropriate to assess agile project health are listed below. However, note that many of these measures are not currently comparable across projects because of different calculation methods for story points and velocity.
4. # of completed (“done”) stories / Running Tested Features (RTF) over time/iteration
5. # of released features or business value (revenue, profit) delivered per release
6. # of running automated tests
7. # of releases
8. # of stories in work (try to keep low; a high number indicates too much WIP)
9. # of stories that did not get completed in the committed iteration (drop, defer)
10. # of open defects at iteration close
11. Mean defect time-to-close
12. Team velocity vs. capacity
13. Team budget burn vs. value (story) delivery
14. Code quality (various, particularly complexity)
15. Working software is the primary measure of progress. [Agile Manifesto Principle]

#### *Be Wary of Misused Metrics*

1. Various metrics that are traditionally used to measure project health or quality can be misused on agile projects. Specifically,
2. Test coverage can be misleading. A low number is always bad, but a high number does not necessarily mean that the tests are verifying the right things (or anything at all, for that matter!).
3. Lines of code can be misleading when you consider that refactoring and simple design promote small, cohesive components that keep lines of code minimal.
4. Effort hours can be misleading if not compared to stories that are delivered during that effort. In general, smaller teams are more productive than larger teams.
5. Percent complete (completed stories vs. non complete stories) can be misleading because the highest-value and highest-risk stories are often completed early. Also, most large project plan a ‘hardening iteration’ or two towards the end of each release cycle. [TH]

Manage Project Exceptions

#### *Do Not Move Milestone Dates*

1. Resist the temptation to change milestone dates such as iteration end dates and pre-agreed release dates. Firm milestone dates force the project team to make decisions and trade-offs in order to meet those dates, which is a key success criterion for iterative and agile. This is also why the highest-priority (value) features should be developed early (to obtain feedback allowing for timely correction/change). Milestones provide a framework for delivery and decision-making (adaptation)--if the team does not stick to them, it will tend to "drift". [4i]

#### *Be Transparent (Do Not Ignore “Bad News”)*

1. Agile is good at surfacing issues quickly and early. For example, overly-optimistic estimating. However the team must use this “bad news” to make corrections. If the team is not meeting commitments or following agile principles or practices, ADMIT IT (don’t ignore it), and decide as a team what needs to be done differently to correct the situation. Adaptive development done well requires constant small corrections (the XP “driving a car” metaphor). Ignoring the bad news leads to loss of trust. [TH]
2. Transparency is important. Make sure the Release Plan, Sprint Plan, Release Burn-Up chart, Iteration Burn-Down chart (and others) are posted in a public place (war room, team portal, wiki) where everyone (including the customer) can see it. [TH]
3. Encourage the team to embrace the power of “No”. – Don’t be afraid to say “No”. Examples:

* "No, we don't have the capacity to do A & B at the same time."
* “No, we cannot work on this new story now, it will need to wait until the next iteration.”
* “No, this person cannot be pulled-off the team mid-iteration to work on your special project without stopping the Sprint.”
* "No, at our current velocity, we cannot see a way to complete all the features you want in the time frame you desire."
* "No, adding another person to the team at this time will slow us down." (be mindful of Brooks’s Law - Brooks's law is a principle in software development which says that "adding manpower to a late software project makes it later"). [CC]

1. Scrum keeps everything about a project visible to everyone. [Scrum “Openness” value]
2. Have the courage to commit, to act, to be open, and to expect respect. [Scrum “Courage” value]

Manage Intellectual Property

[no agile-specific guidance]

Manage Requests for Change

#### *Change Control Still Applies to Agile*

1. Neither Iterative nor Agile imply no change control. Make sure the customer understands that while (same size) feature substitution is a no-cost option, new features (especially those that affect the contractually-specified scope or Project Charter/Vision) require standard change management. The formality of the change control process should be informed by the contract (variable-scope contracts require less formal change control than fixed-scope). [4i, SB]
2. A more “agile friendly” approach to change control is to write a variable-scope contract where we simply extend the contract until the totality of the required features (both original and new/changed during the project) have been built. In this scenario, “change control” can be very informal (completely at the discretion of the Product Owner). If formal change control is “painful” to the client, suggest changing the contract to variable-scope. [TC]
3. When adding new features/stories to the backlog, change will be accommodated provided the total Story Points do not exceed the total amount stated in the Release. Features/stories that are inserted into the Product Backlog should require equivalent size features/stories are removed from the release. [CC]

