Quality is an inherent aspect of true agile software development. The majority of agilists take a test-driven approach to development where they write a unit test before they write the domain code to fulfill that unit test, with the end result being that they have a regression unit test suite at all times. They also consider acceptance tests as first-class requirements artifact, not only promoting regular stakeholder validation of their work but also their active inclusion in the modeling effort itself. Agilists refactor their source code and database schema to keep their work at the highest possible quality at all times. The challenge for quality professionals is that agilists work in a highly collaborative and evolutionary (iterative and incremental) manner, often requiring traditional quality professionals to change their approach.

Key words: agile model driven development, agile software development, collaborative development, evolutionary, incremental development, iterative development, refactoring, software processes, test-driven development

SQP References

The Role of Testers in the Agile Methods

vol. 7, issue 3
Alan S. Koch

Extreme Programming Series

vol. 5, issue 1
Pieter Botman and Ray Schneider

Quality in an Agile World

BY SCOTT AMBLER
Ambysoft, Inc.

INTRODUCTION

Modern software processes are evolutionary, iterative, and incremental in nature. This includes processes such as Rational Unified Process (RUP) (Kruchten 2004), Extreme Programming (XP) (Beck 2000), Enterprise Unified Process (EUP) (Ambler, Nalbone, and Vizdos 2005), and rapid application development (RAD), to name a few. Furthermore, many modern processes are agile, which for the sake of simplicity I will characterize as both evolutionary and highly collaborative in nature. The fundamental nature of software development is changing, and quality professionals must change with it. In this article I introduce readers to common agile software development techniques and argue that they lead to software that proves in practice to be of much higher quality than what traditional software teams usually deliver. It is common to say that agilists are "quality infected," and that the greater emphasis on quality implies a changed and perhaps even smaller role for quality professionals on agile software development projects.

Why are developers adopting evolutionary, and more often, agile software processes? Over the years developers have discovered that traditional software development processes, in particular the "well-defined" prescriptive processes, work poorly in practice (Larman 2004). First, the Chaos report, published by the Standish Group (http://www.standishgroup.com), still shows a significant failure rate within the industry, indicating that prescriptive processes simply aren't fulfilling their promise. Second, it appears that most developers do not want to adopt prescriptive processes and will find ways to undermine efforts to adopt them, either consciously or subconsciously. Third, the "big design up front" (BDUF) approaches to software development, where significant modeling and documentation occurs before a single line of code is written, are incredibly risky in practice because they don't easily support change or feedback. For example, one might build what was defined in the specifications, but if the specifications don't reflect what people actually require then there is serious trouble. This risk is often ignored, if it is recognized at all, by the people promoting these approaches. Fourth, most prescriptive processes promote activities that are only slightly related to the actual development of

software. In short, the bureaucrats have taken over in many traditional information technology (IT) organizations. Something has to give.

THE AGILE MOVEMENT

To address the challenges inherent to traditional development a group of 17 methodologists formed the Agile Software Development Alliance (http://www.agilealliance.org), often referred to simply as the Agile Alliance, in February 2001. Although all members of this group came from different backgrounds they were able to come to an agreement on issues that methodologists typically do not agree upon. They concluded that to succeed at software development one must focus on people-oriented issues and follow development techniques that readily support change.

The "agile movement" started when they published their manifesto of four values and 12 principles (see sidebar, "The Values and Principles of the Agile Alliance"). A software process is considered agile when it conforms to these values and principles. Agile software processes include XP, Scrum, dynamic system development method (DSDM), the Crystal family, feature-driven development (FDD), agile modeling (AM), and the agile data (AD) method. Although XP is clearly the most popular agile process, there are a variety to choose from.

Just as there are common development techniques and concepts, such as code inspections and data modeling, within the traditional world, there are also common development techniques within the agile world. Interestingly, many of these techniques are focused on the creation and delivery of high-quality software. These agile techniques and concepts include:

- Refactoring
- Test-driven development (TDD)
- Tests replace traditional artifacts
- Agile model driven development (AMDD)

Refactoring

Refactoring (Fowler 1999) is a disciplined way to make small changes to source code to improve its design, making it easier to work with. A critical aspect of a refactoring is that it retains the behavioral semantics of the code—developers neither add nor remove anything when they refactor, they merely improve its quality. An example refactoring would be to rename the getPersons() operation to getPeople(). To implement this refactoring they must change the operation definition and then change every invocation of this operation throughout the application code. A refactoring isn't complete until the code runs again as before.

