White Paper

Improved Software Quality with Agile Processes

Author: Tabinda Aftab

Research and Development

Application Services

www.ephlux.com

Ephlux

Introduction

The fundamental nature of software development is changing, and quality professionals must change with it. With the acceptance of this motion, modern software processes have changed to evolutionary, iterative, and incremental rather than the traditional models that are linear and highly inflexible towards changes.

In this whitepaper it is emphasized that although agile development model is built on “lighter” and ”faster” deliveries they still lead to software that proves in practice to be of much higher quality than what traditional software teams usually deliver. It is common to say that Agilists are required to be quality conscious as the Agilist developers are the first to test their own codes. Therefore, it might result in changed and perhaps even smaller role for quality professionals on agile software development projects. More of this elaborated in the later section

Agile methodology have gained tremendous acceptance in the commercial arena since late 90s mainly because it deals with unstable and volatile requirements by using a number of techniques of which most notable are:

1) Simple planning,
2) Short iteration,
3) Earlier release, and
4) Frequent customer feedback.

These characteristics enable agile methods to deliver product releases in a much short period of time compared to the traditional approach.

Quality is an inherent aspect of true agile software development. In an agile world, the developers are required to code in a manner that the requirement of quality assurance is lessen. Therefore, it can be easily said that an appropriately designed code, self tested by the developer himself before the customer will lead to a better quality of the system as the development will be itself ”quality conscious” enough to discover and mend any loopholes or loose ends.

Quality Assurance Process in Agile Framework

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

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

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

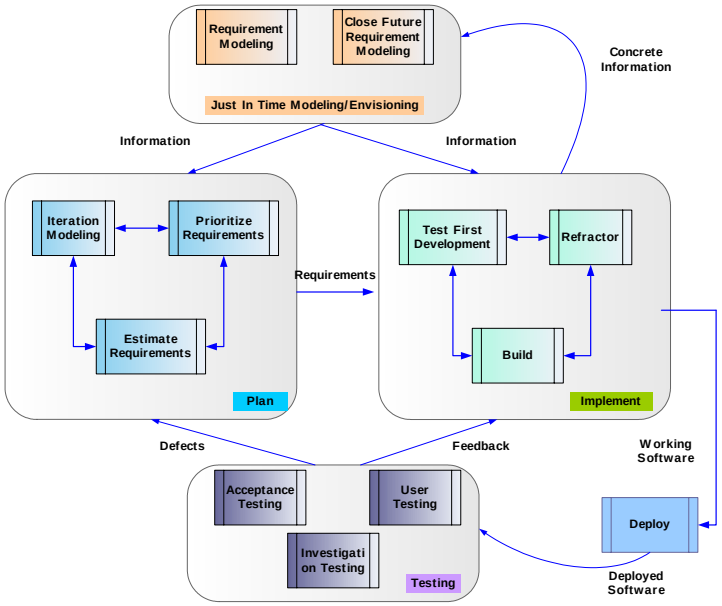

Figure1: shows the overall process involving development and QA tasks.

Refactoring

Refactoring is a disciplined way to make small changes to source code to improve its design and making it easier to work with by removing redundancy, eliminating unused functionality, and rejuvenating obsolete designs. A critical aspect of refactoring is that it retains the behavioral semantics of the code. The developers neither add nor remove any functionality or logic when they refactor, they merely improve its quality.

A database refactoring is a simple adjustment to the database diagram that improves its design while retaining both its behavioral and informational semantics.

Refactoring neither fixes bugs nor adds new functionality. Rather it improves the understandability of the code or changes its internal arrangement and design, and removes dead code, to make it easier for human maintenance in the future. Refactoring throughout the entire project life cycle saves time and increases quality.

Refactoring is safer in Test Driven Development (TDD) because you always have the test required for validating the component, independently of its internal implementation. A good time for refactoring is after a test pass state, thus you can refactor and verify that the test is still passing.