#### *Embrace Change*

1. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage. [Agile Manifesto Principle]

Sprint N

Plan Sprint

#### *Plan the Iteration as a Team*

1. In agile, the whole team plans (not just leads/management) and commits to the plans. Do not plan in a vacuum—iteration planning (Sprint Planning Session) should take place at the beginning of the iteration (just-in-time) with attendance by the whole team. Full-team planning has the following benefits:
2. Commitments are more trustworthy, because every member of the team had the opportunity to participate in the plan creation (rather then a planner committing for them)
3. Team members learn that making and meeting commitments is a whole-team responsibility (vs. individual)
4. Every member of the team is aware of the relative importance of the various plan components and their personal responsibility for making it happen
5. Minimal wait time for decisions or commitments. Note that waiting for decisions is a significant waste of time and can lead to miscommunication or bad assumptions.
6. More and better insights into the planning process, which should reap better results. [CC & TH]

#### *Choose an Appropriate Iteration Length*

1. Optimum iteration length depends on many project-specific variables (e.g., team experience, size, distribution, compliance expectations, requirements dynamism). Inexperienced teams should consult an agile coach/SME when deciding what an appropriate length should be. [SB]
2. Two-week, three-week, and 30-day iterations are typical of HP ES agile projects. [4i, TH]
3. In general, avoid iterations longer than six weeks or shorter than two weeks. Only experienced teams in demanding environments should attempt one week iterations. [4i, TH]
4. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale. [Agile Manifesto Principle]

#### *Keep Iteration Length Consistent Throughout the Release Cycle*

1. Timebox the iteration - Iteration end dates must be firm and inflexible. Do not use different iteration lengths within a single release cycle. If the iteration length is inconsistent, the team will have a difficult time creating a cadence/flow, and the planning team will have a difficult time assessing velocity and associated trends. [TH]

#### *Synchronize Sprints Across Teams*

1. If a project has multiple scrum teams, synchronize the sprints (i.e., start and end on the same days). [CC]

#### *Consider Utilizing Inter-Sprint Breaks*

1. Consider having an Inter-Sprint break. That is, do not schedule back-to-back sprints but instead schedule one or two days between sprints. An Inter-Sprint break can help:
2. Avoid stress and Burnout: In real life, you can’t always sprint. In Scrum sprints are intensive - we sprint to the end goal, not walk.
3. Pay down technical debt: reduce the number of compiler warnings; improve test coverage for old code that lacks coverage;
4. Innovation & creativity: use the time to hold brown bags to share new ideas or research new technologies that may help the project; perform technical spikes on future stories;
5. Reduce distractions: schedule non-project meetings during this period, catch-up on organizational announcements;
6. Planning: any learnings from the completed sprint should be reflected in the next sprint stories which may require re-estimation
7. Perform any configuration management merges between scrum teams. [CC]

### **Consume Lessons Learned**

#### *Promote Team Learning*

1. High-performing agile teams continually seek out new ways to learn, share knowledge, and adapt. At the beginning of each iteration (minimally), socialize the lessons learned and improvement items identified during retrospectives and adapt how the team will work during this iteration. View each iteration as an experiment and an opportunity for continuous improvement. [SB]

### **Determine Team Velocity**

[no agile-specific guidance]

### **Define Goals and Select Features**

#### *Avoid the Temptation to Over- or Under- Commit*

1. Be careful of “over-stuffing” the iteration - don't break trust with a team by stuffing more work into the iteration. Don't de-motivate a team by expecting them to complete more than their original commitment, despite the extra work of the emergency. Every exception you grant erodes the fundamentals of agile and the chance your effort to change your development method will succeed. Stick to working at a sustainable pace, sticking to iterations, managing scope one iteration at a time, delivering working software every iteration. Pull, don’t push. [CC]
2. As a guide, “must have” iteration outcomes (MoSCoW prioritization technique) should be estimated to take no more than 70% of total estimated effort. Otherwise there is high risk of Must Haves not being completed leading to iteration failure. [CC]

