Agile Software Development at Scale

Scott W. Ambler

Practice Leader Agile Development, IBM Rational
scott_ambler@ca.ibm.com

Abstract: Since 2001 agile software development approaches are being adopted across a wide range of organizations and are now being applied at scale. There are eight factors to consider – team size, geographical distribution, entrenched culture, system complexity, legacy systems, regulatory compliance, organizational distribution, governance and enterprise focus – when scaling agile. Luckily a collection of techniques and strategies exist which scale agile approaches, including considering the full development lifecycle, Agile Model Driven Development (AMDD), continuous independent testing, adopting proven strategies, agile database techniques, and lean development governance. It is possible to scale agile approaches, but you will need to look beyond the advice provided by the "agile in the small" literature.

Introduction

Agile software development is being adopted by both the majority of organizations, a recent survey [1] shows that 69% of organizations are taking agile approaches on one or more projects, and by a wide range of organizations. The same survey also indicated that organizations are attempting large agile projects, several respondents indicated that they were not only doing but successful with agile project teams of over 200 people, and many indicated that they were applying agile in distributed environments. Agile project teams also appear to have higher rates of success than do traditional teams [2] indicating that agile approaches are likely here to stay.

Agile techniques have clearly been proven in simple settings and we're seeing that many organizations are now applying agile at scale. In this paper I explore the factors surrounding apply agile techniques at scale, large or distribute teams are just two of the many issues which agile teams now face, and overview a collection of techniques which I've applied successfully in practice.

Scaling Factors

When you read some of the agile literature it sounds rather naïve at times. Although we would all love nothing more than to work with small, co-located, closely-knit teams of highly-skilled professionals who are building brand new systems it rarely seems to be the case in practice. Instead one or more "scaling factors" seems to ruin this perfect scenario for us. When you think about scaling agile approaches the first factors that you consider are team size and geographical distribution [3], and although these are clearly important scaling factors they're not the only ones. At IBM Rational we have found that when applying agile strategies at scale you are likely to run into one or more of the following complexity factors:

1. Team size. Large teams will be organized differently than small teams, and they'll work differently too. Strategies that work for small co-located teams won't be sufficient for teams of several hundred people.
2. Geographical distribution. Some members of a team, including stakeholders, may be in different locations. Even being in different cubicles within the same building can erect barriers to communication, let alone being in different cities or even on different continents.
3. Entrenched culture. Most agile teams need to work within the scope of a larger organization, and that larger organization isn't always perfectly agile. The existing people, processes, and policies aren't always ideal. Hopefully that will change in time, but we still need to get the job done right now.
4. System complexity. The more complex the system the greater the need for a viable architectural strategy. An interesting feature of the Rational Unified Process (RUP) [4] is that its Elaboration phase's primary goal is to prove the architecture via the creation of an end-to-end, working skeleton of the system. This risk-reduction technique, described later in this paper, is clearly a concept which Extreme Programming (XP) [5] and Scrum [6] teams can clearly benefit from.
5. Legacy systems. It can be very difficult to leverage existing code and data sources due to quality problems. The code may not be well written, documented, or even have tests in place, yet that doesn't mean that your agile team should rewrite everything from scratch. Some legacy data sources are questionable at best, or the owners of those data sources difficult to work with, yet that doesn't given an agile team license to create yet another database.
6. Regulatory compliance. Regulations, including the Sarbanes-Oxley act, BASEL-II, and FDA statutes can increase the documentation and process burden on your projects. Complying with these regulations while still remaining as agile as possible can be a challenge.
7. Organizational distribution. When a team is made up of people working for different divisions, or from different companies (such as contractors, partners, or consultants), then management complexity rises.
8. Degree of governance. If you have one or more IT projects then you have an IT governance process in place. How formal it is, how explicit it is, and how effective it is will be up to you. Agile/lean approaches to governance are based on collaborative approaches which enable teams to do the right thing, as opposed to traditional approaches which implement command-and-control strategies [7]. More on this later.
9. Enterprise focus. It is possible to address enterprise issues, including enterprise architecture, portfolio management, and reuse within an agile environment. The Enterprise Unified Process (EUP) extends evolutionary processes such as RUP or XP to bring an enterprise focus to your IT department [8].

The point is that agile is relative, that different environments will require different strategies to scale agile approaches effectively. This implies that you need to have a collection of techniques at your disposal.