Similarly, a database refactoring (Ambler 2003a) is a simple change to a database schema that improves its design while retaining both its behavioral and informational semantics. Developers could refactor either structural aspects of the database schema, such as table and view definitions, or func-

Some Definitions

This article uses the terms collaborative, incremental, and iterative development, and it is important to understand how each term is used:

1. **Collaborative development.** When a team takes a collaborative approach they actively strive to find ways to work together effectively; teams should even try to ensure that project stakeholders such as business customers are active team members. They should strive to adopt the "hottest" communication technique applicable to their situation: prefer face-to-face conversation around a whiteboard to a telephone call, prefer a telephone call to sending an e-mail, and prefer an e-mail to sending someone a detailed document. The better the communication and collaboration within a software development team, the greater its chance of success (Cockburn 2002).
2. **Incremental development.** With an incremental approach a team organizes its system into a series of releases instead of one big one.
3. **Iterative development.** Working iteratively, a team does a little bit of an activity such as modeling, testing, coding, or deployment at a time, and then does another little bit, then another, and so on. This is different than a serial approach where they identify all of the requirements they are going to implement, create a detailed design, implement to that design, test and, finally, deploy the system.

tional aspects, such as stored procedures and triggers. When they refactor their database schema, not only must they rework the schema itself but also the external systems, such as business applications or data extracts, which are coupled to their schema. Database refactorings are clearly more difficult to implement than code refactorings; therefore, developers need to be careful.

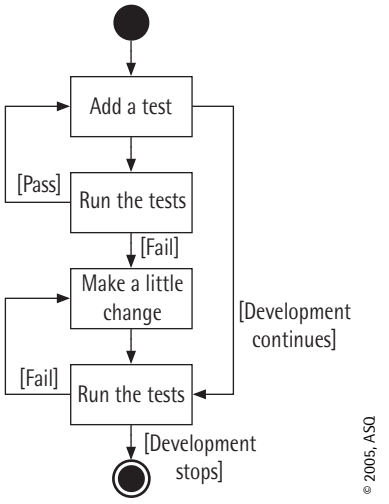
Many agile developers, and in particular extreme programmers (XPers), consider refactoring to be a primary development practice. It is just as common to refactor a bit of code as it is to introduce an "if" statement or a loop. Agile developers will refactor their code mercilessly because they understand that they are most productive when they are working on high-quality source code; therefore, they will refactor their code whenever needed. When agilists have a new feature to add to their code, the first question they ask is, "Is this code the best design possible that enables me to add this feature?" If the answer is "yes," then they add the feature. If the answer is "no," they first refactor their code to make it the best design possible and then add the feature. On the surface this sounds like a lot of work, but in practice if they start with high-quality source code, and then refactor it to keep it so, this approach works incredibly well.

Test Driven Development (TDD)

TDD (Beck 2003; Astels 2003), also known as test-first programming or test-first development, is an evolutionary (iterative and incremental) approach to programming where agile software developers must first write a test that fails before they write new functional code. The steps of TDD, as shown in Figure 1, are:

1. Quickly add a test, basically just enough code so that the tests now fail.
2. Run the tests, often the complete test suite, although for sake of speed they may run only a subset to ensure that the new test does in fact fail.
3. Update the functional code so it passes the new test.
4. Run the tests again.
5. If the tests fail return to step 3.
6. Once the tests pass the next step is to start over (agilists may also want to refactor any duplication out of their design as needed).

FIGURE 1 Test-driven development



There are several advantages of TDD for agile software development. First, TDD forces developers to think about what new functional code should do before they write it—in other words, to do detailed design just in time (JIT) before writing the code. Second, it ensures that agile developers have testing code available to validate their work, ensuring that they test as often and early as possible. Third, it gives them the courage to refactor their code to keep it the highest quality possible, because they know there is a test suite in place that will detect if they have "broken" anything as the result of refactoring.