#### *Limit Up-Front Design, but Address Technical Risk Early*

1. Big Up-Front Design (BUFD) is an anti-pattern. Ideally, there should be very short cycles (minutes—not days) between design-code-test activities on a given story/task. The developer should work with other members of the team (potentially including specialists in data engineering, testing, security, etc. and/or pair programming) for a story from concept to implementation. Avoid “over-the-wall” communication. [TH]
2. Concentrate on designing, validating, documenting, and building a solid architecture in the early iterations (address technical risk early). [4i]
3. The team should strive to strike a balance of using the early iterations to address the following areas. It is important that none of these areas are neglected in the early iterations—for example, the team should not “perfect” the build environment to the detriment of actually producing working, valuable features.
4. Create the development and test infrastructure (continuous integration, automated build and test, etc.)
5. Address high technical risk areas (“architecturally-significant”)
6. Produce high-value business features [4i]

### **Decompose Selected Features into Tasks**

[no agile-specific guidance]

### **Estimate Tasks**

[no agile-specific guidance]

### **Finalize Sprint Backlog**

[no agile-specific guidance]

### **Commit to Tasks**

[no agile-specific guidance]

### **Manage Sprint**

[no agile-specific guidance]

### **Facilitate Daily Scrum Meeting**

[no agile-specific guidance]

### **Manage Issues**

[no agile-specific guidance]

### **Track Sprint Status**

#### *Use a Burndown Chart to Track the Iteration*

1. Utilize a Burn-Down chart to track work remaining in the iteration. Update at least every other day. Publish in a location easily accessible to the team (SharePoint, wiki, etc.). [4i]
2. For projects with longer iteration lengths (> 3 weeks), use mini-milestones (McConnell; also called "inch-pebbles"); not an unmanageable 1500 line project plan. [4i]
3. Successful agile delivery requires constant feedback and correction. The most useful feedback (that which can result in timely course correction) is obtained from small batches in short cycles. Try to optimize the size (smaller is better) and frequency (more often is better) of feedback cycles. Examples:
4. HOURLY: break-down features and stories into small bite-sized pieces, develop and integrate code, run builds, monitor code quality
5. DAILY: run acceptance tests on the build, collect iteration status, update the iteration burn down chart
6. WEEKLY: develop in short iterations (1-4 weeks), reflect on process (Iteration Retrospective Report), update the release burn up chart
7. MONTHLY: release in small, regularly-scheduled increments and reflect on the business value (ROI) and quality of results (Release Retrospective Report)
8. QUARTERLY: periodically reflect on the project's goals (Product Vision) and priorities (Feature Log) [4i]
9. Do not react too quickly at the beginning of an iteration if things seem to be “off” a bit. This is typical. Don’t micro-manage the iteration. [4i]
10. However, don’t wait too long, either. If a story looks like it won’t get completed, try to split it or defer it to the next iteration. [4i]
11. Do not count partially-done stories towards iteration completion. If this results in wild swings in completed story points from iteration-to-iteration, it is generally an indication that the velocity is being set too high. Notch the velocity down until the delivery rate is consistent and realistic. [TH]

### **Prepare for Future Sprints**

[no agile-specific guidance]

### **Communicate Sprint Status**

[no agile-specific guidance]

### **Develop Features**

#### *Avoid Multi-Tasking and Limit WIP*

1. Individuals are more productive when single-tasking. Similarly, teams are more productive when limiting work-in-progress (WIP). Try to complete individual tasks and stories before moving on to the next one in the iteration. This applies to both individuals (don’t be working on too many tasks simultaneously) and teams (everyone on the team should be working on a few high-priority stories before moving on to the next). Of course, there may be blocks or waiting periods where a developer might get a head start on a future story, but unnecessary multi-tasking should be avoided. Developers who are accustomed to waterfall methods are tempted to start with requirements analysis on S1, S2, and S3 in parallel. They then move on to coding, then finalize with testing. Such an approach is not optimal because there is too much WIP in the iteration pipeline (testers are sitting around waiting for *something* to be ready to test) and risk is increased because many stories are not “done” until the very end of the iteration. [TH]
2. Do your job. Focus all your efforts and skills on doing the work that you’ve committed to doing. Don’t worry about anything else. [Scrum “Focus” value]