Strategies for Scaling Agile Approaches

It is not only possible to scale agile software development approaches, the strategies to do so already exist. These strategies are:

- Consider the full system lifecycle
- Agile Model Driven Development (AMDD)
- Continuous independent testing
- Risk and value-driven development
- Agile database techniques
- Lean development governance

Consider the Full System Lifecycle

Figure 1 depicts a system development lifecycle (SDLC) which shows the phases and major activities involved with the development and release into production of a system following an agile approach [9]. There are four phases to this SDLC, taking their name from the phases of the Unified Process [4, 8]:

- Inception. This is the initial phase of the project where you gain initial funding, perform initial requirements and architecture envisioning, obtain initial resources, and set up your environment. The goal is to define a firm foundation for your project team. This phase is also referred to as Iteration 0, Sprint 0, and Warm Up by various agile methods.
- Elaboration & Construction. During this phase you develop working software which meets the needs of your stakeholders. This phase is also referred to as Development, Construction, and Implementation by various agile methods.
- Transition. During this phase you do the work required to successfully deploy your system into production. This includes finalizing testing, finalizing documentation, baselining your project work products, training end users, training operations and support staff, and running pilot programs as necessary. This phase is also referred to as Release, Deployment, or End Game by various agile methods.
- Production. During this phase, typically the majority of the system lifecycle, you operate and support the system and your end users (hopefully) use it. This phase is also referred to as Maintenance and Support by some agile methods.

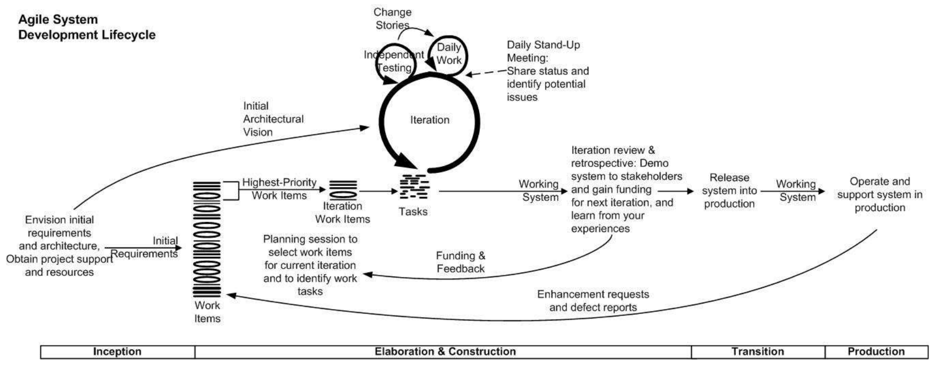
Agile System Development Lifecycle
Change Stories
Independent Testing
Daily Work
Daily Stand-Up Meeting: Share status and identify potential issues
Iteration
Initial Architectural Vision
Envision initial requirements and architecture, Obtain project support and resources
Initial Requirements
Work Items
Highest-Priority Work Items
Iteration Work Items
Tasks
Working System
Iteration review & retrospective: Demo system to stakeholders and gain funding for next iteration, and learn from your experiences
Release system into production
Working System
Operate and support system in production
Planning session to select work items for current iteration and to identify work tasks
Funding & Feedback
Enhancement requests and defect reports
Inception
Elaboration & Construction
Transition
Production

Figure 1. The Agile System Development Lifecycle (SDLC).

There are several reasons why it is important to adopt the lifecycle of Figure 1. First, too many agile teams focus on the construction aspects of the SDLC without taking into account the complexities of initiating a project, deploying into production, or even running the system once it is in production. The risks addressed by these phases are critical regardless of scale, but increase in importance in proportion to the rise in complexity resulting from the scaling factors mentioned earlier. Second, the lifecycle explicitly includes important scaling techniques such as initial requirements and architecture envisioning as well as continuous independent testing.

Agile Model Driven Development (AMDD)

As the name implies, Agile Model Driven Development (AMDD) is the agile version of Model Driven Development (MDD). MDD is an approach to software development where extensive models are created before source code is written. With traditional MDD a serial approach to development is often taken where comprehensive models are created early in the lifecycle. With AMDD you create agile models which are just barely good enough for the current situation at hand to that drive your overall development efforts. Figure 2 depicts the AMDD lifecycle for a project.

