# 1. Agile Software Development

This section is a brief introduction to agile software development. It is organized into the following sections:

## 1.1 Defining Agile

One frustration many people new to agile have is that there is no official definition of agile software development, although many people will point to the values and principles of the Agile Manifesto. Having said that, my definition of disciplined agile software development is:

> **An iterative and incremental (evolutionary) approach to software development performed in a highly collaborative manner by self-organizing teams within an effective governance framework, with "just enough" ceremony that produces high quality software in a cost effective and timely manner to meets the changing needs of its stakeholders.**

The criteria to look for in determining whether a team is taking a disciplined approach to agile development comprises the following:

First, is the team is doing developer regression testing, or better yet taking a test-driven approach to development? Second, are stakeholders active participants in development? Third, is the team producing high-quality, working software on a regular basis? Fourth, is the team is working in a highly collaborative, self-organizing manner within an effective governance framework? Fifth, are they improving their approach throughout the project?

## 1.2 The Agile System Development Lifecycle

To understand agile testing and quality strategies you must understand how they fit into the overall agile system/software development lifecycle (SDLC). Figure 1 depicts a high-level view of the agile development lifecycle, showing that agile projects are organized into a series of time-boxes called iterations (in the Scrum methodology they call them "Sprints" and some people refer to them as cycles). Although many people with tell you that the agile lifecycle is iterative this isn't completely true, as you can see it is really serial in the large and iterative in the small. The serial aspect comes from the fact that there are at least three different iteration flavors -- initiation iterations (light green), construction iterations (light blue), and release iterations (blue) -- wherein the nature of the work that you do varies. The implication is that your approach to testing/validation also varies depending on where you are in the lifecycle. As a result, it is important to understand each of the high-level activities depicted by this lifecycle:

- **Iteration 0**. The goal of this iteration, or iterations on complex projects, is to initiate the project. You will perform initial requirements envisioning, initial architecture envisioning, begin identifying and organizing the development team, come to some sort of stakeholder concurrence as to the goal(s) of the project, and gain funding and support for the project. Testing/validation activities include beginning to set up your testing environment and tools as well as potentially reviewing the initial models, plans, and vision or stakeholders goals document.
- **Construction iterations**. During each construction iteration (iterations 1 to N in Figure 1) the goal is to produce more potentially shippable software. An Agile team follows the prioritized requirements practice; in each iteration, it takes the most important requirements remaining from the work item stack (what Scrum teams call a product backlog) and implements them. Agile teams will take a whole team approach, wherein testers are embedded in the development team, working side by side with them to build the system. The focus of their testing efforts is often on confirmation testing via developer regression testing or better yet Test-Driven Development (TDD).
- **Parallel independent testing**. Disciplined agile teams perform continuous independent testing in parallel to construction iterations throughout the lifecycle. The goal of this effort is to find defects that got past the development team, often performing higher forms of testing such as exploratory testing, system integration testing, security testing, usability testing, and so on which require significant testing skills, complex testing tools, and often complex pre-production testing environments. There is often a 10:1 to 15:1 ratio between people on the development team and independent testers supporting them. In larger organizations, this independent test team typically supports several development teams (thus enabling more sophisticated system integration testing because they can more easily work with versions of multiple systems under development).
- **Release iteration(s)**. The goal of the release iteration(s), the dark blue "R" iteration in Figure 1, is to deploy your system into production successfully. This can be quite complex in practice, including training of end users, support people, and operations people; communication/marketing of the product release; backup and potential restoration (if things go bad); pilot/staged deployment of the system; final translation of the UI and documentation; finalization of system and user documentation; and so on. During the release, iteration there is still some testing at the end of the lifecycle to ensure that the system is ready for production.

**Figure 1. High-level agile development lifecycle**

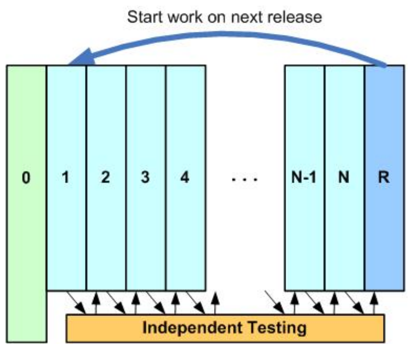
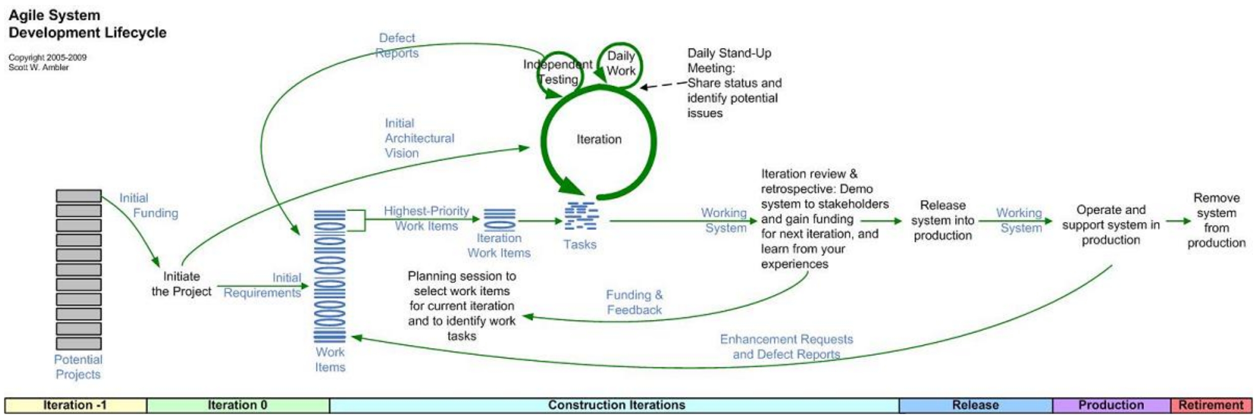


Figure 2 shows a detailed version of the SDLC, fleshing out the details of Figure 1. Figure 2 also adds new phases so as to depict the full end-to-end lifecycle, including "iteration -1" where you identify potential projects, the production phase where you operate and support the system once it has been released, and the retirement phase where you fully remove an unneeded system from production.

**Figure 2. A detailed agile SDLC.**


Agile System Development Lifecycle
Copyright 2005-2009 Scott W. Ambler
Defect Reports
Independent Testing
Daily Work
Daily Stand-Up Meeting: Share status and identify potential issues
Initial Architectural Vision
Iteration
Initial Funding
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
Remove system from production
Initiate the Project
Initial Requirements
Planning session to select work items for current iteration and to identify work tasks
Funding & Feedback
Enhancement Requests and Defect Reports
Potential Projects
Work Items
Iteration -1
Iteration 0
Construction Iterations
Release
Production
Retirement

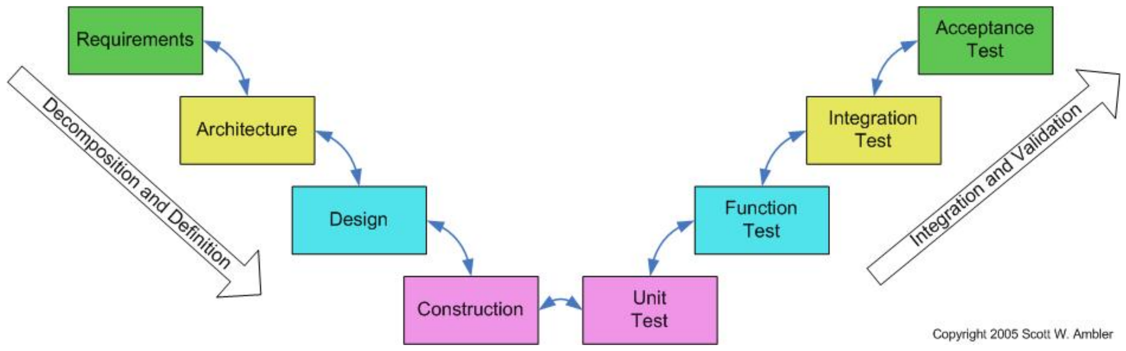
## 1.3 The Traditional System Development Lifecycle

Figure 3 depicts the V Model for software development, basically a sophisticated form of the traditional waterfall model. With the V model the work on the left-hand side of the diagram is validated later in the lifecycle through corresponding activities later in the lifecycle (for example requirements are validated through acceptance testing, the architecture via integration testing, and so on). Although this approach is better than not testing at all, it proves to be very expensive in practice because of several systemic problems:

- **Deliver the wrong functionality**. The V model promotes an approach where the requirements are defined in detail early in the project. Although this may be good theory (note the use of the word "may") in practice it proves to be a very poor way to work. These "big requirements up front" BRUF approaches result in significant wastage because at best the project team will build something to specification instead of something that the stakeholders actually need.
- **Build to a fragile design**. Similarly, although in theory it may be a good idea to think through the details of the architecture/design up front the reality is that you end up making, and then committing to, technical decisions are too early when you have the least amount of information available to you (during architecture/design phases you're making decisions based on what you hope will work, as opposed to what you know will work based on actual working software).
- **Hand-offs inject defects**. Every time you hand off between two groups of people, defects due to misunderstandings will be injected into the product. Although this problem can be partly mitigated via reviews, this proves to be expensive compared to more agile approaches such as non-solo development.

- **Fixing defects is expensive**. The greater the feedback cycle the greater the average cost of fixing a found defect. The V model approach promotes very long feedback cycles.
- **Increased time to value**. The V model lengthens the amount of time, through increased bureaucracy and waiting time, it takes to deliver functionality into production. This in turn lowers the opportunity benefits and net present value (NPV) provided by the release.

**Figure 3. Serial SDLC/V Model.**



## 1.4 How Agile is Different

Traditional testing professionals who are making the move to agile development may find the following aspects of agile development to be very different from what they are used to:

- **Greater collaboration**. Agile developers work closely together, favoring direct communication over passing documentations back and forth to each other. They recognize that, in rapidly changing/dynamic environments, documentation is the least effective manner of communication between people, even when economical due to disparate locations, although even this is mitigated in the era of real-time video conferencing.
- **Shorter work cycle**. The time between specifying a requirement in detail and validating that requirement is now on the order of minutes, not months or years, due to the adoption of test-driven development (TDD) approaches, greater collaboration, and less of a reliance on temporary documentation.
- **Agilists embrace change**. Agile developers choose to treat requirements like a prioritized stack that is allowed to change throughout the lifecycle. A changed requirement will be a competitive advantage if you're able to implement it.
- **Greater flexibility is required of testers**. Gone are the days of the development team handing off a "complete specification" against which the testers can test. The requirements evolve throughout the project. Ideally, acceptance-level "story tests" are written before the production code. This fulfills them, implying that the tests become detailed requirement specifications.
- **Greater discipline is required of IT**. It's very easy to say that you're going to work closely with your stakeholders, respect their decisions, produce potentially shippable software on a regular basis, and then write a single test before writing just enough

production code to fulfill that test (and so on) but is actually a lot harder to do them. Agile development requires far greater discipline than traditional development.