#### *Tester Involvement in User Story Specification\*\*\**

1. Every user story would have at least one test case to determine the meaning of 'done'.
2. Behaviour-driven development (BDD) is one way to write acceptance criteria. BDD is similar in concept to Test Driven Development (TDD), but while TDD is technically-focused, BDD calls for black-box testing at the user-story level and is implementation-agnostic.
3. Specification-by-example is another technique that can be used in specification of user stories, these examples can be converted to test cases fairly easily.
4. Provide training to testers to help write good acceptance criteria.
5. Testers would work with developers to develop test cases. Testers use the prototype created by the developer early in the process, to further refine the user story and test cases.
6. Testers would keep boundary conditions in mind as they develop test cases during elaboration of user stories
7. Testers would get the test cases reviewed by the product owner as the user stories are being elaborated
8. Testers would perform effort estimation based on user story, and indicate to product owner what is possible to achieve in each scrum cycle. [PE]

#### *Travel Light*

1. Neither Iterative nor Agile implies “no documentation”. Both espouse documentation that is meaningful to the development of the product. Typically, “Iterative” is associated with more prescribed documentation than is “Agile”. [4i]
2. Do not become a “slave to the process” (check-the-box mentality) or overly-concerned with the completeness of intermediate artifacts. When doing anything, ask "why?" Once the purpose has been achieved, STOP. The 80/20 rule almost always applies. Examples:
3. When doing any task, concentrate on finding the 20% of the effort that will give you 80% of the desired results. Do that 20% first.
4. In the early iterations, deliver the 20% of the features that give the client 80% of the value.
5. When writing code, only write enough code to make the tests pass. Avoid ‘gold plating’ or the temptation to over-design for speculative “what if” scenarios that may never come to pass (YAGNI). Not only is ‘gold plating’ a waste, it also increases technical debt.
6. When creating a document, concentrate on the 20% that gives the reader 80% of what he/she needs to know, and then be ready to MOVE ON. “Gold plating” is a negative factor in documentation as well as software. [4i, CC]
7. "You aren't gonna need it" (or YAGNI for short) is the principle that you should not add functionality until it is necessary. Always implement things when you actually need them, never when you just foresee that you \*might\* need them. Not only does this reduce waste, it also keeps complexity to a minimum. XP includes guidance that “the 2nd developer in pays for generalization”—while this may sound like folly to those who espouse anticipatory design, experience shows that the cost of refactoring to a generalized solution is much less than the long-term cost of maintaining unnecessarily complex, abstract, and buggy code. [CC]
8. Consider developing documentation, particularly high-level documents such as an Architecture Notebook [OpenUP] in an iterative fashion, as well. You do not need to document everything up-front--begin with an outline and continue to add detail to the documentation in each iteration as you learn what works and what new team members need to understand in order to be effective. Use Cases should also be developed in a breadth-before-depth manner. Resist the temptation to document more than you truly know (based on experience) during the project. Speculative outlines are ok, but they should remain in a lightweight form until you know the approach is going to work. The idea is that when changes occur, it is easier (and less painful) to reverse-course or do re-work when you are not heavily-invested. Do not equate "documentation" with "certainty"--such thinking can often lead to a painful reality down the road, when it will be much more difficult to change. Some general rules of thumb apply:
9. "Demonstrate—THEN Document"
10. "Detail in the near-term, but outline for the long-term"
11. “Short-term clarity, but long-term vagueness” [4i]

#### *Minimize/Eliminate “Go-Betweens”*

1. Ideally, detailed requirements should come directly from the customer (or Product Owner) to the developer(s) building the feature. This does not preclude business analysts, testers, and other specialists from being present, but this transmission is ideally through face-to-face communication in a single session. Obviously, time-zone difference can complicate this communication and require e-mails or more formal documentation to be passed back-and-forth. The general principle, though, is to remove as many barriers as possible to real-time, direct translation from requirements to code. See Lean Value-Stream Mapping. [TH]
2. Business people and developers must work together daily throughout the project. [Agile Manifesto Principle]

#### *Apply appropriate Engineering Practices to maximize quality*

1. Not all practices have to be applied at once. You can incrementally adopt the most appropriate engineering practices to the context of your project. [CC]
2. The following engineering practices should be considered:
3. Unit Testing (automated)
4. Automated acceptance testing
5. Continuous Integration (CI)
6. Refactoring
7. Test Driven Development (TDD)
8. Pair Programming
9. Collective Code ownership
10. Coding standards
11. YAGNI (You Ain’t Gonna Need It)
12. DTSTTCPW (Do The Simplest Thing That Could Possibly Work)
13. [CC, TH]