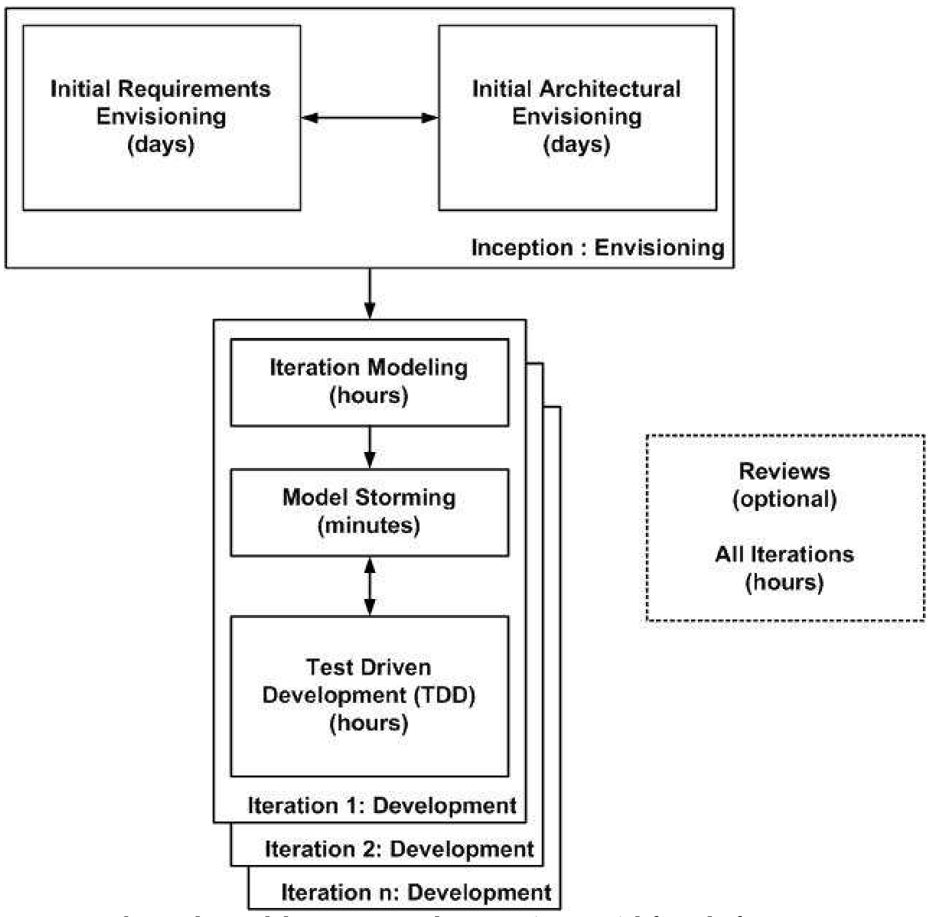

Figure 2. The Agile Model Driven Development (AMDD) lifecycle for a project.

As you can see in Figure 2 there are four critical modeling and specification activities:

1. Envisioning. The envisioning effort is typically performed during the first week of a project, the goal of which is to identify the scope of your system and a likely architecture for addressing it. To do this you will do both high-level requirements and high-level architecture modeling. The goal isn't to write detailed specifications but instead to explore the requirements and come to an overall strategy for your project. For short projects (perhaps several weeks in length) you may do this work in the first few hours and for long projects (perhaps on the order of twelve or more months) you may decide to invest two weeks in this effort due to the risks inherent in over modeling.
2. Iteration modeling. At the beginning of each Construction iteration the team must plan out the work that they will do that iteration, and an often neglected aspect of this effort is the required modeling activities implied by the technique. Agile teams implement requirements in priority order, as you can see with the work item stack of Figure 1, pulling an iteration's worth of work off the top of the stack. To

do this you must be able to accurately estimate the work required for each requirement, then based on your previous iteration's velocity (a measure of how much work you accomplished) you pick that much work off the stack. To estimate a work item effectively you will need to think through how you intend to implement it, and very often you'll model (often using inclusive tools such as whiteboards or paper) to do so.