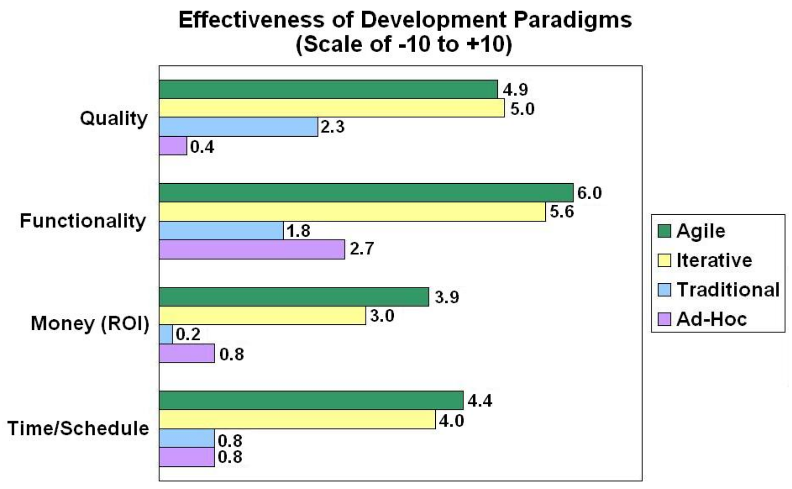
- **Greater accountability is required of stakeholders**. One of the implications of adopting the practices active stakeholder participation, prioritized requirements, and producing working software on a regular basis is that stakeholders are now accountable for the decisions that they make.
- **Greater range of skills is required**. It isn't enough to be just a tester, or just a programmer, or just an analyst, or just an anything anymore. Agilists are moving away from the Tayloristic approaches of traditional development that motivate people to become overly specialized, and instead are moving towards a highly iterative and collaborative approach which requires generalizing specialists (as opposed to generalists).

## 1.5 Comparing Agile and Traditional Approaches

The agile approach offers many benefits over the traditional V model:

- **Greater ability to deliver required functionality**. Agile teams work closely with their stakeholders, ideally following the practice active stakeholder participation. This practice, in combination with an evolutionary approach, results in greater ability of agile teams to understand and then implement the actual needs of their stakeholders. Figure 4 summarizes results from Dr. Dobb's Journal (DDJ)'s 2008 Project Success Survey, showing that agile teams are more effective at delivering required functionality than traditional teams.
- **Greater quality**. Figure 4 also shows that agile approaches result in greater quality than traditional approaches, most likely due to increased collaboration within the team and earlier and very often more testing throughout the lifecycle.
- **Improved designs**. Agile architecture and agile design strategies are evolutionary in nature, and this, in combination with the greater levels of collaboration exhibited by agile teams, results in better designs when compared to approaches that are more traditional. Architecture and design are so important to agile teams that they do these activities throughout the lifecycle, not just during early lifecycle phases.
- **Improved economics**. The statistics in Figure 4 reveal that agile teams are providing greater return on investment (ROI) than traditional teams. This is due in part to the shorter feedback cycle of agile approaches which lowers the average cost of addressing defects. Furthermore, because agile teams are working smarter, not harder, they often get the functionality delivered quicker, thereby providing accelerated time to value and thus greater benefit.

**Figure 4. Success factors by paradigm (Scale is from -10 to +10).**



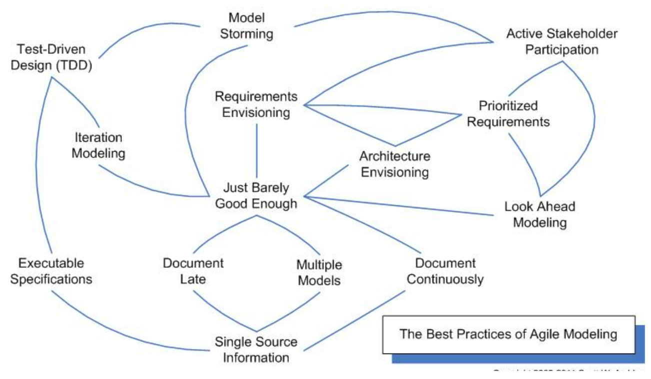
Finally, I just wanted to point out that the results depicted in Figure 4 aren't an anomaly. DDJ's 2008 Agile Adoption Survey also found that people believed that agile teams were producing greater quality than traditional teams, providing better stakeholder satisfaction, and providing greater levels of productivity.

## 2. Agile Requirements Strategies

This section provides an overview of agile approaches to requirement elicitation and management. This is important because your approach to requirements goes hand-in-hand with your approach to validating those requirements; therefore, to understand how disciplined agile teams approach testing and quality you first need to understand how agile teams approach requirements. Figure 5 depicts a process map of the best practices of Agile Modeling (AM) which address agile strategies for modeling and documentation, and in the case of TDD and executable specifications arguably strays into testing. This section is organized into the following topics:

- Active Stakeholder Participation
- Functional Requirements Management
- Initial Requirements Envisioning
- Iteration Modeling
- Just in Time (JIT) Model Storming
- Non-Functional Requirements Management>
- Who is Doing This?
- The Implications for Testing

**Figure 5. The best practices of Agile Modeling.**



## 2.1 Active Stakeholder Participation

Agile Modeling's practice of Active Stakeholder Participation says that 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. When stakeholders work closely with development, it increases the chance of project success by increasing the:

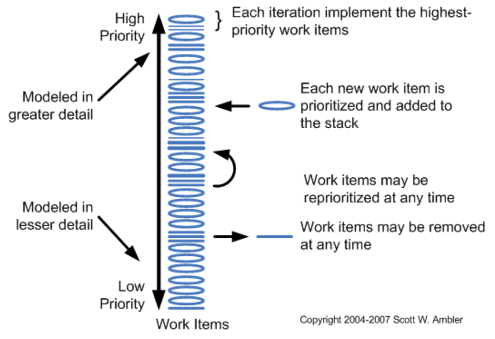
- Chance that the developers will understand the actual needs of the stakeholders
- Stakeholder's ability to steer the project by evolving their requirements based on seeing working software being developed by the team
- Quality of what is being built by being actively involved with acceptance testing throughout the lifecycle

The traditional approach of having stakeholders participate in a requirements elicitation phase early in the project and then go away until the end of the project for an acceptance testing effort at the end of the lifecycle proves to be very risky in practice. People are not very good at defining their requirements up front and as a result, with a serial approach to development a significant effort is invested in building and testing software, which is never even used once the system is in production. To avoid these problems agilists prefer an evolutionary approach where stakeholders are actively involved, an approach that proves more effective at delivering software that people actually want.

### 2.2 Functional Requirements Management

A fundamental agile practice is Prioritized Requirements Stack, called Product Backlog in Scrum. The basic ideas, shown in Figure 6, are that you should implement requirements in prioritized order and let your stakeholders evolve their requirements throughout the project as they learn. The diagram also indicates several advanced agile concepts. First, it's really a stack of work items and not just functional requirements (defect reports also appear on the stack as you can see in Figure 2, more on this later, and you also need to plan for work such as reviewing artifacts from other teams and taking vacations). Second, to reduce the risks associated with complex work items, not all work items are created equal after all, you will want to consider modeling a bit ahead whenever a complex work item is an iteration or two away.

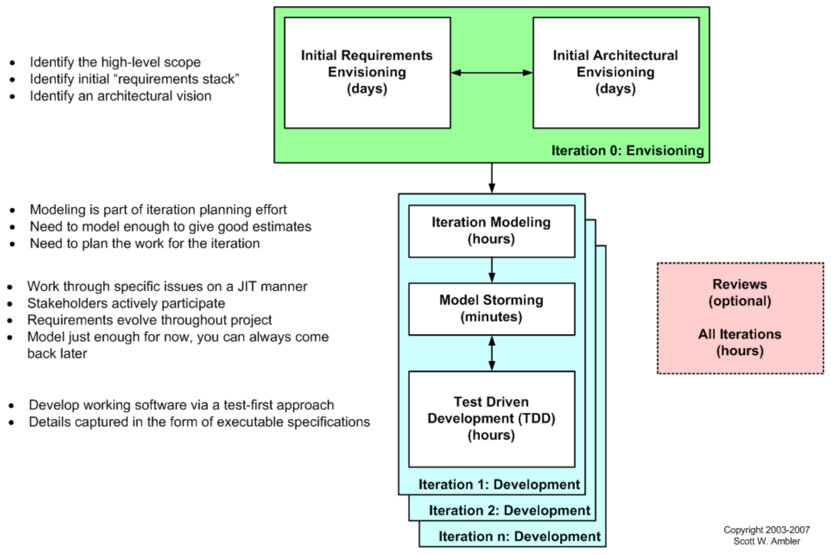
**Figure 6. Agile requirements change management process.**


High Priority
Each iteration implement the highest-priority work items
Modeled in greater detail
Each new work item is prioritized and added to the stack
Work items may be reprioritized at any time
Modeled in lesser detail
Work items may be removed at any time
Low Priority
Work Items
Copyright 2004-2007 Scott W. Ambler

## 2.3 Envisioning Initial Requirements

Figure 7 depicts the project lifecycle of Agile Model Driven Development (AMDD). As you see in Figure 7, during Iteration 0 agilists will do some initial requirements modeling with their stakeholders to identify the initial, albeit high-level, requirements for the system. The goal of initial requirements envisioning is to do just enough modeling to identify the scope of the system and to produce the initial stack of requirements that form the basis of your prioritized work item list (it just doesn't magically appear one day, after all). The goal is not to create a detailed requirements specification as that strategy actually increases your project risk in practice.

**Figure 7: The Agile Model Driven Development (AMDD) Lifecycle.**


Identify the high-level scope
Identify initial "requirements stack"
Identify an architectural vision
Initial Requirements Envisioning (days)
Initial Architectural Envisioning (days)
Iteration 0: Envisioning
Modeling is part of iteration planning effort
Need to model enough to give good estimates
Need to plan the work for the iteration
Iteration Modeling (hours)
Work through specific issues on a JIT manner
Stakeholders actively participate
Requirements evolve throughout project
Model just enough for now, you can always come back later
Model Storming (minutes)
Reviews (optional)
All Iterations (hours)
Develop working software via a test-first approach
Details captured in the form of executable specifications
Test Driven Development (TDD) (hours)
Iteration 1: Development
Iteration 2: Development
Iteration n: Development
Copyright 2003-2007 Scott W. Ambler

Depending on logistics issues (it can be difficult to get all the right people together at roughly the same time) and your organization's ability to make decisions within a reasonable timeframe, Iteration 0 may last for a period of several days to several months of calendar time. However, your initial requirements modeling effort should only take up several days of effort during that period. Also, note that there is a bit more to Iteration 0 than initial modeling -- the AMDD lifecycle of Figure 7 only depicts modeling activities. An important activity during Iteration 0 is garnering initial support and funding for the project, something that requires an understanding of the initial scope. You may have already garnered initial support via your pre-project planning efforts (part of portfolio management), but realistically at some point somebody is going to ask what are we going to get, how much is it going to cost, and how long is it going to take. You need to be able to provide reasonable, although potentially evolving, answers to these questions if you're going to get permission to work on the project. In many organizations, you may need to take it a step further and justify your project via a feasibility study.