### **Design Feature**

#### *Design for Testability and Loose Coupling*

1. Design self-contained components that are easier to build, test, and deploy independently. Note that this is a valuable and often overlooked side-benefit of an agile approach—the compressed development timeframe almost forces the team to create loosely-coupled, independently-testable and deployable components. [TH]

#### *Parameterize for Agility*

1. Parameterize components so that they are easier to test and deploy in different environments. This can often be accomplished easily with dependency-injection frameworks such as Spring. [TH]

#### *Refactor Mercilessly, Use Design Patterns Judiciously*

1. Continuous attention to technical excellence and good design enhances agility. [Agile Manifesto Principle]

### **Design and Develop Test Scenario**

#### *Specify by Example*

1. Consider using examples as a key element of defining user stories (i.e. specification by example). Examples are often easier for users, developers, and testers to frame discussions around. Examples also can form the basis for encapsulating the acceptance criteria in executable unit tests and acceptance tests (FIT). [TH]

#### *Create Executable Test Plans*

1. Frameworks such as xUnit and FIT allow tests to be created in an executable format (code or Excel spreadsheets, respectively) rather than as separate text documents. This allows for faster and more reliable development, as well as a common point of reference for the team. Prefer executable tests to “one-off” text documents. [TH]

### **Develop the Feature**

[no agile-specific guidance]

### **Perform Agile Unit Testing**

#### *Automate Unit Tests in Parallel with Development*

1. Whether or not the team uses Test-Driven Development, every developed story should include a suite of automated unit tests that are executed and verified automatically in the continuous integration build. The team should set a minimum test coverage expectation for checked-in code. [TH]
2. Write unit tests (and preferably ALL automated tests) in the same Sprint that the code is developed. [TH]

#### *Even Legacy Code Should Have Automated Unit Tests*

1. Michael Feathers, in his excellent book “Working Effectively With Legacy Code” defines legacy code as “*code without tests*”. Using this definition, many teams are either writing legacy code today (!) or letting legacy code become an excuse for not using agile techniques such as TDD. There are techniques for writing effective tests against code that was not previously being tested. To the degree possible, agile developers should add tests to legacy code that they are changing or refactoring. [TH]

### **Check-in and Integrate Application Components**

#### *Fix Broken Builds Immediately*

1. Do not allow builds to remain “broken” (compilation and/or automated tests fail) for more than a few hours. Leaving the build broken (even if due to known causes) is a hindrance to continuous integration. [TH]
2. Globally-distributed teams have found use in making a rule that each site should ensure the build is “green” before closing business for the day. Such a rule prevents a situation where a team begins the day’s work trying to figure out why the build is broken. [TH]

### **Conduct Client Review**

[no agile-specific guidance]

# **Test Application**

Perform Integration Testing

#### *Automate Integration Tests Within a Continuous Build Environment*

1. Create automated tests at various levels (unit, component, and integration) that execute as part of the automated build. Ideally, the results of these tests are visible to the team immediately, and will “fail the build” if they fail. This is typically achieved with tools such as Maven, Hudson, FIT, Selenium, and SoapUI. [TH]

Perform System Level Testing

[no agile-specific guidance]

Optimize Application Performance

[no agile-specific guidance]

Review Application with Client

[no agile-specific guidance]

Commit Work

Facilitate Defect Causal Analysis

[no agile-specific guidance]

Conduct Review/Demo

#### *Keep Sprint Review Informal*

1. The sprint review meeting is intentionally kept very informal, typically with rules forbidding the use of PowerPoint slides and allowing no more than two hours of preparation time for the meeting. A sprint review meeting should not become a distraction or significant detour for the team; rather, it should be a natural result of the sprint. [EPF wiki]

Finalize the Sprint

#### *Demonstrate Progress Using a Burn-Up Chart of Completed Stories*

1. At the close of each iteration, create/update a “burn-up” chart that shows the number of completed (“Done Done”) stories (vertical axis) across each Sprint (horizontal access). This chart should demonstrate clear progress towards the agreed-upon milestone dates and is the primary measure of progress for stakeholders and governance organizations. [TH]
2. At the close of each iteration, inform customer stakeholder(s) which features were completed and the value that is/can be derived from them. This keeps them aware that the team is delivering working tested software at the end of each iteration—which is probably one of the key reasons they “went agile” to start with. [TH]