Tests Replace Traditional Artifacts

When agile developers become "test infected" they start to realize that tests are truly first-class artifacts that need to be developed and maintained throughout a project. They also realize that tests, when created properly, are more than just tests. For example, agilists consider acceptance tests to be first-class requirement artifacts—if an acceptance test defines a criteria that the system must exhibit, then clearly it is a requirement. Another common philosophy is that unit tests are detailed design artifacts. With a TDD-based approach agilists write their unit test before writing their source code; in other words, they think through what the source code must do before they write it. The implication is that a unit test, when written before the domain code, effectively becomes a design specification for that portion of code.

These philosophies can dramatically reduce the amount of work agilists need to do. The AM methodology version 2 (http://www.agilemodeling.com) includes a practice called Single Source Information, which advises them to record information once, ideally in the best place possible. Often, the best place to record technical information is in their tests and source code, not in voluminous documentation that rarely gets updated. Furthermore, because the agile developers are creating far less documentation, their traceability needs are thereby reduced, once again reducing their overall work effort.

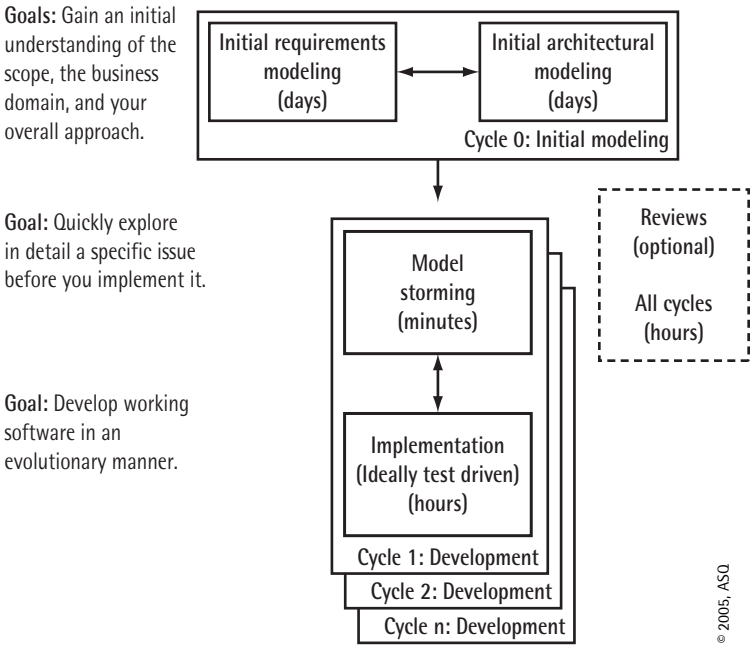
Agile Model Driven Development (AMDD)

Some people may have heard that agile software developers don't model, but frankly nothing can be further from the truth. They still model, they just don't write as much documentation because they know it is not a good investment of their time. Because they work closely with their project stakeholders, they discover that they don't need a lot of documentation telling them what they are doing; instead, they simply look for themselves. Because they don't need to write as much documentation they can instead focus their efforts on building high-quality working software.

The agile approach to modeling is described in the AMDD method (Ambler 2003b; Ambler 2004). AMDD describes how developers and stakeholders can work together to create models that are just barely good enough. It assumes that each individual has some modeling skills, or at least some domain knowledge that he or she will apply together in a team in order to get the job done. It is reasonable to assume that developers will understand a handful of modeling techniques, but not all of those available to them (see http://www.agilemodeling.com/artifacts for a list). It is also reasonable to assume that people are willing to learn new techniques over time, often by working with someone who already has those skills. AMDD does not require everyone to be a modeling expert, it just requires him or her to be willing to try. AMDD also allows people to use the most appropriate modeling tool for the job, which is often a simple tool such as a whiteboards or paper, because it is important to find ways to communicate effectively, not document comprehensively. There is nothing wrong with sophisticated CASE tools in the hands of people who know how to use them, but AMDD does not depend on such tools.