Test Driven Development

DD also known as test-first programming is an evolutionary (iterative and incremental) approach to programming where agile software developers must first write a test that fails before they write new functional code. The steps of TDD, as shown in Figure 3, are:

1. Add a test, basically just enough code to test the required functionality.
2. Run the tests, the complete test suite or only a subset to ensure that the new test does in fact fail.

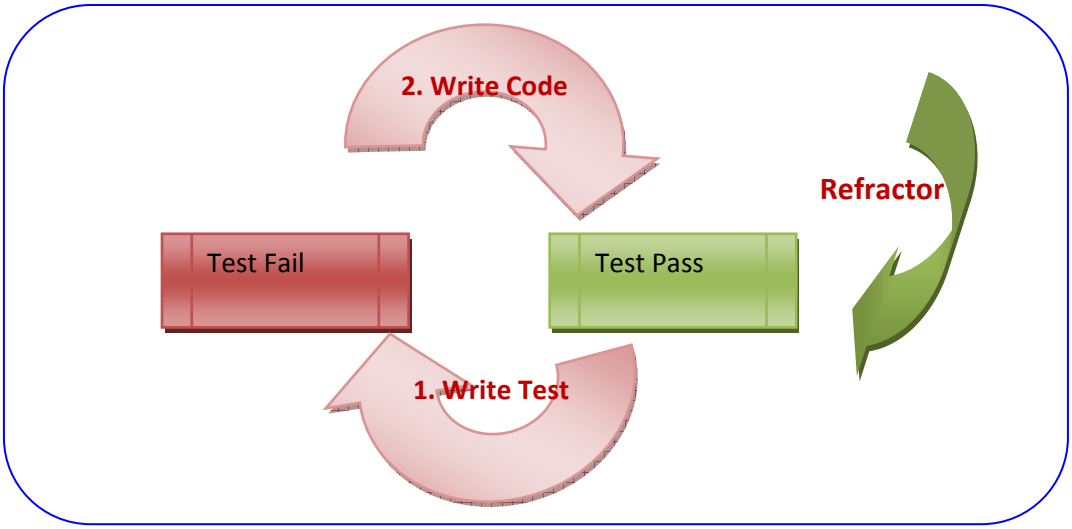

Figure 2: Shows Refactoring with Test Driven Development (TDD)

3. Make the required changes to update the functional code so it passes the new test.

- ✓ It gives them the courage to refactor their code to keep it at the highest quality possible, because they know there is a test suite in place that will detect if they have "broken" anything as the result of refactoring.

4. Run the tests again.
5. If the tests fail return to step 3.
6. Once the tests pass the next step is to move forward. At the end of this code refactoring is done.

a side effect of ensuring that your source code is thoroughly unit tested.