#### *Do Not Count Work Not “Done”*

1. Resist the temptation (especially with inexperienced teams) to get a feature/story "mostly" complete within the iteration and calling it "done". As Scrum expert Kelly Waters says, "Done" means "DONE!" See also Definition of Done. Consider:
2. Splitting a story that will not get completed by the iteration end date (and putting the non-completed story on the Product Backlog).
3. Moving a developer from one story to help on the story that is behind
4. Deferring the story to the next Sprint [4i]

Conduct Technical Reviews

[no agile-specific guidance]

Conduct Retrospective

#### *It’s More Than Just Practices*

1. Successful agility requires not only a different way of *working* (practices), but also a different way of *thinking* (values, principles). It is almost impossible to be successful with agile unless every member of the team is willing to change and be adaptable. Culture is not something that can be influenced directly, so it is ok to focus on practices early on, but at some point the team must start evolving from “*doing* agile” to “*being* agile.” Here is a reliable rule of thumb: A team that is mostly fixated on process, tools, documents, contract compliance, and detailed planning to a higher degree than focus on human interactions, developing working software, customer collaboration, and/or adaption to change is NOT agile (see [Agile Manifesto](http://agilemanifesto.org)). [4i]
2. Agile is different. Doing it successfully requires changes in both mindset and practice. Do not assume that “best practices” in previous projects will be equally successful in an agile context. In fact, “best practices” from a traditional environment can be counter-productive in an agile context, and vice-versa. Best practices from previous agile projects should be used for learning and improvement. Context is important when applying agile best practices. This guidance applies to both management and technical folks. [TH, CC]
3. Agile requires much more than simply adopting its language and mechanics. Agile processes aren’t just lightweight versions of traditional processes, and a team that doesn’t follow a defined development process isn’t an agile team. On the contrary, agile processes are defined by the presence – not the absence – of certain development practices. [CC]
4. Agile processes are characterized by their ruthless focus on the elimination of waste. [CC]

#### *Conduct a Facilitated Full-Team Retrospective after EVERY Sprint*

1. From the retrospective, choose one to three of the highest-impact items to implement on the next iteration. Do not commit to too many changes—agile is best implemented with small course corrections. [TH]
2. If working in a compliance environment (CMMI, ISO 9001), create a document detailing the decisions made. Otherwise simply enact the decisions on the next iteration and/or add to the product backlog. [TH]
3. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly. [Agile Manifesto Principle]
4. The Scrum Team, the Product Owner, and the Scrum Master will all attend, and a neutral outsider will facilitate the meeting. [Scrum EPF]

Release System

[no agile-specific guidance]

Deploy System

[no agile-specific guidance]

Close Down Project

[no agile-specific guidance]

Additional Guidance

Governance and Compliance

#### *Agile Projects Will Affect Governance*

1. Agile Governance is NOT conformance or enforcement of standards. [CC]
2. Governance should not be a hurdle to agility. It should reduce surprises, increase trust and confidence, and help the team execute in line with business strategy. [CC]
3. Governance is a shared responsibility. Standards are developed jointly by project teams and senior technology management; and compliance is enforced via unobtrusive processes. [CC]
4. Business stakeholders should be fully engaged in the Governance process, with an understanding of its value. This becomes easy when Governance is focused on removing obstacles to project success, and not just standards enforcement. [CC]
5. The table below shows areas in which agile changes traditional paradigms. [DL]

|  |  |  |  |
| --- | --- | --- | --- |
| Process | Waterfall Development | Iterative and Incremental | **Agile Development** |
| Measure of Success | Conformance to Plan  (time, budget, scope) | 🡪 | Response to change, working code, delivering value |
| Management Culture | Command and control | 🡪 | Leadership/ collaborative |
| Requirements and Design | Big and up front | 🡪 | Continuous/emergent/ just-in-time |
| Coding and Implementation | Code all features in parallel, test later | 🡪 | Code and unit test, deliver serially [incrementally] |
| Test and Quality Assurance | Big, planned / Test late | 🡪 | Continuous/concurrent/ test early |
| Planning and Scheduling | PERT/detailed/fix scope, estimate time and resource | 🡪 | Two-level plan/fix date, estimate scope |

*Based on chapter 7 of “Scaling Software Agility: Best Practices for Large Enterprises” by Dean Leffingwell*