3. Model storming. Model storming is just in time (JIT) modeling: you identify an issue which you need to resolve, you quickly grab a few team mates who can help you, the group explores the issue, and then everyone continues on as before. These "model storming sessions" are typically impromptu events, one project team member will ask another to model with them, typically lasting for five to ten minutes (it's rare to model storm for more than thirty minutes). The people get together, gather around a shared modeling tool (e.g. the whiteboard), explore the issue until they're satisfied that they understand it, then they continue on (often coding). Extreme programmers (XPers) would call modeling storming sessions stand-up design sessions or customer Q&A sessions.
4. Test-driven development (TDD). TDD is a technique where you write a single test and then just enough production code to fulfill that test [11]. Not only are you validating your software to the extent of your understanding of the stakeholder's intent up to that point, you are also specifying your software on a JIT basis. In short, with TDD agile teams capture detailed specifications in the form of executable tests instead of static documents or models.

Sinaalto and Abrahamsson [12] found that TDD may produce less complex code but that the overall package structure may be difficult to change and maintain. In other words, they found that although TDD is effective for "design in the small" that it is not effective for "design in the large." AMDD enables you to scale TDD through initial envisioning of the requirements and architecture as well as just-in-time (JIT) modeling at the beginning and during construction iterations.

AMDD also helps to scale agile software development when the team is large and/or distributed and when "the team" is the entire IT effort at the enterprise level. Figure 3 shows an agile approach to architecture at the program and enterprise levels [13, 14]. The architecture owners, the agile term for architects, develop the initial architecture vision through some initial modeling. They then become active participants on development teams, often taking on the role of architecture owner or technical team lead, and thereby help the team implement their part of the overall architecture. They take issues, and what they've learned from their experience on the project teams, back to the architecture team on a regular basis (at least weekly) to evolve the architecture artifacts appropriately.

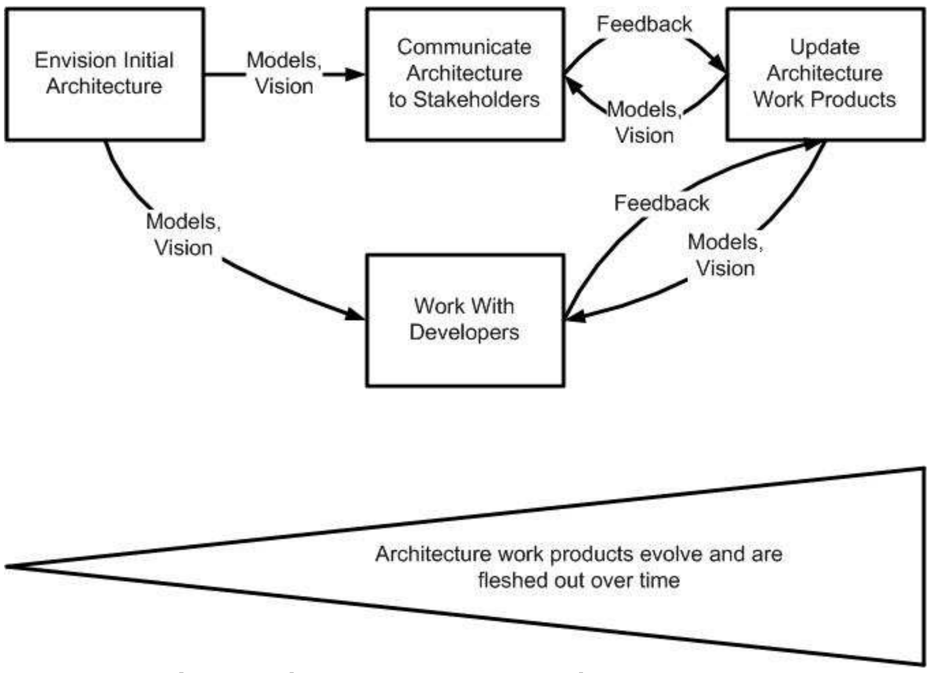
Envision Initial Architecture
Models, Vision
Communicate Architecture to Stakeholders
Feedback
Models, Vision
Update Architecture Work Products
Models, Vision
Feedback
Models, Vision
Work With Developers
Architecture work products evolve and are fleshed out over time

Figure 3. An agile approach to program/enterprise architecture.

Continuous Independent Testing

Although AMDD scales the specification aspects of TDD, it does nothing for the validation aspects. TDD is in effect an approach to confirmatory testing where you validate the system to the level of your understanding of the requirements. This is the equivalent of "smoke testing" or testing against the specification – while important, it isn't the whole validation picture. The fundamental challenge with confirmatory testing, and hence TDD, is that it assumes that stakeholders understand and can describe their requirements. Although iterative approaches increase the chance of this there are no guarantees. A second assumption of TDD is that developers have the skills to write and run the tests, skills that can be gained over time but which they may not have today.