## 2.4 Iteration Modeling

As you see in Figure 6, an agile team will implement requirements in priority order by pulling iteration's worth of work off the top of the stack. To do this successfully 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 and at what speed), you pick that much work off the stack. For example, if last iteration you accomplished 15 points worth of work, then the assumption is that, all things being equal, you'll be able to accomplish that much work this iteration. The implication is that at the beginning of each Construction iteration an agile team must estimate and schedule the work that they will do that iteration. To estimate each requirement accurately you must understand the work implementing it requires, and this is where modeling comes in. You discuss how you're going to implement each requirement, modeling where appropriate to explore or communicate ideas. This modeling in effect is the analysis and design of the requirements being implemented that iteration. Two-week iterations usually require roughly half a day of iteration planning, including modeling, and a four-week iteration will typically take a day. The goal is to plan the work accurately for the iteration, identifying the highest-priority work items to be addressed, as well as how you will do it. In other words, to think things through in the short term. The goal is *not* to produce a comprehensive Gantt chart, or detailed specifications for the work to be done. The bottom line is that an oft-neglected aspect of Mike Cohn's planning poker is the required modeling activities implied by the technique.

## 2.5 Just in Time (JIT) Model Storming

The details of these requirements are modeled on a just in time (JIT) basis in model storming sessions during the development iterations. Model storming is just in time (JIT) modeling: you identify which issue(s) you need to resolve, quickly grab a few teammates who can help you explore the issue, and then everyone continues on as before. One of the reasons why you model storm is to analyze the details of a requirement. For example, you may be implementing a user story that indicates the system you're building must be able to edit student information. The challenge is that the user story doesn't include any details as to what the screen should look like. In the agile world we like to say that user stories are "reminders to have a conversation with your stakeholders," meaning, in other words, to do some detailed requirements modeling. So, to gather the details you call your product owner over and together you create a sketch of what the screen will look like, drawing several examples until you come to a common understanding of what needs to be built. In other words, you model storm the details.

## 2.6 Non-Functional Requirements

Non-functional requirements, also known as "technical requirements" or "quality of service" (QoS) requirements, focus on aspects that typically crosscut functional requirements. Common non-functionalism includes accuracy, availability, concurrency, consumability/usability, environmental/green concerns, internationalization, operations issues, performance, regulatory concerns, reliability, security, serviceability, and supportability. Constraints, which for the sake of simplicity I will lump in with non-functionalism, define restrictions on your solution, such as being required to store all corporate data in DB2 per your enterprise architecture, or only being allowed to use open source software (OSS), which conforms to a certain level of OSS license. Constraints can often impact your technical choices by restricting specific aspects of your architecture, defining suggested opportunities for reuse, and even architectural customization points. Although many developers will bridle at this, the reality is that constraints often make things much easier for your team because some technical decisions have already been made for you. I like to think of it like this—agilists will have the courage to make tomorrow's decisions tomorrow, disciplined agilists have the humility to respect yesterday's decisions as well.

Although agile teams have pretty much figured out how to address functional requirements effectively, most are still struggling with non-functionalism. Some teams create technical stories to capture non-functionalism in a simple manner as they capture functional requirements via user stories. This is great for documentation purposes but quickly falls apart from a management and implementation point of view. The agile requirements management strategy described earlier assumes that requirements are self-contained and can be addressed in a finite period of time—an assumption that doesn't always hold true for non-functionalism.

There are four fundamental strategies, all of which should be applied in addressing non-functional requirements on an agile project:

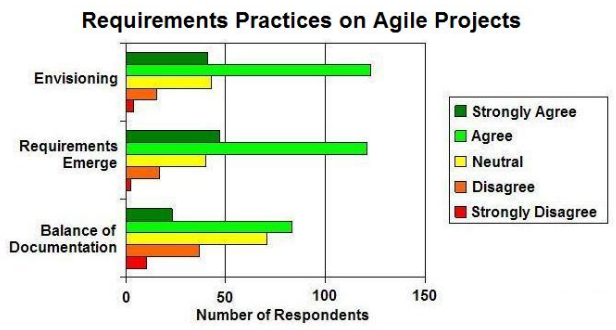
1. **Initial envisioning**. It is during your initial requirements envisioning that you will identify high-level functional requirements and non-functionalism. All forms of requirements will drive your architecture envisioning efforts, which occur iteratively in parallel with requirements envisioning. The goal of your requirements envisioning efforts is to identify the high-level requirements and the goal of your architecture envisioning efforts is to ensure that your architecture vision effectively addresses those requirements. You don't need to write detailed specifications at this point in time, but you do want to ensure that you're going in the right direction.
2. **JIT model storming.** Just in time (JIT) model storming through the construction lifecycle to explore the details
3. **Independent parallel testing**. This is performed throughout the lifecycle to ensure that the system addresses the non-functional requirements appropriately. More on this later.

4. **Education**. Developer education so that they understand the fundamentals of the full range of architectural concerns described in the requirements.

## 2.7 Who's doing this?

Figure 8 summarizes some results from Ambysoft's 2008 Agile Practice and Principles Survey. As you can see, it is quite common for agile teams to do some up-front requirements envisioning and that requirements details will emerge over time (via iteration modeling and model storming). A tools-based view is shown in Figure 9, which summarizes some results from Ambysoft's 2008 Test Driven Development (TDD) Survey. Although there is a lot of rhetoric around acceptance test-driven development (TDD) the fact is that not only hasn't it replaced agile requirements modeling techniques, it doesn't even appear to be as popular. The implication is that requirements are explored via several techniques on agile teams, and rightfully so, because one single strategy is rarely sufficient for real-world situations.

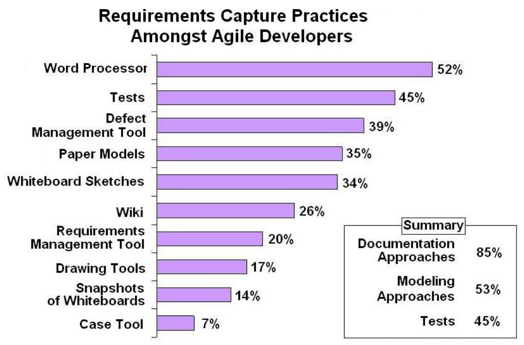
**Figure 8. Requirements practices on agile projects.**



How well do you agree with the statements:

1. We do some initial requirements modeling (envisioning) at the beginning of agile projects for scoping and planning purposes.
2. The requirements details emerge over time on our agile projects.
3. Our agile teams have an understanding of the correct balance of documentation or other artifacts for delivery.

**Figure 9. Requirements capture practices on agile teams.**



## 2.8 The Implications for Testing

There are several important implications that agile requirements strategies have for agile testing:

1. **Agile testing must be iterative**. Agile requirements activities, and design activities, and construction activities, are iterative in nature. So must testing activities.
2. **Agile testers cannot rely on having complete specifications**. As you saw in Figures 2 and 7 requirements are identified, explored, and implemented throughout the lifecycle. There isn't a single requirements phase that produces a comprehensive requirements specification, therefore your test strategies cannot rely on having a complete specification available.
3. **Agile testers must be flexible**. Testers must be prepared to work to the best of their ability, with the information provided to them at the time, and with the full understanding that the information they are basing their work on today could and probably will change tomorrow.

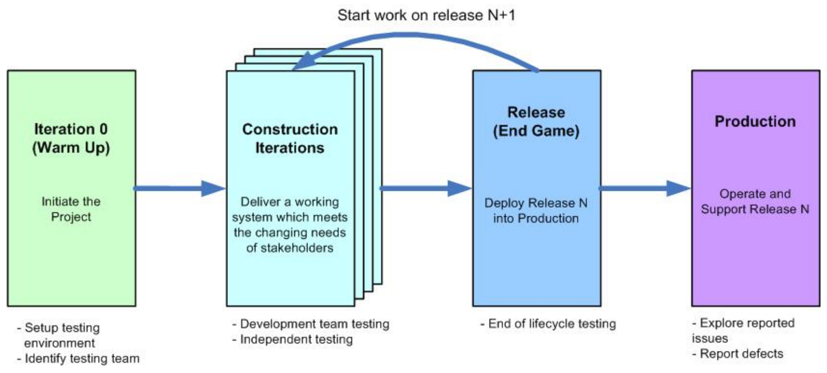
The good news is that agile testing techniques exist which reflect these implications. The challenge is that you need to be willing to adopt them.

## 3. Agile Testing Strategies

To understand how testing activities fit into agile system development it is useful to look at it from the point of view of the system delivery lifecycle (SDLC). Figure 10 is a high-level view of the agile SDLC, indicating the testing activities at various SDLC phases. This section is organized into the following topics:

- Project initiation
  - The whole team
  - The independent test team
  - Test environment setup
- Development team testing
  - Continuous integration
  - Test-driven development (TDD)
  - Test-immediately after approach
- Parallel independent testing
- Defect management
- End-of-lifecycle testing
- Who is doing this?
- Implications for test practitioners

**Figure 10. Testing throughout the SDLC.**


Start work on release N+1
Iteration 0 (Warm Up)
Initiate the Project
Construction Iterations
Deliver a working system which meets the changing needs of stakeholders
Release (End Game)
Deploy Release N into Production
Production
Operate and Support Release N
- Setup testing environment
- Identify testing team
- Development team testing
- Independent testing
- End of lifecycle testing
- Explore reported issues
- Report defects

## 3.1 Project Initiation

During project initiation, often called "Sprint 0" in Scrum or "Iteration 0" in other agile methods, your goal is to get your team going in the right direction. Although the mainstream agile community doesn't like talking about this much, the reality is that this phase can last anywhere from several hours to several weeks depending on the nature of the project and the culture of your organization. From the point of view of testing, the main tasks are to organize how you will approach testing and start setting up your testing environment if it doesn't already exist. During this phase of your project, you will be doing initial requirements envisioning (as described earlier) and architecture envisioning. As the result of that effort you should gain a better understanding of the scope, whether your project must comply to external regulations such as the Sarbanes-Oxley act or the FDA's CFR 21 Part 11 guidelines, and potentially some high-level acceptance criteria for your system – all of this is important information which should help you to decide how much testing you will need to do. It is important to remember that one process size does not fit all, and that different project teams will have different approaches to testing because they find themselves in different situations. The more complex the situation, the more complex the approach to testing (among other things). Teams finding themselves in simple situations may find that a "whole team" approach to testing will be sufficient, whereas teams in more complex situations will find that they need an independent test team, working in parallel to the development team. Regardless, there's always going to be some effort setting up your test environment.

### 3.1.1 "Whole Team" Organizational Strategy

An organizational strategy common in the agile community, popularized by Kent Beck in Extreme Programming Explained 2nd Ed, is for the team to include the right people so that they have the skills and perspectives required for the team to succeed. To deliver a working system on a regular basis successfully, the team will need to include people with analysis skills, design skills, programming skills, leadership skills, and, yes, even people with testing skills. Obviously, this isn't a complete list of skills required by the team, nor does it imply that everyone on the team has all of these skills. Furthermore, everyone on an agile team contributes in any way that they can, thereby increasing the overall productivity of the team. This strategy is called "whole team."

In a whole team approach, testers are "embedded" in the development team, actively participating in all aspects of the project. Agile teams are moving away from the traditional approach where someone has a single specialty that they focus on – for example Sally just does programming, Sanjiv just does architecture, and John just does testing – to an approach where people strive to become generalizing specialists with a wider range of skills. So, Sally, Sanjiv, and John will all be willing to be involved with programming, architecture, and testing activities and more importantly will be willing to work together and to learn from one another to become better over time. Sally's strengths may still lie in programming, Sanjiv's in architecture, and John's in testing, but that won't be the only things that they'll do on the agile team. If Sally, Sanjiv, and John are new to agile and are currently only specialists, that's ok, because by adopting non-solo development practices and working in short feedback cycles they will quickly pick up new skills from their teammates (and transfer their existing skills to their teammates as well).