#### *Agile Projects Have a Different Model of Change*

1. The table below shows areas in which agile impacts traditional models of change (influenced by complex adaptive systems theory). [MC]

|  |  |  |  |
| --- | --- | --- | --- |
| Area of Focus | Waterfall Development | Iterative and Incremental | Agile Development |
| Behavior | Predictable and controllable | 🡪 | Unpredictable and uncontrollable |
| Direction | Determined by a few leaders | 🡪 | Determined through emergence and by many people |
| Cause and Effect | Every effect has a cause | 🡪 | Every effect is also a cause |
| Relationships | Relationships are directive | 🡪 | Relationships are empowering |
| Measure of Value | Efficiency and reliability | 🡪 | Responsiveness to the environment (business challenge) |
| Decisions | Based on facts and data | 🡪 | Based on patterns and tensions |
| Leaders | Experts and authorities | 🡪 | Facilitators and supporters |

*Based on Olson and Eoyang “Facilitating Organization Change”*

#### *Agile Projects Have a Different Measure of Success*

1. Agile project success is to be measured by how well they deliver value (ROI) in a changing business environment. They are not measured by the traditional “delivered on time, on budget, and as specified” definition of success. [DL]

Distributed Teams

We currently do not have enough experience with distributed agile to recommend guidelines. There are some points of guidance incorporated throughout this document (e.g., *Location Considerations for Agile testers*). Additional guidance will be incorporated into a future revision. For additional guidance, please refer to Martin Fowler’s collection of lessons learned. At: <http://martinfowler.com/articles/agileOffshore.html>.

Documentation

#### *Agile Manifesto: Working software over comprehensive documentation*

1. Agile does not mean no documentation. Agile documentation is lean and concise. [CC]
2. Write what is necessary, and no more. [CC]
3. Understand what you actually mean by 'documentation'. Well commented code is good documentation. Well written code with understandable and meaningful variable names, and conditional statements etc., is good documentation. [CC]
4. Write clean code. Developers don't want or need documentation for a system they understand. [CC]
5. Use an appropriate software tool to generate documentation. [CC]
6. Documentation imposes a future burden of maintenance and upkeep. Documentation should describe information that is less like to change. [CC]
7. Printed documentation that is distributed becomes out of date almost immediately. [CC]
8. Incorrect and inaccurate documentation is worse than no documentation. [CC]
9. Understand that requirements documentation go stale over time. [CC]
10. Written documentation does not have to be formal. A photo of a whiteboard taken on a phone, or electronic scan of a piece of paper is often sufficient documentation that can be loaded into SharePoint or wiki. [CC]
11. Documentation that are useful for communication, should be used in the context of actual communication rather than preserve them for posterity. [CC]
12. Agile documentation describe “good things to know”. [CC]
13. Like any other work on an agile project, documentation should have a ROI. Agile documents maximize stakeholder investment. Agile documentation should fulfill a purpose. [CC]
14. The YAGNI (You Ain’t Gonna Need It) principle applies to documentation. [CC]
15. DTSTTCPW (Do The Simplest Thing That Could Possibly Work) for documentation. [CC]

References

|  |  |
| --- | --- |
| Reference | Source |
| 4i | 4i Iterative and Agile Initiative (HP), multiple participants and project experience, 2007-2008 |
| CC | Chris Chan (HP) – Certified Scrum Practitioner, trainer, agile architecture SME, HPES Agile Development Lead |
| CS | Project Clean Sheet (hp.com) |
| DL | Dean Leffingwell “Scaling Software Agility: Best Practices for Large Enterprises” |
| EPF | Eclipse Process Framework Wiki (Scrum, OpenUP, DSDM, XP) |
| HR | Harish Raju (HP), CSM, State of Michigan DMV project |
| MC | Mike Cohn, “Succeeding With Agile” |
| PE | Prabhu Eshwarla – Quality Assurance and Testing, Global Capability Manager |
| SB | Steve Baumer (HP), agile management SME, 4i Initiative lead |
| SW | Shore & Warden “The Art of Agile Development” |
| TC | Teri Chaplin (HP) – CSM, agile PPM SME |
| TH | Terry Healey (HP) – Scrum Master and agile Java SME, AT&T PIMS, VerizonBusiness LMS-J, Purple Tree EARS projects |