The implication is that you need to add independent, investigative testing practices into your software process [15]. The goal of investigative testing is to explore issues that your stakeholders may not have thought of, such as usability issues, system integration issues, production performance issues, security issues, and a multitude of others. Agile teams, particularly those working at scale, often having a small independent test team working in parallel with them, as you can see depicted in Figure 1. The development team deploys the current working build into the testing sandbox, an environment which attempts to simulate the production environment. This deployment effort occurs at least once an iteration, although minimally I suggest doing so at least once a week if not nightly (assuming your daily build was successful).

The independent testers don't need a lot of details: The only documentation that they might need is a list of changes since the last deployment so that they know what to focus on first, because most likely new defects would have been introduced in the implementation of the changes. They will use complex, and often expensive, tools to do their jobs and will usually be very highly skilled people.

When the testers find a potential problem, it might be what they believe is missing functionality or it might be something that doesn't appear to work properly, which they write up as a "change story." Change stories are basically the agile form of a defect report or enhancement request. The development team treats change stories like requirements—they estimate the effort to address the requirement and ask their project stakeholder(s) to prioritize it accordingly. Then, when the change story makes it to the top of their prioritized work-item list, they address it at that point in time. Any potential defect found via independent investigative testing becomes a known issue that is then specified and validated via TDD. Because TDD is performed in an automated manner to support regression testing, the implication is that the investigative testers do not need to be as concerned about automating their own efforts, but instead can focus on the high-value activity of defect detection.

Risk and Value-Driven Development

The explicit phases of the Unified Process (UP) and their milestones are important strategies for scaling agile software development to meet the real-world needs of modern organizations. The UP lifecycle is risk and value driven [16]. What this means is that UP project teams actively strive to reduce both business and technical risk early in the lifecycle while delivering concrete feedback throughout the entire lifecycle in the form of working software. Where agile processes such as XP and Scrum are clearly value driven, they can be enhanced to address risk more effectively. This is particularly important at scale due to the increased risk associated with the greater complexity of such projects.

Each UP phase addresses a different kind of risk:

1. Inception. This phase focuses on addressing business risk by having you drive to scope concurrence amongst your stakeholders. Most projects have a wide range of stakeholders, and if they don't agree to the scope of the project and recognize that others have conflicting or higher priority needs you project risks getting mired in political infighting.
2. Elaboration. The goal of this phase is to address technical risk by proving the architecture through code. You do this by building and end-to-end skeleton of your system which implements the highest-risk requirements. These high-risk requirements are often the high-business-value ones anyway, so you usually need to do very little reorganization of your work items stack to achieve this goal.
3. Construction. This phase focuses on implementation risk, addressing it through the creation of working software each iteration. This phase is where you put the flesh onto the skeleton.
4. Transition. The goal of this phase is to address deployment risk. There is usually a lot more to deploying software than simply copying a few files onto a server, as I

indicated above. Deployment is often a complex and difficult task, one which you often need good guidance to succeed at.

5. Production. The goal of this phase is to address operational risk. Once a system is deployed your end-users will work with it, your operations staff will keep it up and running, and your support staff will help end users to be effective. You need effective processes in place to achieve these goals.

The first four phases end with a milestone review, which could be as simple as a short meeting, where you meet with prime stakeholders who will make a "go/no-go" decision regarding your system. They should consider whether the project still makes sense, perhaps the situation has changed, and that you're addressing the project risks appropriately. This is important for "agile in the small" but also for "agile in the large" because at scale your risks are often much greater. These milestone reviews enable you to lower project risk. Although agile teams appear to have a higher success rate than traditional teams, some agile projects are still considered failures [2]. The point is that it behooves us to actively monitor development projects to determine if they're on track, and if not either help them to get back on track or cancel them as soon as we possibly can.