This approach can be significantly different from that which traditional teams are used to. On traditional teams it is common for programmers (specialists) to write code and then "throw it over the wall" to testers (also specialists) who then test it and report suspected defects back to the programmers. Although better than no testing at all, this often proves to be a costly and time-consuming strategy due to the handoffs between the two groups of specialists. On agile teams programmers and testers work side-by-side, and over time the distinction between these two roles blur into the single role of developer. An interesting philosophy in the agile community is that real IT professionals should validate their own work to the best of their ability, thus striving to get better at doing so over time.

The whole team strategy isn't perfect, and there are several potential problems:

1. **Groupthink**. Whole teams, just like any other type of team, can suffer from what's known as "groupthink." The basic issue is that because members of the team are working closely together, something that agile teams strive to do, they begin to think alike and as a result will become blind to some problems that they face. For example, the system that your team is working on could have some significant usability problems in it but nobody on the team is aware of them because either nobody on the team had usability skills to begin with or the team decided, mistakenly, to downplay usability concerns at some point and team members have forgotten that it happened.
2. **The team may not actually have the skills it needs**. It's pretty easy to declare that you're following the "whole team" strategy but not so easy to actually do so sometimes. Some people may overestimate their skills, or underestimate what skills are actually needed, and thereby put the team at risk. For example, a common weakness amongst programmers lies in database design skills; or, rather, the lack thereof. This results from over-specialization in organizations where "data professionals" focus on data skills and programmers do not. Hence, a cultural impedance mismatch is erected between developers and data professionals, making it difficult to pick up data skills. Worse yet, database encapsulation strategies such as object-relational mapping frameworks like Hibernate motivate these "data-challenged" programmers to believe that they don't need to know anything about database design because they enjoy encapsulated access to it. Consequently, the individual team members, falsely believing they possess the skills to

get the job done, in fact do not, and, in this case, the system will suffer because the team lacks the requisite skills to design and use the database appropriately.

3. **The team may not know what skills are needed**. Even worse, the team may be oblivious to what skills are actually needed. For example, the system could have some very significant security requirements but if the team didn't know security was likely to be a concern then they could miss those requirements completely, or misunderstand them, and inadvertently put the project at risk.

Luckily, the benefits of the whole team approach outweigh the potential problems. First, whole team appears to increase overall productivity by reducing and often eliminating the wait time between programming and testing inherent in traditional approaches. Second, there is less need for paperwork such as detailed test plans due to the lack of hand-offs between separate teams. Third, programmers quickly start to learn testing and quality skills from the testers and as a result do better work to begin with – when the developer knows that they'll be actively involved with the testing effort they are more motivated to write high-quality, testable code to begin with.

### 3.1.2 The Independent Test Team (Advanced Strategy)

The whole team approach works well in practice when agile development teams find themselves in reasonably straightforward situations. However, when the environment is complex, either because the problem domain is itself inherently complex, the system is large (often the result of supporting a complex domain), or it needs to integrate into your overall infrastructure, which includes a myriad of other systems, then a whole team approach to testing proves insufficient. Teams in complex environments, as well as teams which find themselves in regulatory compliance situations, will often discover that they need to supplement their whole team with an independent test team. This test team will perform parallel independent testing throughout the project and will typically be responsible for the end-of-lifecycle testing performed during the release/transition phase of the project. The goal of these efforts is to find out where the system breaks (whole team testing often focuses on confirmatory testing which shows that the system works) and report such breakages to the development team so that they can fix them. This independent test team focuses on complex forms of testing at a level beyond “whole team’s” ability to conduct on its own. More on this later.

Your independent test team will support multiple project teams. Most organizations have many development teams working in parallel, often dozens of teams and sometimes even hundreds, so you can achieve economies of scale by having an independent test team support many development teams. This allows you to minimize the number of testing tool licenses that you need, share expensive hardware environments, and enable testing specialists (such people experienced in usability testing or investigative testing) to support many teams.

It's important to note that an *agile* independent test team works significantly differently than a traditional independent test team. The agile independent test team focuses on a small minority of the testing effort, the hardest part of it, while the development team does the majority of the testing grunt work. With a traditional approach, the test team would often do both the grunt work as well as the complex forms of testing. Placed into perspective: an optimum ratio of 15:1 or 20:1 (development team members to independent test team members) is reasonable in an agile model, whereas in the traditional model these ratios are often closer to 3:1 or 1:1 (and in regulatory environments may be 1:2 or more).

### 3.1.3 Testing Environment Setup

At the beginning of your project, you will need to start setting up your environment, including setting up your work area, your hardware, and your development tools (to name a few things). You will naturally need to set up your testing environment, from scratch if you don't currently have such an environment available or through tailoring an existing environment to meeting your needs. There are several strategies that I typically suggest when it comes to organizing your testing environment:

1. **Adopt open source software (OSS) tools for developers**. There are a lot of great OSS testing tools available, such as the xUnit testing framework targeted at developers. These tools are easy to install and use (although learning how to test effectively proves to be another matter).
2. **Adopt commercial tools for independent testers**. Because your independent test team will often address more complex testing issues, such as integration testing across multiple systems (not just the single system that the development team is focused on), security testing, usability testing, and so on they will need testing tools which are sophisticated enough to address these issues.
3. **Have a shared bug/defect tracking system**. As you see in Figure 2, and as I've mentioned in the previous section, the independent test team will send defect reports back to the development team, who in turn will often consider these defect reports to be just another type of requirement. The implication is that they need to have tooling support to do this. When the team is small and co-located, as we see at Agile Process Maturity Model (APMM) levels 1 and 2, this could conceivably be as simple as a stack of index cards. In more complex environments you'll need a software-based tool, ideally one that is used to manage both the team's requirements as well as the team's defects (or more importantly, their entire work item list). More on this later.
4. **Invest in testing hardware**. Both the development team and the independent team will need hardware on which to do testing.
5. **Invest in virtualization and test lab management tools**. Test hardware is expensive and there's never enough. Virtualization software which enables you to load a testing environment onto your hardware easily, as well as test-lab management tools which enable you to track hardware configurations, are critical to the success of your independent test team (particularly if they're support multiple development teams).
6. **Invest in continuous integration (CI) and continuous deployment (CD) tools**. Not explicitly a category of testing tool. but CI/CD tools are critical for agile development teams because of practices such as test-driven development (TDD), developer regression testing in general, and the need to deploy working builds into independent testing environments, demo environments, or even production. Open source tools such as Maven and CruiseControl work well in straightforward situations, although in more complex situations (particularly at scale), you'll find that you need commercial CI/CD tools.

## 3.2 Development Team Testing Strategies

Agile development teams generally follow a whole team strategy where people with testing skills are effectively embedded into the development team and the team is responsible for the majority of the testing. This strategy works well for the majority of situations but when your environment is more complex you'll find that you also need an independent test team working in parallel to the development and potentially performing end-of-lifecycle testing as well. Regardless of the situation, agile development teams will adopt practices such as continuous integration which enables them to do continuous regression testing, either with a test-driven development (TDD) or test-immediately after approach.

### 3.2.1 Continuous Integration (CI)

Continuous integration (CI) is a practice where at least once every few hours, preferably more often, you should:

1. **Build your system**. This includes both compiling your code and potentially rebuilding your test database(s). This sounds straightforward, but for large systems that are composed of subsystems you need a strategy for how you're going to build both the subsystems as well as the overall system (you might only do a complete system build once a night, or once a week for example, and only build subsystems on a regular basis).
2. **Run your regression test suite(s)**. When the build is successful, the next step is to run your regression test suite(s). Which suite(s) you run will be determined by the scope of the build and the size of the system. For small systems, you'll likely have a single test suite of all tests, but in situations that are more complex, you will have several test suites for performance reasons. For example, if I am doing a build on my own workstation then I will likely only run a subset of the tests which validate the functionality that the team has been working on lately (say for the past several iterations) as that functionality is most likely to break and may even run tests against "mock objects". Because I am likely to run this test suite several times a day it needs to run fast, preferably in less than 5 minutes although some people will allow their developer test suite to go up to ten minutes. Mock objects might be used to simulate portions of the system that are expensive to test, from a performance point of view, such as a database or an external subsystem. If you're using mocks or if you're testing a portion of the functionality, then you'll need one or more test suites of greater scope. Minimally you'll need a test suite which runs against the real system components, not mocks, and that runs all your tests. If this test suite runs within a few hours then it can be run nightly, if it takes longer then you'll want to reorganize it again so that very long-running tests are in another test suite that runs irregularly. For example, I know of one system that has a long-running test suite that runs for several months at a time in order to run complex load and availability tests (something that an independent test team is likely to do).
3. **Perform static analysis**. Static analysis is an quality technique where an automated tool checks for defects in the code, often looking for types of problems such as security defects or coding style issues (to name a few).

Your integration job could run at specific times, perhaps once an hour, or every time that someone checks in a new version of a component (such as source code) which is part of the build.

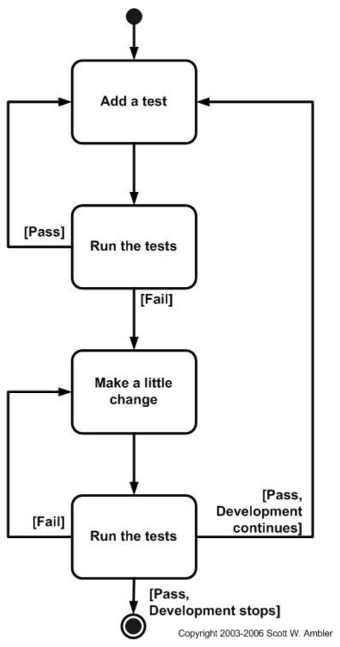
Advanced teams, particularly those in an agility at scale situation, will find that they also need to consider continuous deployment as well. The basic idea is that you automate the deployment of your working build, some organizations refer to this as promotion of their working build, into other environments on a regular basis. For example, if you have a successful build at the end of the week you might want to automatically deploy it to a staging area so that it can be picked up for parallel independent testing. Or if there's a working build at the end of the day you might want to deploy it to a demo environment so that people outside of your team can see the progress that your team is making.

### 3.2.2 Test-Driven Development (TDD)

Test-driven development (TDD) is an agile development technique practice that combines:

1. **Refactoring**. Refactoring is a technique where you make a small change to your existing source code or source data to improve its design without changing its semantics. Examples of refactorings include moving an operation up the inheritance hierarchy and renaming an operation in application source code; aligning fields and applying a consistent font on user interfaces; and renaming a column or splitting a table in a relational database. When you first decide to work on a task, you look at the existing source and ask if it is the best design possible to allow you to add the new functionality that you're working on. If it is, then proceed with TFD. If not, then invest the time to fix just the portions of the code and data so that it is the best design possible so as to improve the quality and thereby reduce your technical debt over time.
2. **Test-first development (TFD)**. With TFD you write a single test and then you write just enough software to fulfill that test. The steps of test first development (TFD) are overviewed in the UML activity diagram of Figure 11. The first step is to add a test quickly, basically just enough code to fail. Next, you run your tests, often the complete test suite although for sake of speed you may decide to run only a subset, to ensure that the new test does in fact fail. You then update your functional code to make it pass the new tests. The fourth step is to run your tests again. If they fail you need to update your functional code and retest. Once the tests pass, the next step is to start over.

