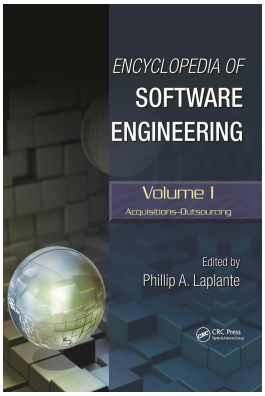

Encyclopedia of Software Engineering

Phillip A. Laplante, William Agresti, Larry Bernstein, Shawn A. Bohner, George Hacken, Mike Hinchey, Tiziana Margaria, Colin J. Neill, Paolo Nesi, Dan Paulish, Raghvinder S. Sangwan, Jing Sun, Jeffrey Voas, Pamela Abbott, William Agresti, Norita Ahmad, Edward Alef, Alain April, Felix Bachmann, Rohit Bahl, Janaka Balasooriya, Larry Bernstein, Fernando Berzal, David Binkley, Dick Brodine, Radu Calinescu, Donald Chand, Kuang-Nan Chang, Ned Chapin, T.Y. Chen, S.C. Cheung, Lawrence Chung, Richard Clayton, Andrea De Lucia, Joanna DeFranco, Maria del Mar Gallardo, Jean Marc Desharnais, Yvonne Dittrich, Scott Donaldson, Christof Ebert, Ghizlane El Boussaidi, Hakan Erdogmus, Letha Etzkorn, John Feminella, Daniel Ferens, Christopher Fox, John Gallagher, Thom Garrett, Swapna Gokhale, Anthony Gold, Márcio Greyck Batista Dias, Vijay Gurbani, George Hacken, Richard Halterman, John Harauz, Ed Harcourt, Eric Hehner, Robert M. Hierons, Mike Hinchey, Jonathan Holt, Caroline Howard, Idris Hsi, Craig Jacobs, Ivar Jacobson, Paul C. Jorgensen, Michael Joy, Ronald K. Kandt, Mira Kajko-Mattsen, Gregory M. Kapfhammer, Rick Kazman, Jon Kern, Thomas Kühne, Stan Kurkovsky, Kathy Land, Gerard Lyons, Jeff Maddalon, Frank Maginnis, Aditya Mathur, James McDonald, Susan Mengel, James Moore, Judith Myerson, Kuang-Nan Chang, Colin Neill, Paolo Nesi, Gunnar Overgaard, Srini Ramaswamy, Franz Rammig, Hassan Reza, David Rico, Michelle Rogers, Colette Rolland, Dieter Rombach, Chris Rouff, Motoshi Saeki, Arthur Salwin, Raghvinder S. Sangwan, Vibha Sazawal, Jim Schiel, Annie Shebanow, Onkar Singh, Paul Solomon, Thanwadee Sunetnanta, Yuen Tak Yu, Richard Turner, Jeffrey Voas, Ellen Walker, Chuck Walrad, Yong Wang, Jonah Weber, Roel Wieringa, Bernhard Westfechtel, Linda Wilbanks, Marcus Wolf, Brendan Wright, Tao Xie, Thomas Zimmerman

Agile Software Development

Agile Software Development

Scott W. Ambler
Chief Methodologist, Agile Development, IBM Rational Software, Toronto, Ontario, Canada

Abstract
Agile software development is an evolutionary, highly collaborative, quality-focused approach to software development where potentially shippable working software is produced on a regular basis. Agile software development processes include Scrum, Extreme Programming (XP), Open Unified Process (OpenUP), and Agile Data (AD), to name a few. Agile approaches are being applied to web applications, mobile applications, fat-client applications, business intelligence (BI) systems, and even mainframe applications. They are being applied by a range of organizations, including but not limited to e-commerce companies, financial companies, manufacturers, retailers, and government agencies. Agile approaches can be scaled for large teams, geographically distributed teams, regulatory situations, complex domains, complex technologies, and to address enterprise issues.

INTRODUCTION

Agile software development is an evolutionary, highly collaborative, quality-focused approach to software development where potentially shippable working software is produced on a regular basis. Agile software development processes include Scrum, Extreme Programming (XP), Open Unified Process (OpenUP), and Agile Data (AD), to name a few. Agile approaches are being applied to web applications, mobile applications, fat-client applications, business intelligence (BI) systems, and even mainframe applications. They are being applied by a range of organizations, including but not limited to e-commerce companies, financial companies, manufacturers, retailers, and government agencies.

Agile software development techniques have taken the industry by storm, with 76% of organizations reporting in mid-2009 that they were adopting agile techniques, and that on average 44% of their project teams were doing so.[1] Agile is becoming widespread because it works—organizations are finding that agile project teams, when compared to traditional project teams, enjoy higher success rates, deliver higher quality, have greater levels of stakeholder satisfaction, provide better return on investment (ROI), and deliver systems to market sooner.[2] But, just because the average agile team is more successful than the average traditional team, that does not mean that all agile teams are successful nor does it mean that all organizations are achieving the potential benefits of agile to the same extent. That is because every organization is unique, with its own goals, abilities, and challenges.

There is no official definition of agile software development. Due to the culture of the agile community, it is unlikely that there ever will be one "official" definition for agile software development. Working with dozens of organizations around the world, I have found the following definition to be effective: *Disciplined agile delivery is an evolutionary (iterative and incremental) approach that regularly produces high-quality solutions in a cost-effective and timely manner via a risk- and value-driven life cycle. It is performed in a highly collaborative, disciplined, and self-organizing manner within an appropriate governance framework, with active stakeholder participation to ensure that the team understands and addresses the changing needs of its stakeholders. Disciplined agile delivery teams provide repeatable results by adopting just the right amount of ceremony for the situation that they face.*

THE AGILE ECOSYSTEM

An ecosystem is a community whose members benefit from each other's participation via symbiotic relationships. The "agile ecosystem" is the community of software professionals who are actively applying, and better yet improving upon, what we call agile software strategies and techniques. This section begins with an overview of the seminal "Agile Manifesto,"[3] it then presents criteria to help you determine if a team is agile, describes how agile is different than traditional approaches, and describes how agile is similar to traditional approaches.

Agile Manifesto

The agile ecosystem started in February 2001 when 17 experienced software practitioners met in Snowbird, Utah to discuss how to effectively develop software-based systems.[4] Out of that meeting came what is now called the Agile Manifesto, which states

Encyclopedia of Software Engineering DOI: 10.1081/E-ESE-120044477

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

1. *Individuals and interactions **over** processes and tools*
2. *Working software **over** comprehensive documentation*
3. *Customer collaboration **over** contract negotiation*
4. *Responding to change **over** following a plan*

That is, while there is value in the items on the right [of the "over"], we value the items on the left more.

A good way to regard the manifesto is that it defines preferences, not alternatives, encouraging a focus on certain areas but not eliminating others. The interesting thing about these value statements is they are something that almost everyone will instantly agree to, yet will often abandon in practice. Senior management will always claim that its employees are the most important aspect of their organization, yet insist they follow bureaucratic processes and treat their staff as replaceable assets. Everyone will readily agree that the creation of software is the fundamental goal of software development, yet insist on spending months producing documentation describing what the software is and how it is going to be built instead of actually building it.

The values of the Agile Manifesto are supported by a collection of 12 principles,[5] which explore in greater detail the philosophical foundation of agile software processes. These principles are as follows:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Criteria for Being Agile

One challenge, which many organizations face, is that project teams will claim to be agile, yet management, who often has little or no experience with agile approaches, cannot tell if the claims are true and just overzealous (I am being polite). Ad hoc teams, those that are not following any acknowledged process, and agile teams alike will point to the Agile Manifesto and claim that they are both agile. Yet to experienced agilists, it is very clear that the ad hoc teams are not agile at all. I have yet to discover an ad hoc development team that met all five, and rarely even two or three, of the following criteria for agile development teams. Furthermore, I have yet to discover an agile team that did not meet the criteria, or at least were well on their way to meeting them. An agile team should take care of the following:

1. **Produce working software on a regular basis**. This is one of the 12 principles behind the Agile Manifesto, and in my experience is a critical differentiator between the teams that are agile and those that are merely claiming it. Ideally, the team should produce potentially shippable software at each iteration (an iteration is also known as a time box or sprint). That does not mean that they will deploy the system into production, or the marketplace, at each iteration but they could if required to do so. Typically the team will deploy into a preproduction testing environment or a demo environment at the end of each iteration (or more often for that matter).
2. **Do continuous regression testing, and better yet, take a test-driven development (TDD) approach**. Agile developers test their work to the best of their ability, minimally doing developer regression testing via a continuous integration (CI) strategy and better yet do developer-level TDD. Agile practices, including both TDD and CI, are described in greater detail below.
3. **Work closely with their stakeholders, ideally on a daily basis**. A common practice of agile teams is to have an on-site customer or product owner who prioritizes requirements and provides information on a timely manner to the team. Disciplined agile teams take it one step further and follow the practice of active stakeholder participation where the stakeholders get actively involved with modeling and sometimes even development.