Agile Database Techniques

Data is an important aspect of any business application, and to a greater extent of your organization's assets as a whole. Just as your application logic can be developed in an agile manner, so can your data-oriented assets [13]. To scale agile effectively, all members of the team must work in an agile manner, including data professionals. The following techniques enable data professionals to be active members of agile teams:

1. Database refactoring. A database refactoring is a simple change to a database schema that improves its design while retaining both its behavioral and informational semantics [17]. A database schema includes both structural aspects such as table and view definitions as well as functional aspects such as stored procedures and triggers. A database refactoring is conceptually more difficult than a code refactoring; code refactorings only need to maintain behavioral semantics while database refactorings also must maintain informational semantics. The process of database refactoring is the act of applying database refactorings in order to evolve an existing database schema, either to support evolutionary/agile development or to fix existing database schema problems.
2. Database testing. Databases often persist mission-critical data which is updated by many applications and potentially thousands if not millions of end users. Furthermore, they implement important functionality in the form of database methods (stored procedures, stored functions, and/or triggers) and database objects (e.g. Java or C# instances). The best way to ensure the continuing quality of these assets, at least from a technical point of view, is to have a full regression test suite which you can run on a regular basis.

3. Continuous database integration. Continuous integration is a development practice where developers integrate their work frequently, at least daily, where the integration is verified by an automated build. The build includes regression testing and possibly static analysis of the code. Continuous database integration is the act of performing continuous integration on your database assets. Database builds may include the creation of the database schema from scratch, something that you would only do for development and test databases, as well as database regression testing and potential static analysis of the database contents. Continuous integration reduces the average amount of time between injecting a defect and finding it, improving your opportunities to address database and data quality problems before they get out of control.
4. Agile data modeling. Agile data modeling is the act of exploring data-oriented structures in an iterative, incremental, and highly collaborative manner. Your data assets should be modeled, via an AMDD approach, along with all other aspects of what you are developing.

Lean Development Governance

Governance is critical to the success of any IT department, and it is particularly important at scale. Effective governance isn't about command and control, instead the focus is on enabling the right behaviors and practices through collaborative and supportive techniques. It is far more effective to motivate people to do the right thing than it is to try to force them to do so. Per Kroll and myself have identified a collection of practices that define a lean approach to governing software development projects [7]. These practices are:

1. Adapt the Process. Because teams vary in size, distribution, purpose, criticality, need for oversight, and member skillset you must tailor the process to meet a team's exact needs. Repeatable results, not repeatable processes, should be your true goal.
2. Align HR Policies With IT Values. Hiring, retaining, and promoting technical staff requires different strategies compared to non-technical staff.
3. Align Stakeholder Policies With IT Values. Your stakeholders may not understand the implications of the decisions that they make, for example that requiring an "accurate" estimate at the beginning of a project can dramatically increase project risk instead of decrease it as intended.
4. Align Team Structure With Architecture. The organization of your project team should reflect the desired architectural structure of the system you are building to streamline the activities of the team.
5. Business-Driven Project Pipeline. Invest in the projects that are well-aligned to the business direction, return definable value, and match well with the priorities of the enterprise.
6. Continuous Improvement. Strive to identify and act on lessons learned throughout the project, not just at the end.

   Embedded Compliance. It is better to build compliance into your day-to-day process, instead of having a separate compliance process that often results in unnecessary overhead.

7. Continuous Project Monitoring. Automated metrics gathering enables you to monitor projects and thereby identify potential issues so that you can collaborate closely with the project team to resolve problems early.
8. Flexible Architectures. Architectures that are service-oriented, component-based, or object-oriented and implement common architectural and design patterns lend themselves to greater levels of consistency, reuse, enhanceability, and adaptability.
9. Integrated Lifecycle Environment. Automate as much of the "drudge work", such as metrics gathering and system build, as possible. Your tools and processes should fit together effectively throughout the lifecycle.
10. Iterative Development. An iterative approach to software delivery allows progressive development and disclosure of software components, with a reduction of overall failure risk, and provides an ability to make fine-grained adjustment and correction with minimal lost time for rework.
11. Pragmatic Governance Body. Effective governance bodies focus on enabling development teams in a cost-effective and timely manner. They typically have a small core staff with a majority of members being representatives from the governed organizations.
12. Promote Self-Organizing Teams. The best people for planning work are the ones who are going to do it.
13. Risk-Based Milestones. You want to mitigate the risks of your project, in particular business and technical risks, early in the lifecycle. You do this by having throughout your project several milestones that teams work toward.
14. Scenario-Driven Development. By taking a scenario-driven approach, you can understand how people will actually use your system, thereby enabling you to build something that meets their actual needs. The whole cannot be defined without understanding the parts, and the parts cannot be defined in detail without understanding the whole.
15. Simple and Relevant Metrics. You should automate metrics collection as much as possible, minimize the number of metrics collected, and know why you're collecting them.
16. Staged Program Delivery. Programs, collections of related projects, should be rolled out in increments over time. Instead of holding back a release to wait for a subproject, each individual subprojects must sign up to predetermined release date. If the subproject misses it skips to the next release, minimizing the impact to the customers of the program.
17. Valued Corporate Assets. Guidance, such as programming guidelines or database design conventions, and reusable assets such as frameworks and components, will be adopted if they are perceived to add value to developers. You want to make it as easy as possible for developers to comply to, and more importantly take advantage of, your corporate IT infrastructure.