**Figure 11. The Steps of test-first development (TFD).**

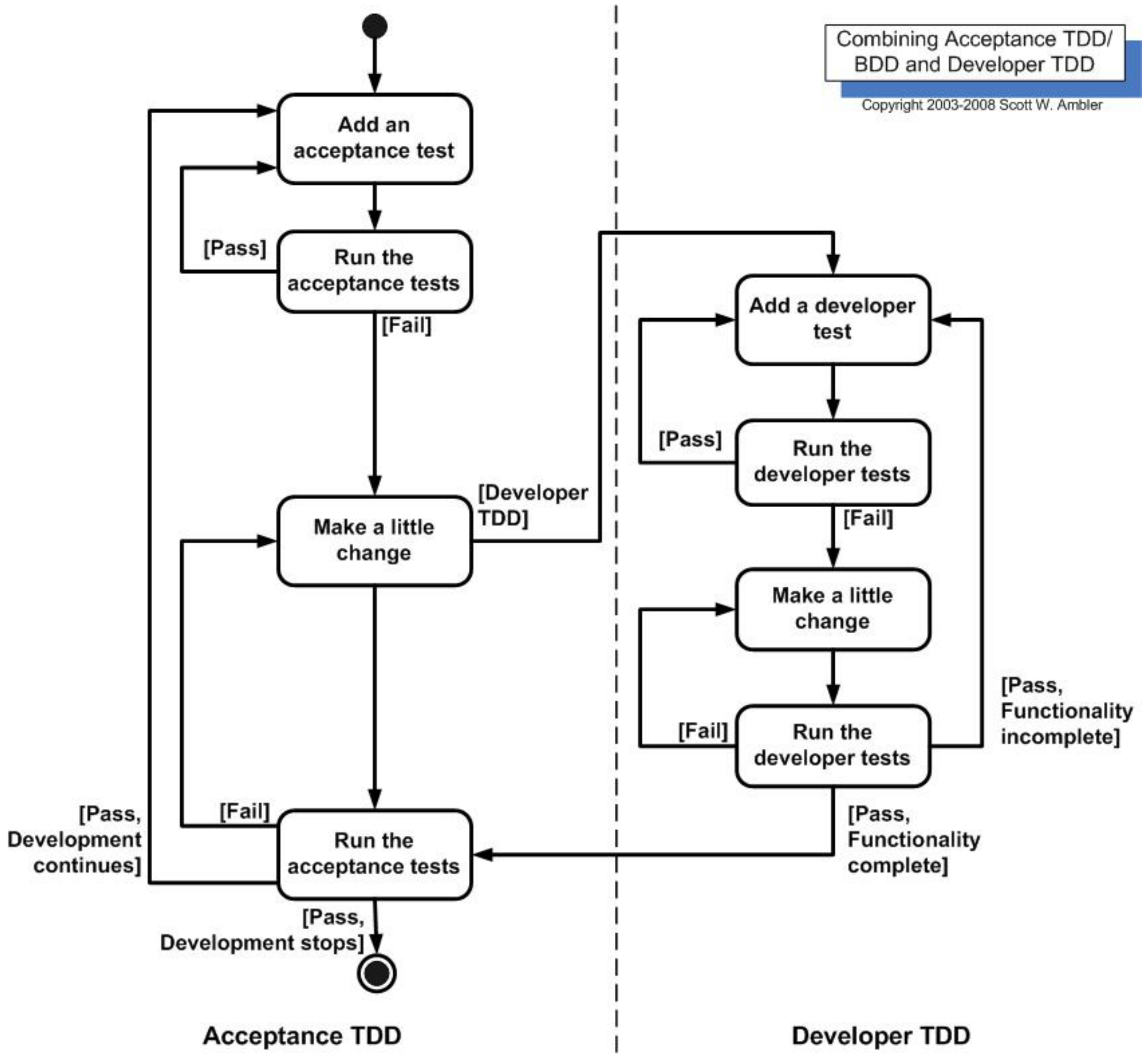

Add a test
Run the tests
[Pass]
[Fail]
Make a little change
Run the tests
[Fail]
[Pass, Development continues]
[Pass, Development stops]
Copyright 2003-2006 Scott W. Ambler

There are two levels of TDD:

1. **Acceptance TDD**. You can do TDD at the requirements level by writing a single customer test, the equivalent of a function test or acceptance test in the traditional world. Acceptance TDD is often called behavior-driven development (BDD) or story test-driven development, where you first automate a failing story test, then driving the design via TDD until the story test passes (a story test is a customer acceptance test).
2. **Developer TDD**. You can also do TDD at the design level with developer tests.

With a test-driven development (TDD) approach, your tests effectively become detailed specifications that are created on a just-in-time (JIT) basis. Like it or not, most programmers don't read the written documentation for a system, instead they prefer to work with the code. And there's nothing wrong with this. When trying to understand a class or operation most programmers will first look for sample code that already invokes it. Well-written unit/developers tests do exactly this – they provide a working specification of your functional code – and as a result, unit tests effectively become a significant portion of your technical documentation. Similarly, acceptance tests can form an important part of your requirements documentation. This makes a lot of sense when you stop and think about it.  Your acceptance tests define exactly what your stakeholders expect of your system, therefore they specify your critical requirements. The implication is that acceptance TDD is a JIT detailed requirements specification technique and developer TDD is a JIT detailed design specification technique. Writing executable specifications is one of the best practices of Agile Modeling.

**Figure 12. How ATDD and developer TDD fit together.**



With ATDD you are not required to also take a developer TDD approach to implementing the production code although the vast majority of teams doing ATDD also do developer TDD. As you see in Figure 12, when you combine ATDD and developer TDD the creation of a single acceptance test in turn requires you to iterate several times through the write a test, write production code, get it working cycle at the developer TDD level. Clearly to make TDD work you need to have one or more testing frameworks available to you. For acceptance TDD people will use tools such as Fitnesse or RSpec and for developer TDD agile software developers often use the xUnit family of open source tools, such as JUnit or VBUnit. Commercial testing tools are also viable options. Without such tools, TDD is virtually impossible. The greatest challenge with adopting ATDD is lack of skills amongst existing requirements practitioners, yet another reason to promote generalizing specialists within your organization over narrowly focused specialists.

### 3.2.3 The Test-Immediately After Approach

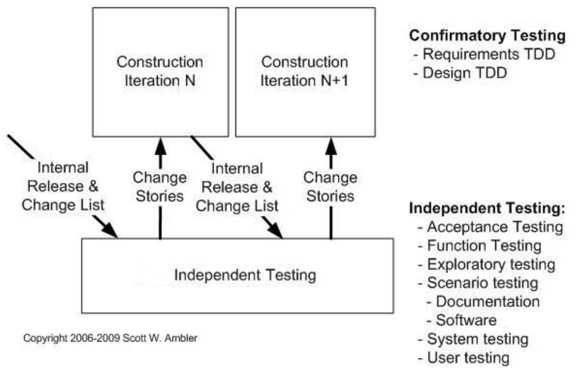
Although many agilists talk about TDD, the reality is that there seems to be far more doing "test after" development where they write some code and then write one or more tests to validate. TDD requires significant discipline; in fact, it requires a level of discipline found in few coders, particularly coders that follow solo approaches to development instead of non-solo approaches such as pair programming. Without a pair keeping you honest, it's pretty easy to fall back into the habit of writing production code before writing testing code. If you write the tests very soon after you write the production code, in other words "test immediately after", it's pretty much as good as TDD, the problem occurs when you write the tests days or weeks later if at all.

The popularity of code coverage tools such as Clover and Jester amongst agile programmers is a clear sign that many of them really are taking a "test after" approach. These tools warn you when you've written code that doesn't have coverage tests, prodding you to write the tests that you would hopefully have written first via TDD.

## 3.3 Parallel Independent Testing

The whole team approach to development where agile teams test to the best of the ability is a great start, but it isn't sufficient in some situations. In these situations, described below, you need to consider instituting a parallel independent test team that performs some of the more difficult (or perhaps advanced is a better term) forms of testing. As you can see in Figure 13, the basic idea is that on a regular basis, the development team makes their working build available to the independent test team, or perhaps they automatically deploy it via their continuous integration tools, so that they can test it. The goal of this testing effort is not to redo the confirmatory testing that is already being done by the development team, but instead to identify the defects that have fallen through the cracks. The implication is that this independent test team does not need a detailed requirements speculation, although they may need architecture diagrams, a scope overview, and a list of changes since the last time the development team sent them a build. Instead of testing against the specification, the independent testing effort will focus on production-level system integration testing, investigative testing, and formal usability testing to name a few things.

**Figure 13. Parallel independent testing.**



**Important:** The development team is still doing the majority of the testing when an independent test team exists. It is just that the independent test team is doing forms of testing that either the development doesn't (yet) have the skills to perform or is too expensive for them to perform.

The independent test team reports defects back to the development, indicated as "change stories" in Figure 12 because being agile we tend to rename everything ;-) . These defects are treated as type of requirement by the development team in that they're prioritized, estimated, and put on the work item stack.

There are several reasons why you should consider parallel independent testing:

1. **Investigative testing**. Confirmatory testing approaches, such as TDD, validate that you've implemented the requirements as they've been described to you. But what happens when requirements are missing? User stories, a favorite requirements elicitation technique within the agile community, are a great way to explore functional requirements but defects surrounding non-functional requirements such as security, usability, and performance have a tendency to be missed via this approach.
2. **Lack of resources**. Furthermore, many development teams may not have the resources required to perform effective system integration testing, resources that 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) that addresses these sorts of issues. System integration tests often require expensive environment that goes beyond what an individual project team will have
3. **Large or distributed teams**. Large or distributed teams are often subdivided into smaller teams, and when this happens system integration testing of the overall system can become complex enough that a separate team should consider taking it on. In short, whole team testing works well for agile in the small, but for more complex systems and agile at scale you need to be more sophisticated.
4. **Complex domains**. When you have a very complex domain, perhaps you're working on life critical software or on financial derivative processing; whole team testing approaches can prove insufficient. Having a parallel independent testing effort can reduce these risks.
5. **Complex technical environments**. When you're working with multiple technologies, legacy systems, or legacy data sources the testing of your system can become very difficult.
6. **Regulatory compliance**. Some regulations require you to have an independent testing effort. My experience is that the most efficient way to do so is to have it work in parallel to the development team.
7. **Production readiness testing**. The system that you're building must "play well" with the other systems currently in production when your system is released. To do this properly you must test against versions of other systems that are currently under development, implying that you need access to updated versions on a regular basis. This is fairly straightforward in small organizations, but if your organization has dozens, if not hundreds of IT projects underway it becomes overwhelming for individual development

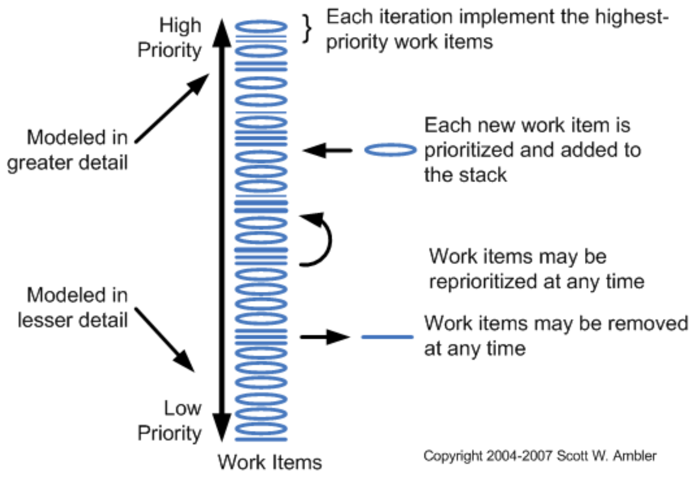
teams to gain such access. A more efficient approach is to have an independent test team be responsible for such enterprise-level system integration testing.