4. **Be self-organizing within an appropriate governance framework**. Agile teams are self-organizing, which means that the people doing the work determines how the work will be done, they are not told by a manager who may not even be directly involved with the work how it will be done. In other words the team does its own planning, including scheduling and estimation. Disciplined agile teams are self-governing within an effective governance framework, ideally a lean development governance framework.[6]
5. **Regularly reflect on how they work together and then act to improve on their findings**. Most agile teams hold a short meeting at the end of each iteration to reflect upon how well things are working and how they could potentially improve the way that they are working together. Sometimes this is done in a more formalized manner in the form of a retrospective,[7] but often it is done informally. The team then acts on one or more of their suggested improvements at the next iteration. Disciplined agile teams take this one step further and measure their software process improvement (SPI) progress over time.

How Agile Is Different

Knowing how agile software development differs from traditional approaches to software development can help you to understand agile better. There are several key differences:

1. **There is a greater focus on collaboration**. Agilists prefer more effective means of communication, such as face-to-face conversation as opposed to documentation.[8] They also insist on stakeholders being actively involved throughout the entire project, not just during a requirements phase and an acceptance testing phase.
2. **There is a greater focus on quality**. Agilists strive to have a full regression test suite for their systems, which they run on a regular basis, often several times a day. They develop loosely coupled and highly cohesive architectures and will refactor[9] their code whenever they need to in order to keep them this way.
3. **There is a greater focus on working software**. One of the principles behind the Agile Manifesto states that the only valid measure of progress on a software development project is the production of working software. By producing potentially shippable working software on a regular basis at each iteration, they provide greater visibility to their stakeholders as to the status of their project. This provides more opportunities for concrete feedback, increasing the ability and frequency of the stakeholders to steer the project, thereby increasing the chance that the team will produce a system that meets their actual needs.[2] A side effect of increasing visibility in this manner is that there is less need for (expensive) detailed specifications and the bureaucracy surrounding them.
4. **Agilists are generalizing specialists**. A generalizing specialist is someone with one or more specialties, such as programming or database administration, at least a general knowledge of the overall software process, a knowledge of the domain that they are working in, and most importantly a desire to improve their skillset.[10] Teams composed of generalizing specialists require less people to accomplish the same goals, there are less hand-offs between those people, and they work together more effectively because they have more in common.
5. **Agile software development is based on practice**. The original 17 people discussed what worked in practice whereas many traditional strategies reflect how we wished things worked in theory. Big difference.

How Agile Is the Same

The "software engineering laws of physics" still apply to agile teams, although the software engineering community is still identifying what those laws actually are. The following issues appear to apply to a software development project regardless of the paradigm being followed:

1. **We cannot go it alone**. Our stakeholders—including end users, operations staff, infrastructure architects, business leaders, and more—need to be actively involved with a software development project. Agilists are very clear about this importance of this issue but in practice often discover that senior management does not provide adequate support for the concept, thereby putting the project at risk.
2. **People and organization are critical**. Since the 1970s we have had a myriad of technologies, tools, and strategies paraded before us promising to solve our problems yet they have had negligible impact on our productivity.[11] The primary determinant of success on software development project has always been, and very likely will always be, people and the way that they work together.[12]
3. **The project must seem feasible**. In order to gain project funding someone is going to ask the team fundamental questions such as how much is it likely to cost, what are you going to produce, how long is it likely to take, do you have a strategy to build it, will you be able to keep it going once it is in production, and what is the expected ROI? The implication is that you will need to do at least some "up-front" work before construction of the system can commence.
4. **Project time lines are linear**. At best, iterative processes such as Scrum,[13] Unified Process,[14] and Extreme Programming[15] are serial in the large and iterative in the small.[16] What I mean by serial in the

large is that projects have distinct "seasons" in their life cycle—at the beginning of the project basic initiation activities such as initial modeling and obtaining resources must occur, throughout the middle of the project construction activities occur, and at the end of the project deployment activities occur. What I mean by iterative in the small is that each day team members may iterate back and forth between several related activities.

5. **Development is only a part of the overall picture**. Not only do we develop systems, we must also operate and support them once in production. Long before the project starts someone must have envisioned the need for the project, explored the idea to some extent, and garnered support for the project. The point is that development is only one small part of the overall system life cycle.
6. **Systems must fit into a larger environment**. Most organizations have several development projects underway, have many systems in production, and must manage accordingly. Disciplined development teams understand that their work must fit into their organization's overall IT environment, conforming to their enterprise architecture vision, take advantage of reuse opportunities, conform to governance protocols, and so on.[17]

AGILE SOFTWARE PROCESSES

To understand agile software development it is critical to understand agile software processes and practices. First, agile software processes. A software process, often referred to as a software method, is a collection of activities or practices, potentially bound together by a life cycle, which results in the creation of, or evolution of existing, software. Not all agile processes address the full delivery life cycle from end to end, or if they do cover the full life cycle they may not encompass all of the activities required to deliver a system. These "partial" processes are often designed specifically to be combined with other partial processes to form a more robust strategy (more on this later).

"Full" agile processes include, but are not limited to the following:

- **Dynamic System Development Method (DSDM)**. DSDM is an agile delivery process originally based on Rapid Application Development (RAD), which is often used to develop user interface-intensive application.[18]
- **Eclipse Way**. The Eclipse Way is an agile delivery process for distributed development teams. Eclipse Way is practices-based and has a life cycle with a 1 year release cycle that has explicit start-up, construction, and release (end-game) phases.[19]
- **Harmony/ESW**. Harmony is a model-based systems engineering (MBSE) process built on the Unified Modeling Language (UML) and the Systems Modeling Language (SysML). Harmony addresses both systems engineering and software development, integrating the two together.[20]
- **Outside-In Development (OID)**. OID is a software delivery process that is based on the philosophy that to create successful software you must have a clear understanding of the goals and motivations of your stakeholders, the ultimate goal being to produce highly consumable systems that meet/exceed the needs of your stakeholders.[21]
- **Unified Process (UP)**. There are several common flavors of the UP, including Rational Unified Process (RUP),[14] Agile Unified Process (AUP),[22] and OpenUP.[23] The AUP life cycle is depicted in Fig. 1, showing explicit phases for starting the project, proving the architecture, building the system, and releasing the system into production.

"Partial" agile processes include, but are not limited to the following:

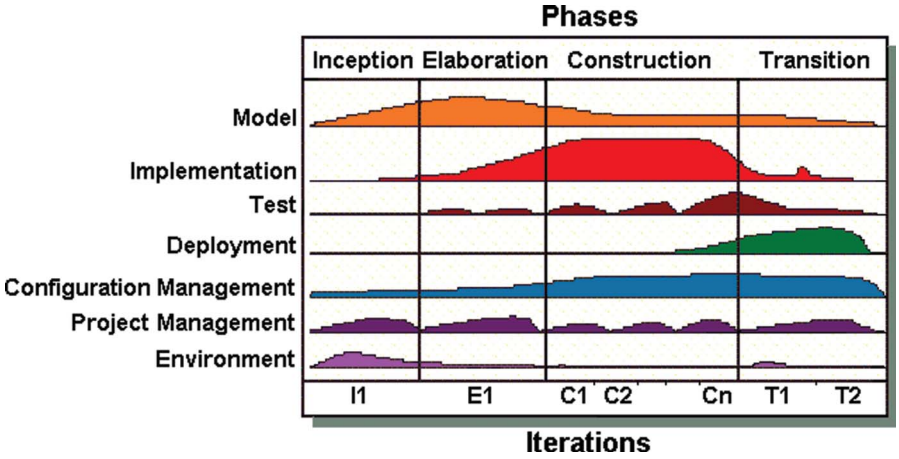

Fig. 1 AUP life cycle.

- **Agile Data**. AD is a collection of practices, such as database refactoring and database testing, for database development on an agile project.[24]
- **Agile Modeling (AM)**. AM is a collection of practices, such as architecture envisioning and prioritized requirements, for light-weight modeling and documentation on an agile project.[25] The Agile Model-Driven Development (AMDD) life cycle for projects, there is also one for enterprise-level modeling,[17] is shown in Fig. 2.
- **Crystal Clear**. Crystal Clear is a highly optimized way to use a small (typically 2–8 people), colocated team, prioritizing for safety in delivering a satisfactory outcome, efficiency in development, and habitability of the working conventions.[26]
- **Extreme Programming (XP)**. XP is a disciplined collection of practices and a life cycle for software construction. Fig. 3 depicts the practices of XP, which are organized into three categories—entire team, development team, and developer.[27]
- **Feature-Driven Development (FDD)**. FDD is a model-driven, short-iteration agile software delivery process that has explicit initiation and construction phases.[28]
- **Scrum**. Scrum, the life cycle for which is shown in Fig. 3, focuses on stakeholder-facing activities around project leadership and requirements management.