Conclusion

It is definitely possible to scale Agile software development to meet the real-world complexities faced by modern organizations. Based on my experiences, I believe that over the next few years we'll discover that agile approaches scale better than traditional approaches. Many people have already discovered this, and have adopted some or all of the strategies outlined in this paper, but as an industry I believe that there isn't yet sufficient evidence to state this as more than opinion. Time will tell.

References

1. Ambler, S.W. (2008). Dr. Dobb's Journal Agile Adoption Survey 2008. www.ambysoft.com/surveys/agileFebruary2008.html. Accessed on March 22 2008.
2. Ambler, S.W. (2007). Dr. Dobb's Journal Project Success Rates Survey 2007. www.ambysoft.com/surveys/success2007.html. Accessed on March 22 2008.
3. Eckstein, J. (2004). Agile Software Development in the Large: Diving into the Deep. New York: Dorset House Publishing.
4. Kruchten, P. (2004). The Rational Unified Process: An Introduction (3rd ed.). Reading, MA: Addison Wesley Longman.
5. Beck, K. (2000). Extreme Programming Explained—Embrace Change. Reading, MA: Addison Wesley Longman.
6. Schwaber, K. (2007). The Enterprise and Scrum. Redmond: Microsoft Press.
7. Kroll, P. and Ambler, S.W. (2007). Lean Development Governance. https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?lang=en_US&source=swg-ldg. Accessed on March 22 2008.
8. Ambler, S.W. , Nalbone, J. and Vizdos, M. J. (2005). The Enterprise Unified Process: Extending the Rational Unified Process. Upper Saddle River, NJ: Pearson Education.
9. Ambler, S.W. (2005). The Agile System Development Lifecycle (SDLC). http://www.ambysoft.com/essays/agileLifecycle.html . Accessed on March 22 2008.
10. Ambler, S.W. (2003) Agile Model Driven Development (AMDD). www.agilemodeling.com/essays/amdd.htm. Accessed on March 22 2008.
11. Astels D. (2003). Test Driven Development: A Practical Guide. Upper Saddle River, NJ: Prentice Hall.
12. Sinaalto, M. and Abrahamsson, P. (2007). Does Test Driven Development Improve the Program Code? Alarming Results from a Comparative Case Study. CEE-SET 2007 Conference Proceedings.
13. Ambler, S. W. (2003). Agile Database Techniques: Effective Strategies for the Agile Software Developer. New York: Wiley.
14. McGovern, J., Ambler, S.W., Stevens, M.E., Linn, J., Sharan, V, & Jo, E.K. (2004). The Practical Guide to Enterprise Architecture. Upper Saddle River, NJ: Prentice Hall PTR.
15. Ambler, S.W. (2007). Agile Testing Strategies. Dr. Dobb's Journal, January 2007. www.ddj.com/development-tools/196603549. Accessed on March 22 2008.
16. Kroll, P. and MacIsaac, B. (2006). Agility and Discipline Made Easy: Practices from OpenUP and RUP. Reading, MA: Addison Wesley Longman.
17. Ambler, S.W. and Sadalage, P.J. (2006). Refactoring Databases: Evolutionary Database Design. Boston: Addison Wesley.

Bio

Scott W. Ambler is Practice Leader Agile Development with IBM Rational. Scott has a Master of Information Science from the University of Toronto and is author of several books including Agile Modeling, Agile Database Techniques, and Refactoring Databases. Scott helps organizations around the world to improve their software processes. He is a Senior Contributing Editor with Dr. Dobb's Journal (www.ddj.com) and writes about strategies for scaling software development at www.ibm.com/developerworks/blogs/page/ambler .