Some agilists will claim that you don't need parallel independent testing, and in simple situations, this is clearly true. The good news is that it's incredibly easy to determine whether or not your independent testing effort is providing value: simply compare the likely impact of the defects/change stories being reported with the cost of doing the independent testing.

## 3.4 Defect Management

Defect management is often much simpler on agile projects when compared to classical/traditional projects for two reasons. First, with a whole team approach to testing when a defect is found it's typically fixed on the spot, often by the person(s) who injected it in the first place. In this case, the entire defect management process is at most a conversation between a few people. Second, when an independent test team is working in parallel with the development team to validate their work they typically use a defect reporting tool, such as IBM Rational ClearQuest or Bugzilla, to inform the development team of what they found. Disciplined agile delivery teams combine their requirements management and defect management strategies to simplify their overall change management process. Figure 14 summarizes this (yes, it's the same as Figure 6) showing how work items are worked on in priority order. Both requirements and defect reports are types of work items and are treated equally -- they're estimated, prioritized, and put on the work item stack.

**Figure 14. Agile defect change management process.**


High Priority
Each iteration implement the highest-priority work items
Modeled in greater detail
Each new work item is prioritized and added to the stack
Work items may be reprioritized at any time
Modeled in lesser detail
Work items may be removed at any time
Low Priority
Work Items
Copyright 2004-2007 Scott W. Ambler

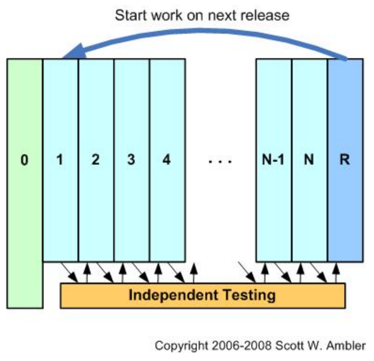
This works because defects are just another type of requirement. Defect X can be reworded into the requirement "please fix X." Requirements are also a type of defect; in effect, a requirement is simply missing functionality. In fact, some agile teams will even capture requirements using a defect management tool, for example the Eclipse development team uses Bugzilla, and the Jazz development team uses Rational Team Concert (RTC).

The primary impediment to adopting this strategy – that of treating requirements and defects as the same thing – occurs when the agile delivery team finds itself in a fixed-price or fixed estimate situation. In such situations, the customer typically needs to pay for new requirements that weren't agreed at the beginning of the project. Even so, he should not pay for fixing defects. In such situations the bureaucratic approach would be to have two separate change management processes and the pragmatic approach would be to simply mark the work item as something that needs to be paid extra for (or not). Naturally, I favor the pragmatic approach. If you find yourself in a fixed-price situation you might be interested that I've written a fair bit about this and more importantly alternatives for funding agile projects. To be blunt, I vacillate between considering fixed-price strategies as unethical or simply as a sign of grotesque incompetence on the part of the organization insisting on it. Merging your requirements and defect management processes into a single, simple change management process is a wonderful opportunity for process improvement. Exceptionally questionable project funding strategies shouldn't prevent you from taking advantage of this strategy.

## 3.5 End of Lifecycle Testing

An important part of the release effort for many agile teams is end-of-lifecycle 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-lifecycle testing can be very short, as the issues have already been substantially covered. As you see in Figure 15 the independent testing efforts stretch into the release phase (called the Transition phase of Disciplined Agile Delivery) of the delivery life cycle because the independent test team will still need to test the complete system once it's available.

**Figure 15. Independent testing throughout the lifecycle.**


Start work on next release
0
1
2
3
4
...
N-1
N
R
Independent Testing
Copyright 2006-2008 Scott W. Ambler

There are several reasons why you still need to do end-of-lifecycle testing:

1. **It's professional to do so**. You'll minimally want to do one last run of all of your regression tests in order to be in a position to declare, officially, that your system is fully tested. This clearly would occur once iteration N, the last construction iteration, finishes (or would be the last thing you do in iteration N, don't split hairs over stuff like this).
2. **You may be legally obligated to do so**. Either because of the contract that you have with the business customer or due to regulatory compliance (many agile teams find themselves in such situations, as the November 2009 State of the IT Union survey discovered).

3. **Your stakeholders require it**. Your stakeholders, particularly your operations department, will likely require some sort of testing effort before releasing your solution into production in order to feel comfortable with the quality of your work.

There is little publicly discussed in the mainstream agile community about end-of-lifecycle testing, the assumption of many people following mainstream agile methods such as Scrum is that techniques such as TDD are sufficient. This might be because much of the mainstream agile literature focuses on small, co-located agile development teams working on fairly straightforward systems. When one or more scaling factors (large team size, geographically distributed teams, regulatory compliance, or complex domains) are applicable, sophisticated testing strategies will be required. Regardless of some of the rhetoric, you may have heard in public, as we see in the next section a fair number of TDD practitioners are indicating otherwise in private.

## 3.6 Who is doing This?

Figure 16 summarizes the results of one of the questions from Ambysoft's 2008 Test Driven Development (TDD) Survey that asked the TDD community which testing techniques they were using in practice. Because this survey was sent to the TDD community it doesn't accurately represent the adoption rate of TDD at all, but what is interesting is the fact that respondents clearly indicated that they weren't only doing TDD (nor was everyone doing TDD, surprisingly). Many were also doing reviews and inspections, end of lifecycle testing, and parallel independent testing, activities that the agile purists rarely seem to discuss. Furthermore, Figure 17, which summarizes results from the "2010 How Agile Are You?" survey, provides insight into which validation strategies are being followed by the teams claiming to be agile. I suspect that the adoption rates reported for developer TDD and acceptance TDD, 53% and 44% respectively, are much more realistic than those reported in Figure 16.

**Figure 16. Testing/Validation practices on agile teams.**

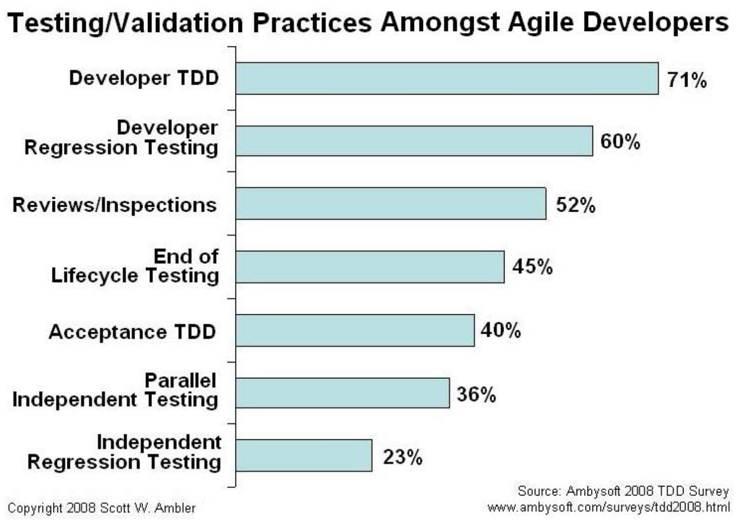
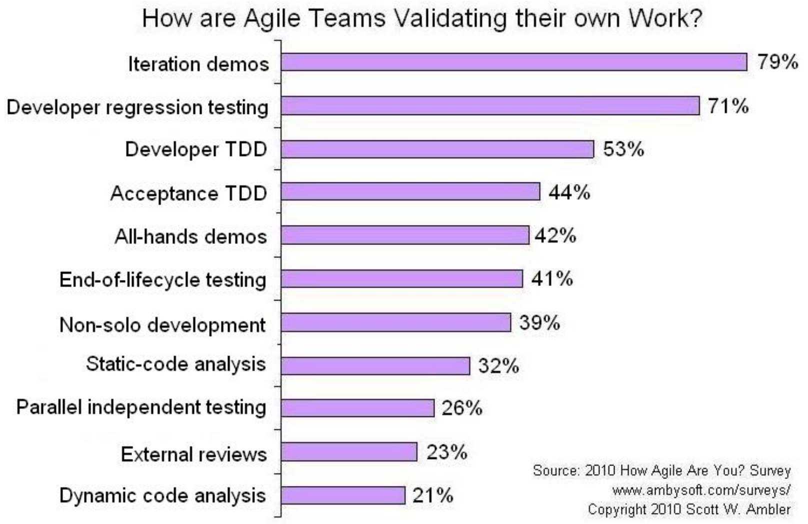


**Figure 17. How agile teams validate their own work.**



## 3.7 Implications for Test Practitioners

There are several critical implications for existing test professionals:

1. **Become generalizing specialists**. The implication of whole team testing is that most existing test professionals will need to be prepared to do more than just testing if they want to be involved with agile projects. Yes, the people on independent test teams would still focus solely on testing, but the need for people in this role is much less than the need for people with testing skills to be active members of agile delivery teams.
2. **Be flexible**. Agile teams take an iterative and collaborative approach that embraces changing requirements. The implication is that gone are the days of having a detailed requirements speculation to work from, now anyone involved with testing must be flexible enough to test throughout the entire life cycle even when the requirements are changing.
3. **Be prepared to work closely with developers**. The majority of the testing effort is performed by the agile delivery team itself, not by independent testers.
4. **Be flexible**. This is worth repeating. ;-)
5. **Focus on value-added activities**. I've lost track of the number of times I've heard test professionals lament that there's never enough time or resources allocated to testing efforts. Yet, when I explore what these people want to do, I find that they want to wait to have detailed requirements speculations available to them, they want to develop documents describing their test strategies, they want to write detailed test plans, they want to write detailed defect reports, and yes, they even want to write and then run tests. No wonder they don't have enough time to get all this work done! The true value added activities that testers provide are finding and then communicating potential defects to the people responsible for fixing them. To do this they clearly need to create and run tests, all the other activities that I listed previously are ancillary at best to this effort. Waiting for requirements speculations isn't testing. Writing test strategies and plans aren't testing. Writing defect reports might be of value, but there are better ways to communicate

information than writing documentation. Agile strategies focus on the value-added activities and minimize if not eliminate the bureaucratic waste which is systemic in many organizations following classical/traditional strategies.

6. **Be flexible**. This is really important.