It is interesting to observe the differences between the four life cycles. The AUP life cycle of Fig. 1 is represented as a set of disciplines, such as modeling, implementation, and testing spread out over a series of phases but the details of the supporting practices, activities, and work products are not explicitly depicted. The AMDD life cycle of Fig. 2 is shown as several practices, such as initial requirements envisioning and iteration modeling, which are organized into iterations. Phases are implied—Iteration 0 maps to the AUP's Inception phase and Iterations 1 through N to the AUP's Elaboration and Construction phases—and artifacts are not shown. XP is depicted as a collection of practices in Fig. 3, focused on construction-oriented activities but missing any discernible activities around project initiation or releasing the system into production. In this case, neither phases nor artifacts are shown. The Scrum life cycle of Fig. 4 depicts high-level activities that process project artifacts such as the prioritized requirements stack (the product backlog) into the creation of working software. The way that the Scrum life cycle is depicted is the closest to a standard process diagram, showing major activities/processes and the flow of artifacts between them, although it is still very different than the standard process diagrams that traditional IT professionals are accustomed to.

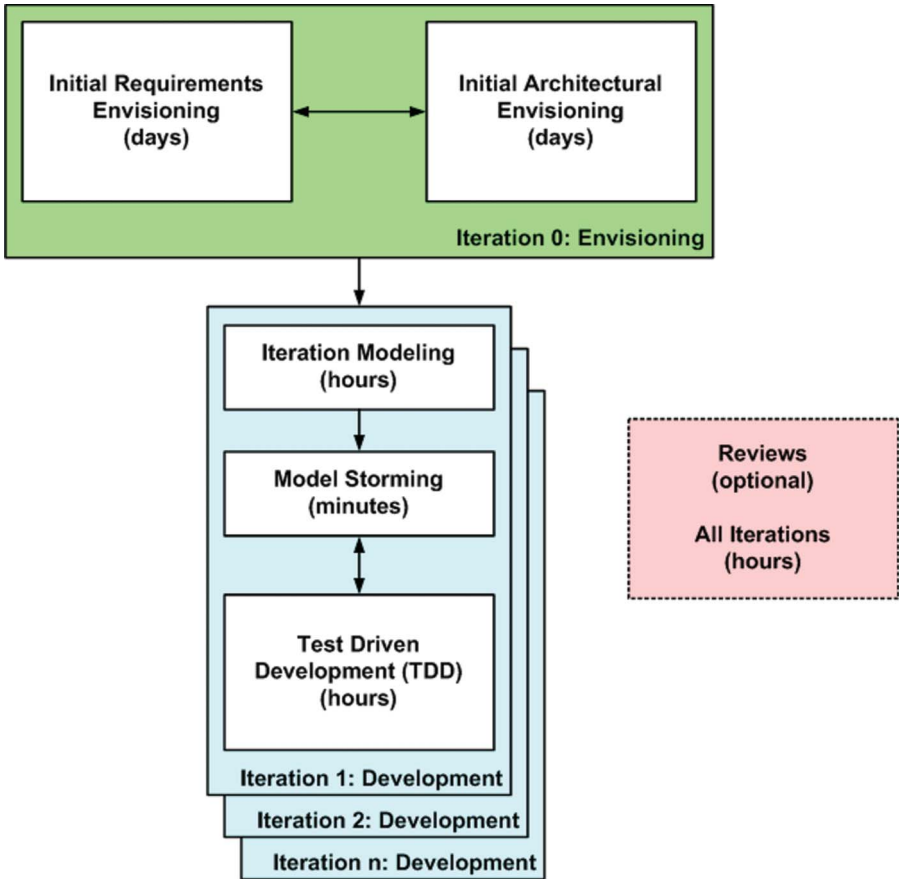

Fig. 2 The Agile Model-Driven Development project life cycle.

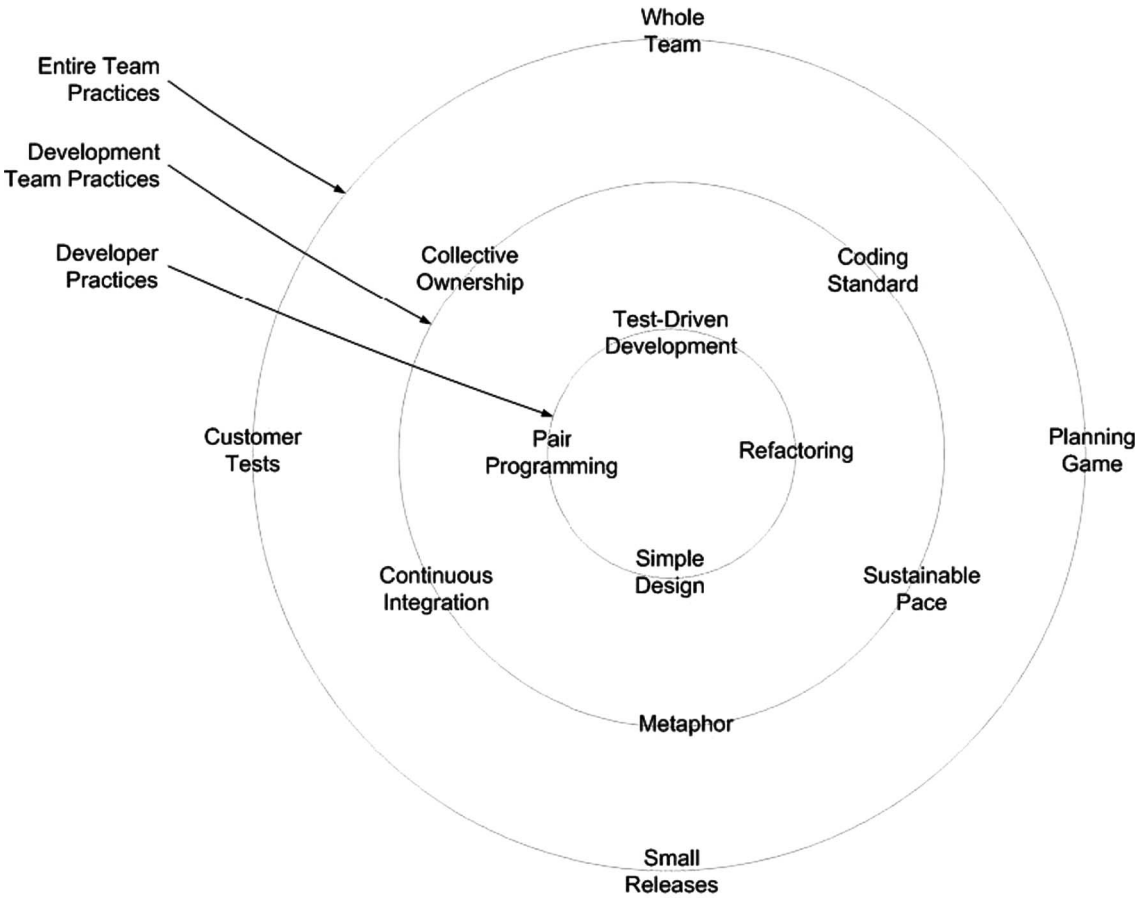

Fig. 3 The practices of Extreme Programming.

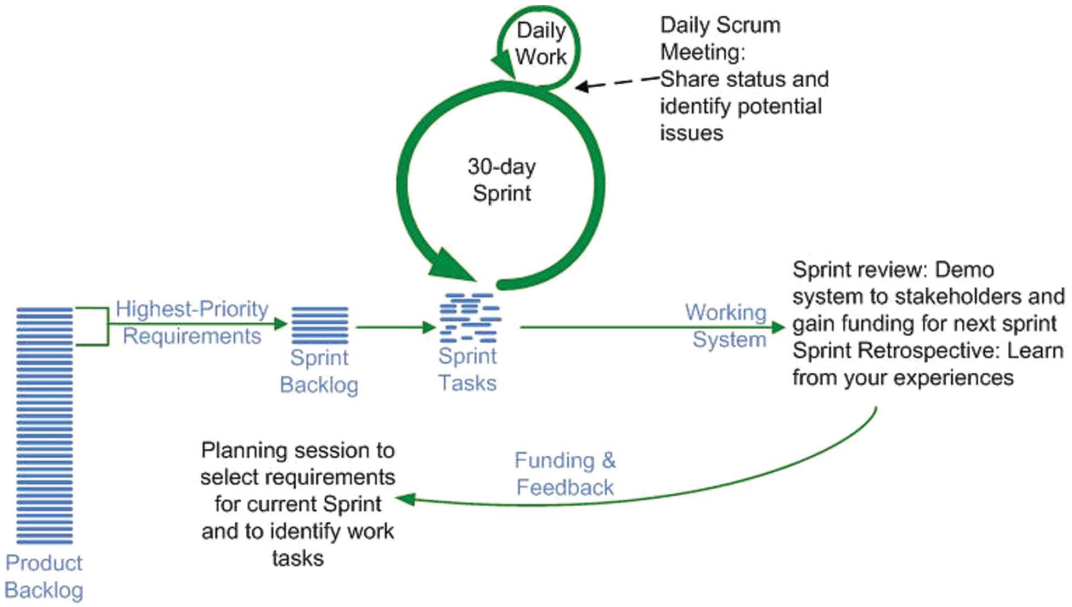

Fig. 4 Scrum construction life cycle.