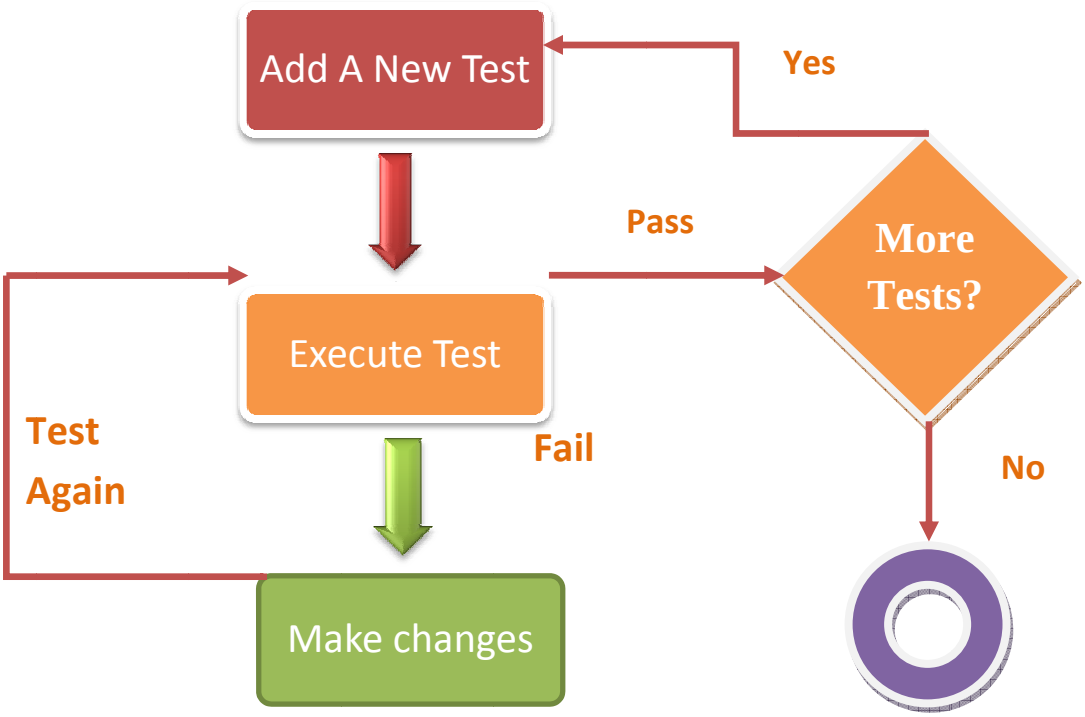

Figure 3: Typical TDD Flow

There are several advantages of TDD for agile software development:

- ✓ TDD forces developers to do detailed design just in time (JIT) before writing the code.
- ✓ It ensures that agile developers have testing code available to validate their work, ensuring that they test as often and early as possible.

As with conventional testing, the greater the risk profile of the system the more systematic your tests need to be. With both conventional testing and TDD you aren't striving for perfection, instead you are testing to the importance of the system you should "test with a purpose" and know why you are testing something and to what level it needs to be tested. An interesting side effect of TDD is that you achieve 100% coverage test – every single line of code is tested – something that traditional testing doesn't guarantee (although it does recommend it). In general I think it's

rather safe to say that although TDD is a specification technique, a valuable side effect is that it results in considerably better code testing than do traditional techniques.

Tests Replace Traditional Artifacts

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

Agile Model Driven Development (AMDD)

With AMDD you do a little bit of modeling and then a lot of coding, iterating back when you need to. Your design efforts are now spread out between your modeling and coding activities, with the majority of design being done as part of your implementation efforts – in many ways this was also true for many traditional projects, the developers would often do significantly different things than what was in the design models, but the designers would often blame the developers instead of question their overly serial processes.

AMDD is different from techniques such as Feature Driven Development (FDD) or the use case driven development (UCDD) styles of EUP and Agile Unified Process (AUP) in that it doesn't specify the type of model(s) to create. FDD insists that features are your primary requirements artifact whereas UCDD insists that use cases are. AMDD works well with both an FDD or a UCDD approach because the messages are similar – all three approaches are saying that it's a good idea to model before you code.

AMDD works for several reasons:

- ✓ **You can still meet your "project planning needs"**. By recognizing the high-level requirements early, and by recognizing a potential architecture early, you have enough information to produce an initial cost estimate and schedule.
- ✓ **You handle technical risk**. Your preliminary architecture modeling efforts enable you to identify the major areas of technical risk early in the project without taking on the risk of over modeling your system. It's a practical "middle of the road" approach to architectural modeling.
- ✓ **You minimize wastage**. A JIT approach to modeling allows you to focus on just the aspects of the system that you're actually going to build. With a serial approach, you frequently model aspects of the system which nobody actually wants.
- ✓ **You ask better questions**. The longer you wait to model storm a requirement, the more knowledge you'll have relating to the domain and therefore you'll be able to ask more intellectual questions.
- ✓ **Stakeholders give better answers**. Similarly, your stakeholders will have a improved understanding of the system that you're building because you'll have delivered working software

on a usual basis and thereby provided them with concrete feedback

Practices to ensure Agile Quality Assurance
Agile methods include many practices that have the potential ability for quality assurance. By identifying these practices and comparing with quality techniques used in waterfall model, we can analyze the quality assurance status of agile methods. We list some agile practices, which have been recognized, as quality techniques below, and we believe there is a number of other techniques have not been explicitly identified yet.