## 4. Agile Quality Strategies

In addition to agile testing strategies, there are also agile quality strategies. These strategies include:

- Refactoring
- Non-solo development
- Static and dynamic code analysis
- Reviews and inspections
    - Iteration/sprint demos
    - All-hands demos
    - Light-weight milestone reviews
- Short feedback cycles
- Standards and guidelines

### 4.1 Refactoring

Refactoring is a disciplined way to restructure your code to improve its quality. The basic idea is that you make small changes to your code to improve its design, making it easier to understand and to modify. Refactoring enables you to evolve your code slowly over time, to take an iterative and incremental approach to programming. A critical aspect of a refactoring is that it retains the behavioral semantics of your code, at least from a black box point of view. For example there is a very simple refactoring called *Rename Method*, perhaps from *getPersons()* to *getPeople()*. Although this change looks easy on the surface you need to do more than just make this single change, you must also change every single invocation of this operation throughout all of your application code to invoke the new name. Once you've made these changes then you can say you've truly refactored your code because it still works again as before. It is important to understand that you do not add functionality when you are refactoring. When you refactor you improve existing code, when you add functionality you are adding new code. Yes, you may need to refactor your existing code before you can add new functionality. Yes, you may discover later on that you need to refactor the new code that you just added. The point to be made is that refactoring and adding new functionality are two different but complementary tasks.

Refactoring applies not only to code, but to your database schema as well. A database refactoring is a simple change to a database schema that improves its design while retaining both its behavioral and informational semantics. For the sake of this discussion, a database schema includes both structural aspects such as table and view definitions as well as functional aspects such as stored procedures and triggers. An interesting thing to note is that 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. There is a database refactoring named *Split Column*, one of many described in A Catalog of

Database Refactorings, where you replace a single table column with two or more other columns. For example you are working on the *Person* table in your database and discover that the *FirstDate* column is being used for two distinct purposes – when the person is a customer this column stores their birth date and when the person is an employee it stores their hire date. Your application now needs to support people who can be both a customer and an employee so you've got a problem. Before you can implement this new requirement, you need to fix your database schema by replacing the *FirstDate* column with *BirthDate* and *HireDate* columns. To maintain the behavioral semantics of your database schema you need to update all source code that accesses the *FirstDate* column to work now with the two new columns. To maintain the informational semantics you will need to write a migration script that loops through the table, determines the type, and then copies the existing date into the appropriate column. Although this sounds easy, and sometimes it is, my experience is that database refactoring is incredibly difficult in practice when cultural issues are taken into account.

Refactoring also applies to your user interface, as Rusty Harold aptly shows in Refactoring HTML. Simple changes to your user interface, such as *Align Fields* and *Apply Common Size*, can improve the quality of the look and feel of your user interface.

## 4.2 Non-Solo Development

With non-solo development approaches, such as XP's pair programming or Agile Modeling's Model With Others, two or more people work together on a single activity. In many ways, non-solo development is the agile implementation of the old axiom "two heads are better than one." With pair-programming, two people literally sit together at a single workstation, one person writing the code while the other looks over their shoulder providing ideas and ensuring that the coder follows development conventions and common quality practices such as test-driven development (TDD). The pair programmers will shift roles on a regular basis, keeping a steady pace. When modeling with others, two or more people will gather around a shared modeling environment such as a whiteboard and work together to explore a requirement or to think through a portion of the design.

Of course, non-solo development isn't just limited to programming and modeling; you can and should work collaboratively on all aspects of an IT project. For example, agile teams will often do detailed project planning as a team, typically on a just-in-time (JIT) basis at the beginning of an iteration/sprint.

There are several key benefits to non-solo development:

1. **Increased quality**. In many ways, non-solo development is simply a continuous form of inspection/review. Because two or more people are working on an artifact simultaneously, if one person injects a defect then it is very likely that that defect will be caught at the point it is injected due to the short feedback cycle.
2. **Increased skill and knowledge sharing**. When people work closely together they learn skills and new knowledge from one another, which over time improves their individual productivity as well as that of the group as a whole due to improved ability to relate to

and interact with one another. This is particularly true when the pairs change often, a practice called "promiscuous pairing."

3. **Improved team cohesion**. Non-solo development is sometimes considered a common success factor for building a true whole team.
4. **Reduced need for documentation**. Because of the increased knowledge sharing amongst team members, and because of the improved quality of the work products, there is less need for external documentation to be produced. Yes, you will still need to write some documentation, therefore it is still important adopt some agile documentation practices.
5. **More actual work often occurs**. An interesting side effect of pair programming is that practitioners report that they're exhausted after 5 or 6 hours of doing it. That's because they're actually working! A few years ago I worked with an organization that believed that pair programming was a bad idea because it would cut productivity in half. I argued that if the pairs were exhausted after 6 hours then they would be getting on average 3 hours of solid, high-quality work out of their programmers every day. They said that wasn't acceptable because they were already getting 8 hours a day out of their programmers. I doubted that was the case, so to prove my point we walked around the office at a specific point in the day (about 10:30) and counted the number of programmers currently coding (we assumed that anyone with code on the screen who had their bodies oriented towards the screen was coding) and the number of programmers in their cubes not coding. For every programmer coding there were four or five not coding, so assuming an eight to ten hour day they appeared to be getting 1.5 to 2 hours of programming work per day out of their coders (and we knew nothing about the quality). My understanding is that pair programming is now an acceptable practice within that organization.

There are two unfortunate misperceptions about non-solo development:

1. **It decreases productivity**. There are some studies which show that pair programming decreases certain aspects of productivity, such as overall time to get something done (several stories show that pairing increases time to value), but I don't know of any which show a decrease in long term productivity where all factors are taking into account (see the list of benefits above). So, the jury is still out as far as I'm concerned.
2. **It is uncomfortable for many IT professionals**. This can be a problem, not everyone is a people person, although "many" may be the wrong word. It's definitely uncomfortable for many people at first, although the vast majority of those people seem to adapt pretty quickly once they get some experience with non-solo development strategies.

I've run into many organizations where philosophical debates rage about the benefits and potential drawbacks of non-solo development techniques, yet it is often the case that these debates are theoretical in nature because the people involved really haven't tried them in practice. Here's my advice:

1. **Give non-solo development an honest try**. My advice is for a team to decide amongst themselves to give non-solo development practices a try for a solid month. During that period solo development practices, such as coding alone in your cube, will not be

tolerated. You need to give it that amount of time because there will be some problems, which you'll need to overcome, it will take time to figure out how to make some aspects of non-solo development work in your environment, and it will take a bit of time to observe whether it works for you. Then at the end of the month decide as a team whether you want to continue with non-solo development. You'll find that 95% or more of the team won't want to go back.

2. **Observe what you do to solve difficult problems**. Whenever you run into a problem, such as not knowing the best way to implement something, you often go and ask someone for help, don't you? So, if this were your strategy when things are difficult, why wouldn't it also be your strategy at other times? More importantly, why wouldn't you always be working on something difficult for you, and thereby learning new things?
3. **Talk with others who have succeeded at non-solo development**. Many organizations have succeeded at this and you can too if you give it a chance.

## 4.3 Static and Dynamic Code Analysis

Static code analysis tools check for defects in the code, often looking for types of problems such as security defects, which 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. Dynamic code analysis is a bit more complicated in that it examines the executing code for problems. Both forms of code analysis are 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 should be used:

1. **As part of your continuous integration efforts**. As part of your integration strategy you may choose to run static or dynamic code analysis as part of every build, but more realistically you may choose to do it at specific points in time (perhaps as part of a nightly or weekly build).
2. **In outsourcing projects**. Performing static or dynamic code analysis on the ongoing work of the outsourcing firm, perhaps on a weekly basis, is an easy strategy for ensuring that the source code produced by the outsourcing firm meets your quality standards.
3. **In regulatory compliance situations**. Many regulations insist that you prove that your work meets certain levels of quality, and code analysis can help to fulfill the burden of proof, at least partially.

As you saw in Figure 17, which summarizes results from the "2010 How Agile Are You?" survey, 32% of respondents claiming to be on agile teams included static code analysis in their builds and 21% dynamic code analysis. It's a start.

## 4.4 Reviews and Inspections

A review – specific approaches to reviews include walkthroughs and inspections – is a validation technique in which an artifact is examined critically by a group of your peers. The basic idea is that a group of qualified people evaluates an artifact to determine if the artifact not only fulfills the demands of its target audience, but is also of sufficient quality to be easy to develop, maintain, and enhance. Potential problems/defects are reported to the creator of the artifact so that they may potentially be addressed.

First and foremost, I consider the holding of reviews and inspections to be a process smell which indicates that you have made a mistake earlier in the life cycle which could be very likely rectified by another, more effective strategy (non-solo development, for example) which has a much shorter feedback cycle. This goes against traditional wisdom which says that reviews are an effective quality technique, an opinion which is in fact backed up by research – in the book Practical Software Metrics for Project Management and Process Improvement Robert Grady reports that project teams taking a serial (non-agile) approach that 50 to 75 percent of all design errors can be found through technical reviews. I have no doubt that this is true, but my point is that although reviews are effective quality techniques in non-agile situations that in agile situations there are often much better options available to you than reviews, so therefore you should strive to adopt those quality techniques instead.

There are several situations where it makes sense to hold reviews:

1. **Regulatory requirements**. When your project is subject to regulations, such as the Food and Drug Administration (FDA)'s 21 CFR Part 11, you may by law be required to hold some reviews. My advice is to read the relevant regulations carefully, determine if you actually are subject to them, and if so how much additional work actually needs to be done to comply with the regulations. If you let the bureaucrats interpret the regulations you will likely end up with an overly bureaucratic process.
2. **To provide stakeholders with a consistent opportunity to provide feedback about work in progress**. This is the primary goal of iteration/sprint demos.
3. **To prove to stakeholders that things are going well**. Consider holding an "all-hands demo" to do this
4. **To ensure your strategy is viable**. Consider holding light-weight milestone reviews.
5. **A work product wasn't created collaboratively**. I'm eager to review artifacts where only one or two people have actively worked on them AND which can't be proven with code (e.g. user manuals, operations documents, etc.). These situations do occur, perhaps only one person on your team has technical writing skills so they've taken over the majority of your documentation efforts. However, if this is actually the case then you should really consider adopting non-solo development techniques to spread this skill around a little more widely.

As you saw in Figure 17, which summarizes results from the "2010 How Agile Are You?" survey, 23% of respondents claiming to be on agile teams indicated that they do external reviews of their work periodically.

### 4.4.1 Iteration/Sprint Demo

As you saw in Figure 2, a common practice of agile teams is to hold a demo of their working solution to date at the end of each iteration/sprint. The goals are to show explicit progress to key stakeholders and to obtain feedback from them. This demo is in effect an informal review, although agile teams working in regulatory environments will often choose to be a bit more formal. As you saw in Figure 17, which summarizes results from the “2010 How Agile Are You?” survey, 79% of respondents claiming to be on agile teams indicated that they do iteration/sprint demos.

### 4.4.2 All-Hands Demos

Although several stakeholders may be working directly with the team, we could have hundreds or even thousands that don’t know what’s going on. As a result, I will often run an "all-hands" demo/review early in the delivery life cycle, typically two to three iterations into the construction phase, to a much wider audience than just the stakeholder(s) we're directly working with.  We do this for several reasons:

1. **For stakeholders to gain confidence in the development team**. A demo/review with a wider audience can be a good way to show everyone that we’re doing great work, and that it in fact is possible to produce working software on a regular basis.
2. **To assess the ability of our product owner**.  Demos/reviews such as this provide your team with an opportunity to assess whether your product owner truly does represent the overall stakeholder community. If the feedback from the review is negative, or at least widely contradictory, that's a sign that product owner doesn't really understand the needs of the overall stakeholder community. This may be because they did an inadequate job of requirements envisioning, because they didn't facilitate reasonable stakeholder consensus during project inception, or because they have their own agenda that conflicts with general stakeholder consensus.
3. **To get feedback**. Your product owner could be doing a great job, but it's still possible that one or more of your stakeholders have some new ideas that your product owner hasn't thought of yet.