The point is that although agile processes are depicted in different ways, often in "non-standard" ways and inconsistent ways, they can and should be combined and tailored to meet the unique situation that a project team finds itself in. The styles can be applied between life cycles. For example, Wells[29] shows the XP life cycle in a similar style to Fig. 4, which I extended in Agile Modeling[25] to include the phases described by Kent Beck in *Extreme Programming Explained*.[15] At http://www.agilemodeling.com, I depict AMDD as a collection of practices using a practices map similar to that of Fig. 3. So, do not let the various depiction strategies distract you from the strategies captured by the processes.

AGILE TEAM ORGANIZATION

Agile teams are often organized in a different manner than traditional teams. First, agile teams typically adopt XP's whole team practice (more on this later) and strive to have sufficient skills within the team itself to get the job done. The implication is that the development team has the requisite testing skills, database skills, user interface skills, and so on and does not rely on external experts or teams of experts for these sorts of things. Second, agile teams are formed of generalizing specialists,[10] sometimes called craftspeople, who work closely with one another instead of passing artifacts back and forth between smaller groups of specialists. A generalizing specialist is someone who has one or more technical specialties (e.g., Java programming, project management, database administration) so that they can contribute something of direct value to the team, has at least a general knowledge of software development and the business domain in which they work, and most importantly actively seeks to gain new skills in both their existing specialties as well as in other areas, including both technical and domain areas. Obviously, novice IT professionals, and traditional IT professionals who are often specialized in just one area, will need to work toward this goal. Generalizing specialists are the sweet spot between the two extremes of specialists, people who know a lot about a narrow domain, and generalists who know a little about a wide range of topics.

There are several roles, which have different names depending on the methodology being followed, common to agile teams. Roles are not positions; any given person takes on one or more roles and can switch roles over time, and any given role may have zero or more people in it at any given point in a project. The common agile roles are the following:

- **Team coach**. This role, called "Scrum Master" in Scrum or team lead in other methods, is responsible for facilitating the team, obtaining resources for it, and protecting it from problems. This role encompasses the soft skills of project management but not the technical ones such as planning and scheduling, activities that are better left to the team as a whole (more on this later).
- **Developers**. This role, sometimes referred to as programmer, is responsible for the creation and delivery of a system. This includes modeling, programming, testing, and release activities, to name a few.
- **Stakeholders**. A stakeholder is anyone who is a direct user, indirect user, manager of users, senior manager, operations staff member, or the "gold owner" who funds the project, support (help desk) staff member, auditors, your program/portfolio manager, developers working on other systems that integrate or interact with the one under development, or maintenance professionals potentially affected by the development and/or deployment of a software project.
- **Product owner**. The product owner, called on-site customer in XP and active stakeholder in AM, is the one person responsible on a team (or subteam for large projects) who is responsible for the prioritized work item list (called a product backlog in Scrum), for making decisions in a timely manner, and for providing information in a timely manner.

At scale we start to see other roles become apparent. Once a team grows to 15 or more people it is common to organize it into a team of teams, as we see in Fig. 5, and introduce new roles. Agile project teams of 200 or more have reported success,[1] and at the time of this writing IBM currently has agile programs of over 500 people working on very complex systems. On large agile teams you need to coordinate project management activities (at scale the technical aspects of project management require more than just self-organization), technical/architectural issues, and requirements/product ownership issues (there will be requirements dependencies between subteams and you need to minimize overlapping work between subteams). Furthermore, you often require a "supporting cast" of specialists to assist the generalizing specialists on the subteams. The additional roles on agile teams at scale include the following:

- **Architecture owner**. This person is responsible for facilitating architectural decisions on a subteam and is part of the architecture owner team that is responsible for overall architectural direction of the project. The architecture owner leads their subteam through initial architecture envisioning for their subsystems and will be involved with the initial architecture envisioning for the system as a whole (as part of the architecture owner team). Architecture owners are different than traditional architects in that they are not solely responsible for setting the architectural direction but instead facilitate its creation and evolution.
- **Technical experts**. At scale the need for technical experts in areas such as system integration, database development, user interface development, security,

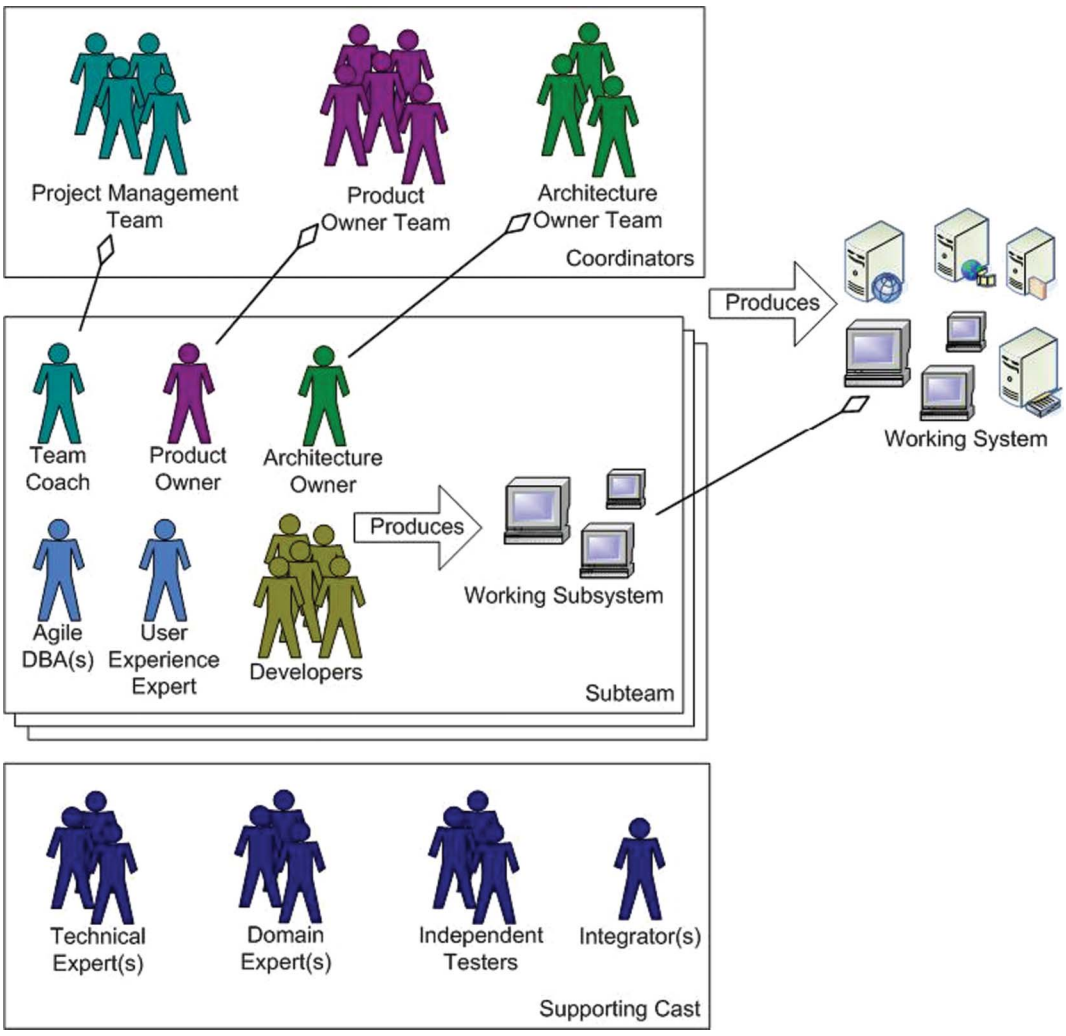

Fig. 5 Organizing a large agile team.

and so on become critical because the problems being addressed at scale are often more complex. Fig. 5 shows the expert roles of Agile database administrator (DBA) and user experience (UE) experts as examples. Sometimes technical experts will be brought in as needed for specific tasks although some experts—particularly Agile DBAs, UE experts, and technical writers—will be required throughout most of the project.