The life cycle for AMDD is shown in Figure 2. Each box represents a development activity. The initial upfront modeling activity occurs during the first week or so of a project. The goal of this activity is to identify the initial requirements and a potential architecture for the system. The end result might be a stack of index cards or a spreadsheet of point-form text representing the initial requirements and some sketches representing the architecture. It is not necessary to have comprehensive documentation to start development; developers just need a good idea as to the scope of the system and how they think they are going to implement it. The details come later in model storming sessions, where they model JIT and just enough to address the issue at hand. Perhaps developers will get together with a stakeholder to analyze the requirement they are currently working on, create a sketch together

FIGURE 2 Agile model driven development life cycle



at a whiteboard for a few minutes, and then go back to coding. Or perhaps they will sketch out an approach to implement a requirement, once again spending several minutes doing so. Or perhaps they will use a modeling tool to model in detail and then generate the code for that requirement. Model storming sessions should not take more than 15 to 20 minutes.

An interesting aspect of Figure 2 is that it indicates that model reviews are optional (Ambler 2003c). Is this heresy? No, it is just common sense. With an

The Values and Principles of the Agile Alliance

The Agile Alliance developed a manifesto defining four values for encouraging better ways of developing software. These values indicate preferences, not alternatives, encouraging a focus on certain areas but not eliminating others. While developers should value the concepts on the right-hand side (for example, processes and tools), they should value the things on the left-hand side (for example, individuals and iteractions) even more. The four values are:

1. **Individuals and interactions** *over* **processes and tools.** The most important factors to consider are the people and how they work together, because if developers don't get that right the best tools and processes won't be of any use.
2. **Working software** *over* **comprehensive documentation.** The primary goal of software development is to create software, not documents; otherwise, it would be called documentation development. Documentation has its place, and written properly it is a valuable guide for understanding how and why a system is built and how to work with the system.
3. **Customer collaboration** *over* **contract negotiation.** Only one's stakeholders can state what they want. They likely do not have the skills to exactly specify the system, they likely won't get it right at first, and they will likely change their minds. Working together with customers is hard, but that's the reality of the job. Having a contract with customers is important, but a contract is not a substitute for communication.
4. **Responding to change** *over* **following a plan.** Change is a reality of software development, a reality that the software process must reflect. People change their priorities for a variety of reasons, their understanding of the problem domain changes as they see the developer's work, the business environment changes, as does technology. Although one needs a project plan, it must be malleable and it can be very simple (unlike many Gantt charts seen in the past).

Because the agile values are not sufficient to form a foundational philosophy for software development, the Agile Alliance also identified 12 principles to which an agile software process must conform:

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 time scale.
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 and then tunes and adjusts its behavior accordingly.

AMDD approach developers follow practices such as model with others, collective ownership, and apply modeling standards, which say one should never model alone, that everyone should have access to every project artifact, and that one should follow a common set of modeling and documentation conventions, respectively. These two practices, combined with the highly collaborative environment common to agile projects and the desire to work with the right people at the right time effectively, result in "mini-reviews" occurring whenever one is working. With AMDD developers are never in a position where one person works on an artifact, nor where a subset of people work on "their" artifacts. Better yet, everyone works to the same set of conventions (with collective ownership it can't be any other way). In practice, because developers are doing in-progress reviews as they are working they quickly find that reviewing things after the fact offers little value.

It is important to point out that AMDD works on both large and small projects. If needed, one can take an agile approach to architecture (Ambler 2002) to ensure that all of the subteams involved in a larger effort work to the same vision.

IMPLICATIONS FOR QUALITY PROFESSIONALS

There are several interesting implications of agile software development for quality professionals:

1. **Greater quality implies less need for quality assurance activities.** Higher-quality code implies less need for quality assurance activities, such as reviews and inspections, elsewhere in the life cycle. Agilists are quality infected; they have accepted what the quality assurance community has been saying for decades, and they are actively developing high-quality system artifacts as a result. The implication is that they will find that they can start diverting quality assurance resources away from agile projects and focus more on traditional projects where the risk of low-quality work is much higher.
2. **Get used to "incomplete" artifacts.** Models, documents, and source code evolve over time. The only time when agilists can honestly say that something is done is when they are ready to release it into production. For example, they won't have a finalized requirements document from which to develop test cases until the very end of the project. The implication is that if they are going to be involved with agile software projects then they need to get used to working with in-progress artifacts.
3. **Become a generalizing specialist.** A critical concept is that agilists need to move away from being narrowly focused specialists to become more of what I like to call generalizing specialists (Ambler 2003d). A generalizing specialist is someone with one or more technical specialties who actively seeks to gain new skills in his or her existing specialties as well as in other areas. A person needs one or more specialties so he or she can add value to the team, but one needs a general understanding of the software process and of the business domain so he or she can interact effectively with others on the team. With evolutionary approaches to development there is no longer room to invest several hours, let alone days or weeks, on a given activity. One does a little bit of requirements elicitation, then some analysis and design, then some implementation. Specialists who want to focus on one narrow task, such as facilitating a review, will not be very effective in this environment.

CONCLUSION

These ideas are not new, but their popularity is. Agile software development is real, it works, and it's here to stay. Agilists are arguably taking to heart what quality professionals have been saying for decades, and because of that the role of quality professionals now needs to evolve to reflect this new reality. The IT industry is undergoing yet another paradigm shift. Are you willing to shift with it?

REFERENCES

Ambler, S. W. 2002. Agile modeling: Effective practices for extreme programming and the unified process. New York: John Wiley and Sons.

Ambler, S. W. 2003a. Agile database techniques: Effective strategies for the agile software developer. New York: John Wiley and Sons.

Ambler, S. W. 2003b. Agile model driven development. See URL: http://www.agilemodeling.com/essays/amdd.htm.

Ambler, S. W. 2003c. Model reviews: Best practice or process smell? See URL: http://www.agilemodeling.com/essays/modelReviews.htm.

Ambler, S. W. 2003d. Generalizing specialists: Improving your IT career skills. See URL: http://www.agilemodeling.com/essays/generalizingSpecialists.htm.

Ambler, S. W. 2004. The object primer, 3rd edition: Agile model driven development with UML 2. New York: Cambridge University Press.

Ambler, S. W., J. Nalbone, and M. J. Vizdos. 2005. The enterprise unified process: Extending the rational unified process. Upper Saddle River, N.J.: Prentice Hall PTR.

Astels, D. 2003. Test driven development: A practical guide. Upper Saddle River, N.J.: Prentice Hall.

Beck, K. 2000. Extreme programming explained—Embrace change. Reading, Mass.: Addison-Wesley Longman, Inc.

Beck, K. 2003. Test-driven development: By example. Boston: Addison-Wesley.

Cockburn, A. 2002. Agile software development. Reading, Mass: Addison-Wesley Longman, Inc.

Fowler, M. 1999. Refactoring: Improving the design of existing code. Menlo Park, Calif.: Addison-Wesley Longman, Inc.

Kruchten, P. 2004. The rational unified process, 3rd edition. Reading, Mass.: Addison-Wesley Longman, Inc.

Larman, C. 2004. Agile and iterative development: A manager's guide. Boston: Addison-Wesley.

Palmer, S. R., and J. M. Felsing. 2002. A practical guide to feature-driven development. Upper Saddle River, N.J.: Prentice Hall PTR.

BIOGRAPHY

Scott W. Ambler is a senior consultant with Ontario-based Ambysoft Inc. (http://www.ambysoft.com), a software services consulting firm that specializes in software process mentoring and improvement. He is founder and thought leader of the Agile Modeling (http://www.agilemodeling.com), Agile Data (http://www.agiledata.org), and Enterprise Unified Process (http://www.enterpriseunifiedprocess.com) methodologies. He helps organizations adopt and tailor these processes to meet their exact needs as well as provides training and mentoring in these techniques.

Ambler is the coauthor of several books, including *Agile Modeling* (John Wiley & Sons), *The Elements of UML 2.0 Style* (Cambridge University Press), and the forthcoming *Database Refactoring* (Prentice Hall PTR). Ambler is a contributing editor with *Software Development* magazine (http://www.sdmagazine.com) and his personal page is http://www.ambysoft.com/scottAmbler.html. He can be reached by e-mail at swa@ambysoft.com.