Note that the term "all-hands" is more of a target than a reality in many cases, particularly in situations with many stakeholders and/or distributed stakeholders. As you saw in Figure 17, which summarizes results from the “2010 How Agile Are You?” survey, 42% of respondents claiming to be on agile teams indicated that they do "all-hands" demos.

### 4.4.3 Light-Weight Milestone Reviews

Milestone reviews, particularly lightweight ones, are a possible option, particularly for disciplined agile teams. The Dr. Dobb’s Journal’s 2008 project success survey found that agile teams have a 70% success rate. Because this rate isn’t 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 upon completing an iteration, the business environment may have changed and negated the potential business value of the system.

## 4.5 Short Feedback Cycles

An important agile quality strategy, an important quality strategy in general, is to reduce the feedback cycle between obtaining information and validating it. The shorter the feedback cycle the less expensive (on average) it is to address any problems, see Why Agile Testing and Quality Techniques Work later in this article, and the greater the chance that you'll be motivated to make the required change in the first place. Table 1 discusses the impact of the feedback cycle of several agile techniques to give you a better understanding.

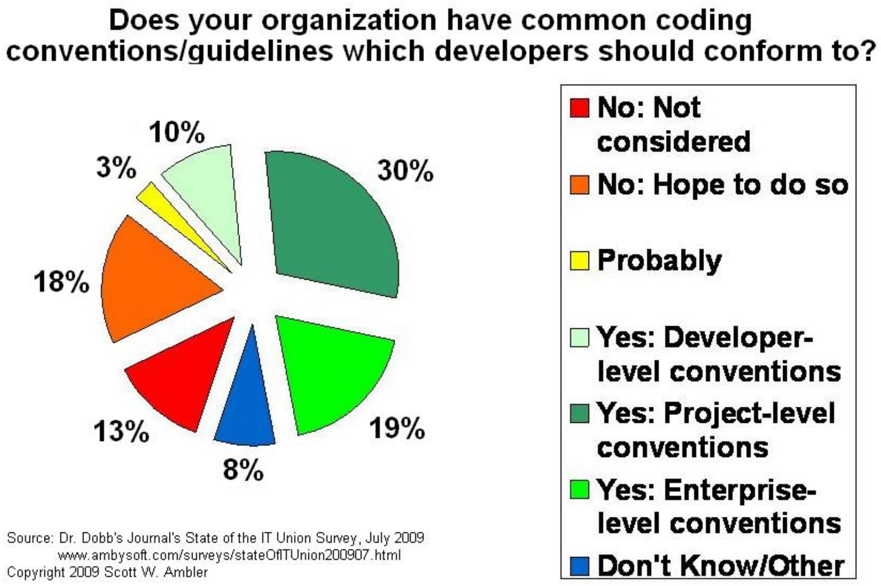
**Table 1. Feedback cycles for some agile strategies.**

| Agile Strategy | Feedback Cycle |
|---|---|
| Continuous Integration | **Minutes**. You check in your code, rerun your build (or have it automatically rerun for you depending on the CI tools that you're using), and from your test results (remember, agilists are at least doing developer regression testing if not TDD), you see whether or not what you just did works as expected. |
| Active Stakeholder Participation | **Seconds**. You discuss with one or more stakeholders what they want, getting feedback in real time regarding your understanding of what they're asking for.<br><br>**Hours to days**. Once you think you understand their intent, you work on developing a solution that fulfills that intent. This will take you a few hours or maybe a few days, after which you show them what you did and get feedback from the stakeholder(s). You iterate as needed. |
| Non-Solo Development | **Seconds**. You do some work as others are looking on and providing input. |
| Test-Driven Development (TDD) | **Minutes**. You a test, write just enough production code to fulfill that test, run your build (see continuous integration), and within minutes you know whether what you just did works or not. |
| Iteration/Sprint Demo | **Weeks**. Your team promises to deliver something at the beginning of the iteration/sprint, and at the end of the iteration/sprint you demo what you built. Because iterations are typically measured in weeks, the feedback cycle provided by an iteration demo at the end of an iteration is measured on the order of weeks because it validates whether you fulfilled the promise(s) that you made at the beginning of the iteration. |

## 4.6 Standards and Guidelines

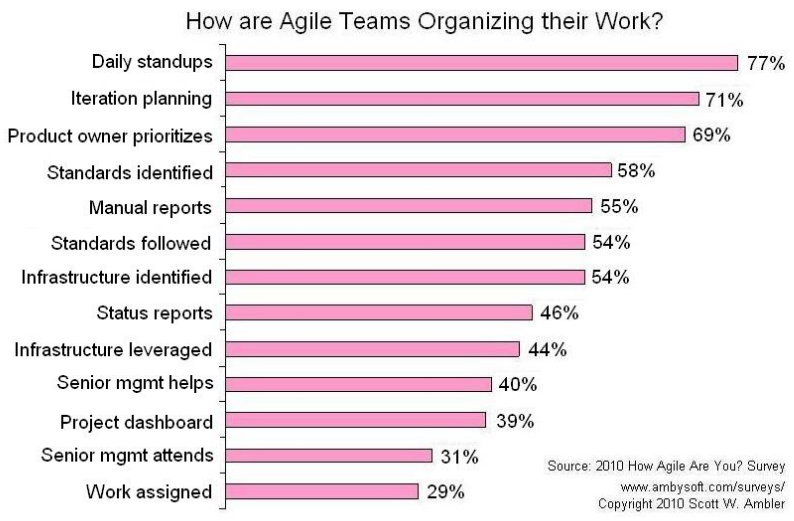
Following development standards and guidelines is an important, and relatively easy to adopt, quality technique. When IT developers follow a common set of development guidelines it leads to greater consistency within their work, which in turn improves the understandability of the work and thus its quality. The good news is that agile methodologies recognize the importance of development guidelines: Extreme Programming (XP) includes the practice Coding Standards, Agile Modeling includes the practice Apply Modeling Standards (see Elements of UML 2.0 Style for examples), and Agile Data promotes the practice of following database guidelines.  The bad news, as you can see in Figure 18 which summarizes results from the DDJ State of the IT Union July 2009 survey , is that there appears to be more talk about following guidelines than there is actual following of said guidelines. Following personal guidelines is a bit better than following no guidelines at all, hopefully, and following project-level guidelines is better still. Ideally, development teams should be following enterprise-level guidelines to promote ease of transfer between teams and reduction of effort around developing guidelines. Better yet, you should strive to adopt and tailor existing industry guidelines for your organization -- does it really make sense for you to create your own set of Java Coding Guidelines? Probably not.

**Figure 18. Development team's approach to following coding conventions (not paradigm specific).**



The Agile Practices and Principles Survey (July 2008) found that when agile development teams were following common development guidelines that they were most likely to be following coding guidelines, and less likely to be following either database guidelines or user interface (UI) guidelines. Figure 19, which summarizes results from the "2010 How Agile Are You?" survey, found that people who believed they were on agile teams reports that 58% of them had development guidelines identified and that 54% (93% of the 58%) were actually following them.

**Figure 19. Agile criterion: Self-organization.**



## 4.7 Implications for Quality Practitioners

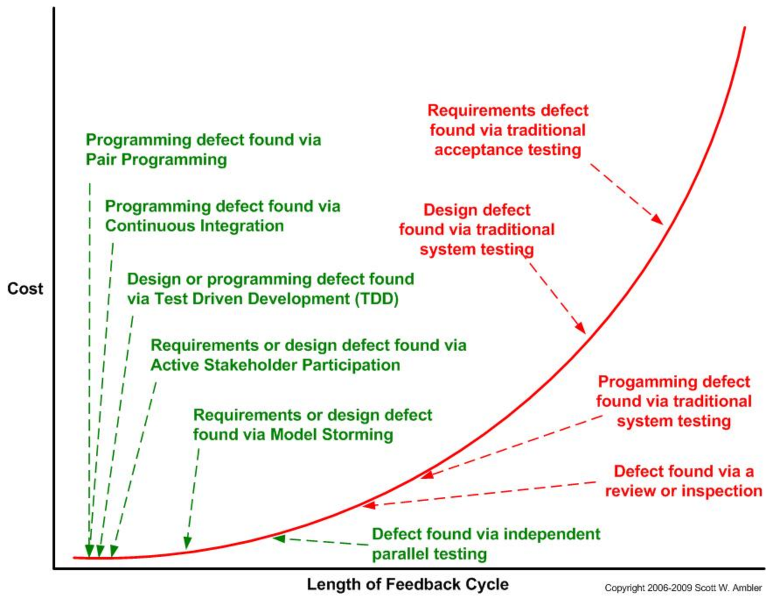
There are several very serious implications for quality practitioners:

1. **Become generalizing specialists**. You need to have a wide range of skills; some of them you may already have but some of them will be new.
2. **Be flexible**. Agile teams work in an evolutionary (iterative and incremental) manner where the detailed requirements and design emerges over time. This requires greater flexibility, as well as greater discipline, than what is typically the case in traditional/waterfall environments (see The Discipline of Agile for more thoughts on this)
3. **Rethink your approach**. Agile teams work in a more collaborative and open manner that reduces the need for documentation-heavy, bureaucratic approaches. The good news is that they have a greater focus on quality-oriented, disciplined, and value-adding techniques than traditionalists do. However, the challenge is that the separation of work isn't there any more – everyone on the team is responsible for quality, not just "quality practitioners". This requires you to be willing to work closely with other IT professionals, to transfer your skills and knowledge to them and to pick up new skills and knowledge from them.

## 5. Why Agile Testing and Quality Techniques Work

Figure 20 depicts several common agile strategies mapped to Barry Boehm's Cost of Change Curve. In the early 1980s Boehm discovered that the average cost to address a defect rises exponentially the longer it takes you to find it. In other words, if you inject a defect into your system and then find it a few minutes later and fix it, the cost is very likely negligible. However, if you find it three months later, that defect could cost you hundreds if not thousands of dollars to address on average because, not only will you need to fix the original problem, you'll also have to fix any work which is based on that defect. Many of the agile techniques have feedback cycles on the order of minutes or days, whereas many traditional techniques have feedback cycles on the order of weeks and often months. So, even though traditional strategies can be effective at finding defects the average cost of fixing them is much higher.

**Figure 20. Mapping common techniques to the cost of change curve.**


Cost
Length of Feedback Cycle
Programming defect found via Pair Programming
Programming defect found via Continuous Integration
Design or programming defect found via Test Driven Development (TDD)
Requirements or design defect found via Active Stakeholder Participation
Requirements or design defect found via Model Storming
Defect found via independent parallel testing
Requirements defect found via traditional acceptance testing
Design defect found via traditional system testing
Progamming defect found via traditional system testing
Defect found via a review or inspection
Copyright 2006-2009 Scott W. Ambler