- **Domain experts**. It is unrealistic for a single product owner to represent the entire range of stakeholders in detail. Product owners will need to bring domain experts into the team for short periods of time to focus on specific details with the team.
- **Independent testers**. A common practice on agile teams at scale is to have a separate, independent testing team working in parallel that addresses more complex forms of testing such as investigative testing, system integration testing, security testing, and usability testing. Agile "whole teams" may not have the skills, mindset, or resources to adequately address these advanced testing issues.[30]
- **Integrators**. The subteams are typically responsible for one or more subsystems, and the larger the overall team generally the larger and more complicated the system being built. In these situations the overall team may require one or more people in the role of integrator who are responsible for building the entire system from its various subsystems. These people often work closely with the independent test team, if there is one, who will want to perform system integration testing regularly throughout the project.

AGILE PRACTICES

Software processes often garner a lot of the fervor within the agile community, but in the end the heart of agile software development is in its practices. This section summarizes common agile practices and assigns them into categories. There are several critical observations to be made first:

1. Some practices fall into two or more categories, for example, *Common Development Guidelines* is arguably both a development and a quality practice, but are only listed once.
2. The goal of this categorization is simply to provide organization to this section and is not meant to be a rigorous taxonomy.
3. This list is not meant to be complete—there are hundreds of practices not just the dozens that I present here—although it is meant to be representative of the wealth of material out there.
4. It is unlikely that you will adopt all of these practices, and if you will then you certainly will not adopt all of them right away. Instead, a project team should adopt the right practices for the situation at hand.

Team Practices

Team practices focus on the way that the team organizes itself, how team members interact with one another, and how they interact with their project stakeholders. These practices include the following:

- **Active stakeholder participation**. Stakeholders should provide information in a timely manner, make decisions in a timely manner, and be as actively involved in the development process through the use of inclusive tools and techniques.
- **Collective ownership**. Collective ownership means that everyone on the team is allowed to change any part of the team's artifacts, including but not limited to source code, tests, documents, models, and plans.
- **Daily stand-up meeting**. The team meets once a day for a short period of time, ideally less than 15 minutes, to share status and identify any issues blocking them from successfully achieving their goals. Members on Scrum teams are asked to answer three questions: What did you do yesterday? What do you think you will do today? What is blocking you? Unfortunately, this approach does not scale very well past 10 team members, so larger teams will just focus on discussing what is blocking them and not on the status information suggested by Scrum.
- **Non-solo development**. With non-solo development approaches, such as XP's pair programming or AM's model with others, two or more people work together on a single activity. This increases the quality, builds team cohesion, enables skill sharing (particularly when the pairs change often), and reduces the need for documentation.
- **Project rhythms**. Activities on an agile project team are often organized into "rhythms" of different lengths—daily, iteration, and release. The daily rhythm encompasses the activities performed each day, the iteration rhythm encompasses the organizational activities for an iteration, and the release rhythm encompasses the organizational activities for a single release of a system.
- **Retrospective**. A process improvement session where four questions are typically addressed: What is working well? What did we learn? What should we do differently next time? What still puzzles us? It is common practice for agile teams to hold a short retrospective, typically no more than an hour, at the end of each iteration to help identify potential process improvements to adopt in future iterations.
- **Self-organization**. The team members are responsible for planning, estimating, and organizing their own work. The people best suited to do technical management tasks such as this are the people who do the actual work, not professional managers.
- **Whole team**. Include everyone on the team with the skills and perspectives required for the team to succeed. Everyone on an agile team contributes in any way that they can (hence the importance of becoming generalizing specialists).

Project Management/Leadership Practices

Project management/leadership practices focus on how the team organizes and manages its work. These practices include the following:

- **Iteration demo**. At the end of the iteration, demo the working system, as it currently works to date, to show explicit progress to key stakeholders.
- **Iteration planning**. During iteration planning the team identifies an iteration's worth of work, based on their velocity, from the top of the prioritized work item list. Team members break down the work into tasks and estimate their cost at a finer level of detail than in release planning.
- **Prioritized work item list**. Agile teams implement requirements in priority order, as defined by their stakeholders, so as to provide the greatest ROI possible. Not only must development teams implement functional requirements, they must also address defects, review the work of other teams, take training, and so on. At scale, the practice of a prioritized product backlog of functional requirements is replaced by a prioritized stack of work items. This is applicable for large teams, distributed teams, or teams working on complex systems. Work item lists are called a Product Backlog in Scrum.
- **Release planning**. The creation and maintenance of a high-level plan by the team and their product owner that reflects the desired scope, team resources, and goals. The release plan will reflect dependencies on other teams, major milestones, and target release window(s). The release plan will be updated regularly throughout the project as the team learns.

- **Risk-/value-driven life cycle**. An extension to the value-driven life cycle approach where major risks are taken into account in the prioritization process so that those risks, both business and technical, may be retired earlier in the life cycle.
- **Small releases**. In each iteration, the team produces potentially shippable, running software that is tested and that delivers business value. The "release" may not necessarily be delivered into production but instead may only be deployed into a demo or preproduction testing environment.
- **Value-driven life cycle**. The product owner prioritizes requirements and the team works on them in priority order, producing potentially shippable software every iteration, which provides measurable business values to stakeholders.
- **Velocity**. The amount of work, typically measured in points, that a team can accomplish in an iteration. At the beginning of a project the velocity is a guess, but throughout the project the velocity is the amount of points fully delivered at the previous iteration.

Requirements and Analysis Practices

Requirements and analysis practices focus on how to work with stakeholders effectively so as to understand their intent. This is so important to agile teams that they will often do so on a daily basis. Agile requirements and analysis practices include the following:

- **Customer testing**. Requirements are validated through detailed customer tests, sometimes called customer acceptance tests or story tests, which are run regressively as part of continuous integration (a development practice).
- **Inclusive models**. To make it easier for project stakeholders to be actively involved with the project team you want to use the simplest, most inclusive modeling tools possible with them. This is typically paper and whiteboards, although very sophisticated stakeholders may be able to work with software-based modeling tools. By keeping it simple you encourage active stakeholder participation and thus increase the chances of effective collaboration.
- **Iteration modeling**. At the beginning of each iteration you will do a bit of modeling as part of your iteration planning activities.
- **Model a bit ahead**. Sometimes requirements that are nearing the top of your prioritized work item list are fairly complex, motivating you to invest some effort to explore them before they are popped off the top of the work item stack so as to reduce overall risk.
- **Model storming**. Throughout an iteration you will model storm on a just-in-time (JIT) basis for a few minutes to explore the details behind a requirement or to think through a design issue.
- **Requirements envisioning**. At the beginning of an agile project you need to do sufficient high-level requirements modeling to understand the scope and identify the initial stack of requirements. This is important on all projects, but particularly critical at scale for large teams or geographically distributed teams.
- **User interface prototyping**. The user interface is the system to most end users, so to explore requirements you will often find the need to create user interface sketches or even functioning software that overviews what the user interface (or a portion thereof) looks like.

Architecture and Design Practices

Architecture and design practices focus on the technical vision and strategy for the system under development. Agile architecture and design practices include the following:

- **Architectural skeleton**. Build a working, end-to-end skeleton or structure for the system early in the project to prove that the architecture works.
- **Architectural spike**. When you run into a technical issue, the solution for which is currently unknown on the team, write just enough code to explore that issue and thereby gain the knowledge required to proceed safely (or to abandon that aspect of your technical vision).
- **Architecture envisioning**. At the beginning of an agile project you will need to perform sufficient high-level architectural modeling, often in the form of sketches, to explore potential technical solutions. This is important for all agile teams but particularly so at scale for large teams or geographically distributed teams.
- **Developer testing**. The design is captured and validated through developer tests, sometimes referred to as unit tests, although this is a misnomer as many developer tests go beyond the traditional meaning of the term "unit test." These tests are run regressively as part of continuous integration (a development practice).
- **Metaphor**. The system metaphor is a simple evocative description of how the system works.
- **Multiple views**. Consider multiple views—process, deployment, data, security, user interface, and so on—when creating the technical vision for the system.
- **Simple design**. Start with a simple technical design, and then strive to always keep it as simple as possible.

Development Practices

Development practices focus on construction activities, particularly programming-oriented ones. Agile development practices include the following:

- **Common development guidelines**. Having a common, usable set of development standards that are easy to understand and to comply to can greatly improve the quality of the systems that you develop. These guidelines may include, but not be limited to, programming guidelines, modeling style guidelines, and data-naming conventions.
- **Continuous deployment**. If the team is globally distributed, or if you are doing independent parallel testing (a testing practice), you often need to automatically deploy portions of your work into different development or testing environments based on certain events (such as successful integration).
- **Continuous integration**. At least once every few hours, preferably more often, you should build your system and run your regression tests.
- **Developer sandboxes**. Developers need their own working environments, called sandboxes, where they can modify the portion of the system that they are building and get it working before they integrate their work with that of their teammates.
- **Test-driven development**. Write a single test, either at the requirements or design level, and then just enough code to fulfill that test. TDD is a JIT approach to detailed specification and a confirmatory approach to testing.

Testing and Quality Practices

Testing and quality practices focus on validation of the artifacts being produced. Agile testing and quality practices include the follwing:

- **Defects are requirements**. Defects found externally to the team, typically as the result of independent parallel testing, end-of-life cycle testing, or from users of the system in production, should be treated like a type of requirement—estimate it, prioritize it, and put it on the work item list.
- **End-of-life cycle testing**. An important part of the release effort for many agile teams is end-of-life cycle testing where an independent test team validates that the system is ready to go into production. If the independent parallel testing practice has been adopted, then end-of-life cycle testing can be very short as the issues have already been substantially covered.
- **Independent parallel testing**. The whole team approach to development where agile teams test to the best of the ability is a great start, but defects surrounding non-functional requirements such as security, usability, and performance have a tendency to be missed via this approach. Furthermore, many development teams may not have the resources required to perform effective system integration testing, resources, which from an economic point of view must be shared across multiple teams. The implication is that you will find that you need an independent test team working in parallel to the development team(s), which addresses these sorts of issues. This is particularly important for large or distributed teams, or teams building complex software.
- **Milestone reviews**. The Dr. Dobb's Journal's 2008 project success survey[2] found that agile teams have a 70% success rate. Because this rate is not 100% it behooves agile teams to review progress to date at key milestone points, perhaps at the end of major project phases or at critical financial investment points (such as spending *X%* of the budget). Milestone reviews should consider whether the project is still viable; although the team may be producing potentially shippable software at each iteration, the business environment may have changed and negated the potential business value of the system.
- **Refactoring**. A refactoring is a simple change to your code that improves the quality of the design without changing the semantics. This practice benefits when teams also follow the continuous integration and TDD practices.
- **Static code analysis**. Static code analysis tools check for defects in the code, often looking for types of problems such as security defects that are commonly introduced by developers, or code style issues. Static code analysis enhances project visibility by quickly providing an assessment of the quality of your code. This is particularly important for large teams where significant amounts of code is written, geographically distributed teams where code is potentially written in isolation, organizationally distributed teams where code is written by people working for other organizations, and any organizations where IT governance is an important issue. Static code analysis can be included as part of your continuous integration efforts.

Documentation Practices

Documentation practices focus on creation, maintenance, and delivery of documentation. Agile documentation practices include the following:

- **Document late**. Write documentation as late as possible, avoiding speculative ideas that are likely to change in favor of stable information. However, this does not mean that all documentation should be left toward the end. You might still want to take notes throughout construction so that you do not lose critical information. By waiting to document information once it has stabilized you reduce both the cost and the risk associated with documentation.
- **Documents are requirements**. Agile teams treat documentation like any other requirement: it should be

estimated, prioritized, and put on your work item list along with all other work items that you must address.

- **Executable specifications**. Specify requirements in the form of executable "customer tests," and your design as executable developer tests, instead of non-executable "static" documentation. When you take a TDD approach, you effectively write detailed executable specifications on a JIT basis.
- **Single source information**. Strive to capture information in one place and one place only, ideally in the most effective place (which in the case of software is often the code). The idea is to record technical information only once and then generate various documentation views as needed from that information.
- **Wall of wonder**. Display critical project artifacts, including project schedules, burn down charts, and important diagrams (both business oriented and technically oriented) in a public manner. For a small, colocated team, this implies building your "wall of wonder" from whiteboards and something to stick paper to, such as corkboards. For large or distributed teams you will need an electronic wall of wonder, such as an internal web site.[31]

Data Practices

Data practices focus on how to address data issues, particularly database development, on agile projects. The primary challenge is how to evolve the database schema in sync with the rest of the overall solution in an efficient manner. The modeling practices mentioned early equally apply to data modeling. AD practices include the following:

- **Database refactoring**. A database refactoring is a small change to your database schema, which improves its design without changing its semantics (e.g., you do not add anything nor do you break anything). The process of database refactoring is the evolutionary improvement of your database schema so as to improve your ability to support the new needs of your customers.
- **Database regression testing**. You should ensure that your database schema actually meets the requirements for it, and the best way to do that is via testing the functionality implemented within the database as triggers and stored procedures as well as testing to ensure data quality.
- **Lean data governance**. The goal of data governance is to ensure the quality, availability, integrity, security, and usability within an organization. The goal of lean data governance is to enable development teams to do these things effectively within your overall IT ecology.

Scaling Practices

Scaling practices focus on how to apply agile strategies on large teams, on distributed teams, and so on (scaling complexity factors are described below). Many of the agile practices mentioned previously can be modified to work at scale, and many, particularly the modeling practices, are arguably scaling practices in their own right. Agile scaling practices include the following:

- **Agile enterprise architecture**. An effective enterprise architecture program can increase developer's productivity by promoting common technology platforms and greater levels of reuse. For enterprise architects to support agile teams, they must work in a collaborative, light-weight, and evolutionary manner consistent with agile approaches.
- **Align team structure with the architecture**. Most of the communication that occurs on development projects occurs between the people building subsystems/subcomponents. The implication is that for large or geographically distributed teams you should organize yourself around the architecture, with each subteam located as close together as possible, so as to minimize the required coordination between teams and thereby minimize overall communication risk on your project. This practice is sometimes referred to as "Conway's Law" after Melvin Conway, who first coined it in the late 1960s.
- **Lean development governance**. The focus of lean development governance is on motivating and enabling the right behaviors among IT professionals, as opposed to traditional approaches that take "command and control" approaches.[6]
- **Portfolio rhythms**. This practice extends the concept of project rhythms to the enterprise level, introducing the concept of a program/package rhythm and a release window rhythm. A program/package rhythm encompasses the release cycle of a collection of systems; for example, Microsoft Office is comprised of several applications such as Word and Excel, where each application project rhythm must reflect the overall package rhythm. A release window rhythm captures the available time slots that project teams are allowed to release software into production, perhaps every third Saturday from 2 A.M. to 6 A.M. It also reflects release blackout dates when you are not allowed to release software into production due to the overall business cycle (e.g., retail organizations in Western nations typically have release blackouts from the middle of November to early January due to the holiday shopping season).

Mapping Practices to Processes

Table 1 maps each of the agile practices described earlier to several of the agile processes described above. An X is

indicated whenever the process includes the practice (or at least something similar) or indicates the name of the practice as it is implemented by the process. Several of the scaling practices are not implemented by any of the processes listed in the table, but instead implemented by other processes such as Eclipse Way or Enterprise Unified Process (EUP).[17]

It is interesting to note from Table 1 that there is an overlap of practices, often because the processes borrowed ideas from each other and because they are based on actual experiences from the "development trenches." For example, the practices product owner (Scrum), on-site customer (XP), and active stakeholder participation (AM) are all variations on the idea that you need to work closely with stakeholders or their representatives. These practices are often used to piece together a (mostly) complete agile software development process, an exercise that can prove difficult, error-prone, and sometimes expensive for organizations that do not have a lot of agile experience (more on this next).

THE AGILE SCALING MODEL

One interesting observation is that the Agile Manifesto was written by a group of people made up predominantly of consultants and developers (and some were both). Although the manifesto reflects their real-world, practical experiences it also reflects their biases. The implication is that the manifesto, while providing an excellent philosophical foundation for agile software development, may not cover the entire gambit required for an agile approach to IT in general. Evidence of this is revealed by the release of the Manifesto for Software Craftsmanship[32] in early 2009, an extension to the Agile Manifesto. This is not to say that this manifesto is sufficient, but it does indicate that some people believe that we require something more than just the ideas captured by the Agile Manifesto and its underlying principles.

Another interesting observation is that there is a range of agile processes, described earlier, some of which more

Table 1 Mapping agile practices to several agile processes.

Practice	Scrum	XP	OpenUP	DSDM	AM	AD
Active stakeholder participation	Product owner	On-site customer	X	X	X	X
Agile enterprise architecture					X	
Align team structure with the architecture			X	X		
Architectural skeleton		X	X		Prove it with code	
Architectural spike		X	X			
Architecture envisioning			X	X	X	X
Collective ownership		X	X		X	X
Common development guidelines		Coding standards	X	X	Apply modeling standards	X
Continuous deployment			X			
Continuous integration		X	X	X		
Customer testing		X	X		Executable specifications	Database regression testing
Daily stand-up meeting	Scrum meeting		X			
Database refactoring						X
Database regression testing						X
Defects are requirements			X	X		
Developer sandboxes		X	X			X
Developer testing		X	X	X	Executable specifications	Database regression testing
Document late		X	X		X	

(Continued)

Table 1 Mapping agile practices to several agile processes. *(Continued)*

Practice	Scrum	XP	OpenUP	DSDM	AM	AD
Documents are requirements		X	X		X	
End-of-life cycle testing			X	X		
Executable specifications		Customer testing, developer testing	X		X	
Inclusive models		X	X	X	X	
Independent parallel testing			X	X		
Iteration demo	Sprint demo	X	X	X		
Iteration modeling					X	
Iteration planning	Sprint planning	Planning game	2-level planning	X		
Lean data governance						X
Lean development governance						
Metaphor		X				
Milestone reviews			X	X		
Model a bit ahead				X	X	
Model storming				X	X	
Multiple views			X	X	X	X
Non-solo development		Pair programming	X		Model with others	Model with others
Portfolio rhythms						
Prioritized work item list	Product backlog	X	X	X	X	
Project rhythms	X	X	X	X	X	
Refactoring		X	X			Database refactoring
Release planning	X	Planning game	2-level planning	X		
Requirements envisioning			Use-Case-Driven Modeling	X	X	X
Retrospective	X		X			
Risk-/value-driven life cycle		X	X	X	X	
Self-organization	X	X	X	X		
Simple design		X	X	X	X	X
Single source information		X	X		X	
Small releases	X	X	X	X		
Static code analysis			X			
Test-driven development		X	X		X	Database regression testing
User interface prototyping			X	X		
Value-driven life cycle	X	X	X	X		
Velocity	X	X	X			
Wall of wonder	X	X	X	X	X	X
Whole team	X	X	X	X		

or less describe how to develop software from beginning to end and others that take on just a portion of the overall system delivery life cycle. Although it is good to have a range of agile processes to choose from, it can be difficult for organizations new to agile to choose between them, or to combine them effectively (or even to recognize that they need to combine them). This challenge is exacerbated by the plethora of agile practices being touted by various practitioners. As a result many organizations find the agile message confusing, in part because of the multitude of voices within the agile community but more so because the agile rhetoric often seems to ignore or gloss over many important issues that organizations face on a daily basis. Some sort of guiding framework is clearly needed.

The Agile Scaling Model (ASM)[33] is a contextual framework for effective adoption and tailoring of agile practices to meet the unique challenges faced by a system delivery team of any size. The ASM distinguishes between three scaling categories: core agile development, disciplined agile delivery, and agility at scale. My experience is that disciplined agile delivery as the minimum that your organization should consider if it wants to succeed with agile techniques—whether you are on a mainframe team writing COBOL code for a bank, software running on millions of mobile phones, or e-commerce code running on the web, your team should still follow a full delivery life cycle in a disciplined manner. You may not be there yet, but still in the learning stages. But our experience is that you will quickly discover how one or more of the scaling factors is applicable, and as a result need to change the way you work.

The ASM distinguishes between three scaling categories:

1. **Core agile development**. Core agile methods, such as Scrum and Agile Modeling, are self-governing, have a value-driven system development life cycle (SDLC), and address a portion of the development life cycle. These methods, and their practices, such as daily stand-up meetings and requirements envisioning, are optimized for small, colocated teams developing fairly straightforward systems.
2. **Disciplined agile delivery**. Disciplined agile delivery processes, which include DSDM and OpenUP, go further by covering the full software development life cycle from project inception to transitioning the system into your production environment (or into the marketplace as the case may be). Disciplined agile delivery processes are self-organizing within an appropriate governance framework and take both risk- and value-driven approach to the life cycle. Like the core agile development category, this category is also focused on small, colocated teams delivering fairly straightforward systems. To address the full delivery life cycle you need to combine practices from several core methods, or adopt a method that has already done so.
3. **Agility at scale**. This category focuses on disciplined agile delivery where one or more scaling factors are applicable. The eight scaling factors are team size, geographical distribution, regulatory compliance, organizational complexity, technical complexity, organizational distribution, domain complexity, and enterprise discipline. All of these scaling factors are ranges, and not all of them will likely be applicable to any given project, so you need to be flexible when scaling agile approaches to meet the needs of your unique situation. To address these scaling factors you will need to tailor your disciplined agile delivery practices and in some situations adopt a handful of new practices to address the additional risks that you face at scale.

Let us explore the differences between the ASM categories. To understand the main differences between core agile development and disciplined agile delivery, compare the full delivery life cycle of Fig. 6[34] with the Scrum construction life cycle of Fig. 4. The normalization of the terminology aside, there are several significant differences:

1. **Explicit project inception**. At the beginning of a project—called the Inception phase, Sprint 0 or Iteration 0—you need to do some initial modeling, start putting together your team, and gain initial project funding. These are important activities that should be explicitly included in the life cycle.
2. **Independent parallel testing**. Agile developers will do a significant amount of testing—either developer regression testing or better yet TDD—as part of their daily jobs. This is a great start, but to ensure that defects do not fall through the cracks disciplined agile teams will often adopt the practice of parallel independent testing.
3. **Prioritized work items**. Not only do agile delivery teams implement functional requirements, they must also fix defects (found through independent testing or by users of existing versions in production), provide feedback on work from other teams, go on training courses, and so on. All of these activities need to be scheduled for, not just functional requirements. Having a single work item stack, instead of several stacks, proves easier to manage in practice for most agile teams.
4. **Explicit transition/release phase**. Releasing software into production can be very complex and risky in practice and your software process needs to reflect this.
5. **Explicit production phase**. Disciplined agile developers recognize that their systems will be operated and supported in production, and that the people doing this work are important stakeholders, and

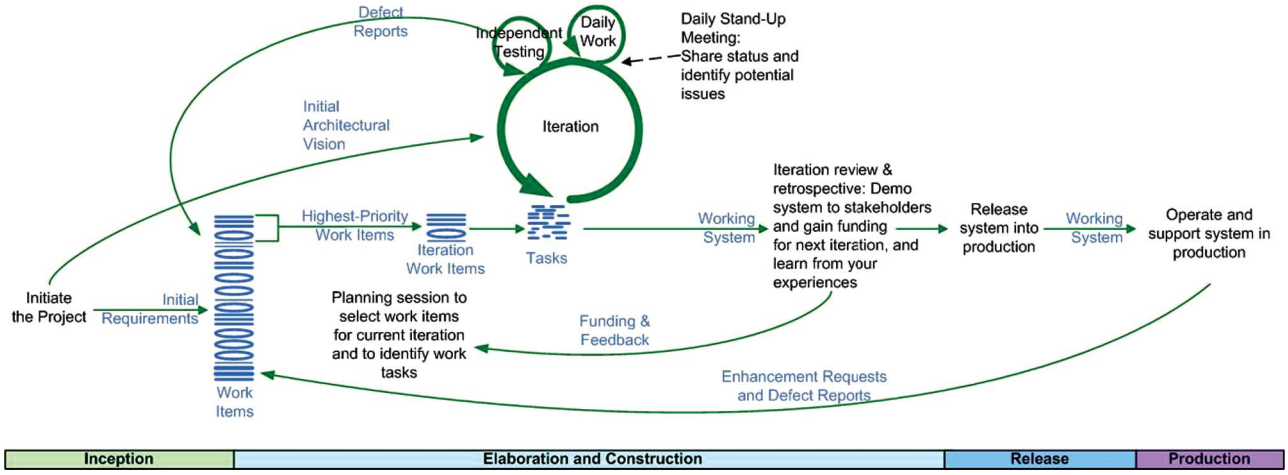

Fig. 6 Agile system development life cycle.

therefore it is critical that your development process explicitly calls this out. Unfortunately at the time of this writing many disciplined agile delivery processes do not include an explicit production phase although will often include operations and support staff as explicit stakeholders. It is a start.

The goal of the ASM's agility at scale category is to make it explicit that the small, colocated team advice of many of the mainstream agile methods falls apart when one or more scaling factors comes into play. These scaling factors, summarized in Table 2, introduce complexities that must be overcome by more sophisticated practices,

Table 2 Potential process scaling factors.

Scaling factor	Issues
Team size	Core agile processes work very well for smaller teams of 10–15 people, but what if the team is much larger? What if it is 50 people? One hundred people? One thousand people? Paper-based, face-to-face strategies start to fall apart as the team size grows.
Geographical distribution	What happens when the team is distributed, perhaps on floors within the same building, different locations within the same city, or even in different countries? Suddenly, effective collaboration becomes more challenging and disconnects are more likely to occur.
Compliance requirement	What if regulatory issues—such as Sarbanes Oxley, ISO 9000, or FDA CFR 21—are applicable? These issues bring requirements of their own that may be imposed from outside your organization in addition to the customer-driven product requirements.
Domain complexity	Some project teams find themselves in situations where the problem domain is complex. Complexities arise as the result of several factors: the nature of the problem, monitoring a biochemical process or managing air traffic control is significantly more complex than developing an informational web site; the requirements change quickly, such as with financial derivatives trading or electronic security assurance; or perhaps the delivery team does not currently have an adequate background in the domain.
Organization distribution	Sometimes a project team includes members from different divisions, different partner companies, or from external services firms. This lack of organizational cohesion can greatly increase the risk to your project.
Technical complexity	There are several sources of technical complexity: existing legacy systems and legacy data sources that are less than perfect; your system involves several technological platforms or several disparate technologies; your system has dependencies on other existing systems or worse yet on systems that are being built in parallel to your own; you are doing a systems engineering project with both hardware and software development.
Organizational complexity	There are several sources of organizational complexity: your existing organization structure, culture, and business processes may reflect waterfall/serial values; your organization struggles to embrace change; your organization has a very rigid leadership style; different subgroups within your organization have different visions for how they should work.
Enterprise discipline	Most organizations want to leverage common infrastructure platforms to lower cost, reduce time to market, and to improve consistency. To accomplish this they need effective enterprise architecture, enterprise business modeling, strategic reuse, and portfolio management disciplines. These disciplines must work in concert with, and better yet enhance, your disciplined agile delivery processes.

or at least more sophisticated versions of practices, and tooling than is often described in core agile development or disciplined agile delivery processes. For example, the Scrum approach of a daily stand-up meeting where everyone answers both status-oriented questions as well as blocking questions works well for small, colocated teams. However, this strategy proves insufficient at scale: larger teams find that they need to focus on only identifying blocking issues and distributed teams need to start applying electronic solutions, something as simple as teleconference calls or a little more complex such as groupware tools, to run their "daily stand-ups" remotely.

Each factor has a range of complexities, and each team will have a different combination and therefore will need a process, team structure, and tooling environment tailored to meet their unique situation. Core agile development processes work best when basically all factors are at the left-hand side (the low complexity side), although they can potentially be tailored to address greater complexity with strategies from higher-level processes. Disciplined agile delivery processes typically assume that one or more of the factors slightly to the right, and scaled agile processes have one or more factors leaning to the right (the high complexity side).

SUMMARY

Many organizations have succeeded at applying agile at scale and you can too. If you keep your wits about you and stay away from some of the rhetoric of the mainstream agile community you should be all right. Minimally, you want a disciplined agile system delivery approach that addresses the full life cycle, not just parts of it. Remember that you will often find yourself in a scaling situation, and that different teams will experience different scaling factors, and that is all right because it is fairly straightforward to scale agile strategies with effective practices and tooling. With a realistic approach to process improvement, and with a bit of help from the outside, you can increase your ROI, quality, stakeholder satisfaction, and time to value through agility at scale.

REFERENCES

1. Ambler, S.W. Dr. Dobb's Journal's July 2009 State of the IT Union Survey. 2009, http://www.ambysoft.com/surveys/stateOfITUnion200907.html (accessed January 2010).
2. Ambler, S.W. Dr. Dobb's Journal's 2008 Project Success Survey. 2008, http://www.ambysoft.com/surveys/success2008.html (accessed January 2010).
3. Beck, K.; Beedle, M.; Van Bennekum, A.; Cockburn, A.; Cunningham, W.; Fowler, M.; Grenning, J.; Highsmith, J.; Hunt, A.; Jeffries, R.; Kern, J.; Marick, B.; Martin, R.; Mellor, S.; Schwaber, K.; Sutherland, J.; Thomas, D. The Agile Manifesto. http://www.agilemanifesto.org 2001 (accessed January 2010).
4. Highsmith, J. History: Agile Manifesto. 2001, http://www.agilemanifesto.org/history.html (accessed January 2010).
5. Beck, K.; Beedle, M.; Van Bennekum, A.; Cockburn, A.; Cunningham, W.; Fowler, M.; Grenning, J.; Highsmith, J.; Hunt, A.; Jeffries, R.; Kern, J.; Marick, B.; Martin, R.; Mellor, S.; Schwaber, K.; Sutherland, J.; Thomas, D. Principles behind the Agile Manifesto. 2001, http://www.agilemanifesto.org/principles.html (accessed January 2010).
6. Ambler, S.W.; Kroll, P. Lean Development Governance, https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?lang=en_US&source=swg-ldg, 2007 (accessed January 2010).
7. Kerth, N. *Project Retrospectives: A Handbook for Team Reviews*; Dorset House Publishing: New York, NY, 2001.
8. Cockburn, A. *Agile Software Development 2nd Edition: The Cooperative Game*; Pearson Education, Inc.: Upper Saddle River, NJ, 2007.
9. Fowler, M. *Refactoring: Improving the Design of Existing Code*; Addison-Wesley Longman: Menlo Park, CA, 1999.
10. Ambler, S.W. Generalizing Specialists: Improving Your IT Skills. 2003, http://www.agilemodeling.com/essays/generalizingSpecialists.htm (accessed January 2010).
11. Jones, C. *Applied Software Measurement: Global Analysis of Productivity and Quality*; McGraw Hill: New York, NY, 2008.
12. Royce, W. *Software Project Management: A Unified Framework*; Addison Wesley Longman, Inc.: Reading, MA, 1998.
13. Beedle, M.; Schwaber, K. *Agile Software Development with SCRUM*. Prentice Hall, Inc.: Upper Saddle River, NJ, 2001.
14. Kruchten, P. *The Rational Unified Process: An Introduction*. 3rd Ed.; Addison-Wesley Professional: Boston, MA, 2003.
15. Beck, K. *Extreme Programming Explained—Embrace Change*; Addison Wesley Longman: Reading, MA, 2000.
16. Ambler, S.W. *The Object Primer, 3rd Edition: Agile Model Driven Development with UML 2*. Cambridge University Press: New York, NY, 2004.
17. Ambler, S.W.; Nalbone, J.; Vizdos, M. *The Enterprise Unified Process: Enhancing the Rational Unified Process*; Addison Wesley: Boston, MA, 2004.
18. Stapleton, J. *DSDM: Business Focused Development*. 2nd Ed.; Addison Wesley: Harlow, U.K., 2003.
19. Frost, R. Jazz and the Eclipse Way of collaboration. IEEE Softw. **2007**, *24* (6), 114–117.
20. Douglass, B.P. *Real-Time Agility: The Harmony/ESW Method for Real-Time and Embedded Systems* Development; Addison-Wesley: Upper Saddle River, NJ, 2009.
21. Kessler, C.; Sweitzer, J. *Outside-In Software Development: A Practical Approach to Building Stakeholder-Based Products*; IBM Press: Upper Saddle River, NJ, 2007.
22. Ambler, S.W. *The Agile Unified Process*. 2005, http://www.ambysoft.com/unifiedprocess/agileUP.html (accessed January 2010).
23. Eclipse Community. *The Open Unified Process (OpenUP)*. 2006, http://www.eclipse.org/epf/ (accessed January 2010).
24. Ambler, S.W. *Agile Database Techniques: Effective Strategies for the Agile Development*; Wiley Publishing: New York, NY, 2004.

25. Ambler, S.W. *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. Wiley Press: New York, NY, 2002.
26. Cockburn, A. *Crystal Clear: A Human-Powered Methodology for Small Teams*; Pearson Education, Inc.: Upper Saddle River, NJ, 2005.
27. Jeffries, R. What Is Extreme Programming? 2001, http://www.xprogramming.com/xpmag/whatisxp.htm (accessed January 2010).
28. Palmer, S.R.; Felsing, J.M. *A Practical Guide to Feature-Driven Development*; Prentice Hall PTR: Upper Saddle River, NJ, 2002.
29. Wells, D. *Extreme Programming: A Gentle Introduction*; 2001, http://www.extremeprogramming.org (accessed January 2010).
30. Ambler, S.W. Agile Testing and Quality Strategies: Discipline over Rhetoric. 2009, http://www.ambysoft.com/essays/agileTesting.html (accessed January 2010).
31. Gottesdiener, E. *Requirements by Collaboration: Workshops for Defining Needs*; Addison-Wesley: Upper Saddle River, NJ, 2002.
32. Manifesto for Software Craftsmanship. http://manifesto.softwarecraftsmanship.org (accessed January 2010).
33. Ambler, S.W. The Agile Scaling Model (ASM): Adapting Agile Methods for Complex Environments. 2009, http://www.ambysoft.com/essays/agileLifecycle.html (accessed January 2010).
34. Ambler, S.W. The Agile System Development Lifecycle (SDLC). 2005, http://www.ambysoft.com/essays/agileLifecycle.html (accessed January 2010).