Customer Centric

Close communication is a general practice in most of the agile methods. Customer helps developer to refine and correct the requirements. The customer should support the development team throughout the whole development process. There is no such activity in the traditional methods. Traditionally customers are only involved in requirement definition and possibly system and software design but not involved as much and contributes as much as they are expected in agile methods.

Pair Programming

Pair programming means two programmers continuously work on the same code. This could improve design quality and reduce defects. Its effect shows that pair programming includes code V&V techniques. Its shoulder-to-shoulder technique serves as a continual design and code review process, and result reducing defect rates. This action has been wildly recognized as continuous code inspection.

Continuous Integration

Continuous integration is also a popular practice among agile methods. Continuous integration means the team does not integrate the code once or twice. The team needs to keep the system fully integrated at all times. Integration may happen several times per day. The key point is that continuous integration catches enough bugs to be worth the cost. Continuous integration also reduces the time that people spend on searching bugs and allows detection of compatibility problems early. This practice can be treated as a code V&V technique and is an example of dynamic techniques.

Acceptance Testing

Acceptance testing is carried out after all unit test cases have passed. This activity is a dynamic quality assurance technique. Waterfall approach has acceptance testing but the difference between agile acceptance testing and traditional acceptance testing is that in agile model it happens much earlier and much more frequently and not only done once.

Tightly Integrated Development & QA Teams

Strive for closely integrated development and QA teams. Ideally, QA Engineers sit next to their developer counterparts and the testing effort is shared and occurs in lockstep with code development. As well, the automated testing infrastructure should be common.

Holistic Quality Decisions

Make holistic quality-related decisions. Involving Development and Product Management with decisions impacting testing resources allows for more effectual use of limited time. QA should focus on areas known to be of high risk, for example new features where unit testing is known to be lacking or the code base is likely to display buggy behavior based on intrinsic complexity. As well, the likely customer impact of a bug in a given vicinity (knowledge usually unique to PM) is precious in

terms of determining whether or not that feature deserves special attention.

Intelligent Tests

Strive for 100% automatic test coverage. This is a brash goal and oftentimes the reality is far from ideal. Develop good tests, whether manual or automated. A bad test (defined as either being redundant, poorly defined, trivial, or at worst misleading) is expensive in terms of maintenance and changed confidence levels.

Neatly Designed Code

Treat test code as production code. Follow good coding conventions, comment well, and refactor each test such that it remains appropriate. This is another example of a pinnacle of testing that is hard to reach.

Investigation Testing

Significant value can be gained by submitting your system to an independent test team at intervals throughout the lifecycle so that they can verify the quality of your work. Agile teams create working software at the end of each construction iteration; therefore, you have something fresh to test at that point.

The investigative test team's task should be to ask, "What could go wrong," and to discover potential scenarios that neither the development team nor business stakeholders may have thought. They're attempting to address the question, "Is this system any good?" and not, "Does this system fulfill the written specification?" The confirmatory testing efforts confirm whether the system perform the intent, so simply repeating that work isn't going to add much value.

Investigative testers illustrate potential problems in the form of defect stories—the agile equivalent of a defect report. A defect story is treated as a form of requirement—it is estimated and prioritized and put on your requirements stack. The need to fix a defect is a type of requirement, so it makes perfect sense to address it just like any other requirement.

Your investigative testing will address common issues such as load/stress testing, integration testing, and security testing. Scenario testing, against both the system itself and the supporting documentation, is also common. You may also do some form of usability testing—the user interface is the system to most end users; therefore, usability is critical to success. The UI includes both the screens that people interact with and the documentation that they read, implying that you need to test both.

It also provides feedback to management that the team is effectively delivering high-quality working software on a regular basis.

Implications For QA Teams

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

1. **Reduced Quality Assurance Activity**

Higher-quality code implies less need for quality assurance activities, such as reviews and inspections, elsewhere in the life cycle. Agilists are quality infected; they have accepted what the quality assurance community has been saying for decades, and they are actively developing high-quality system artifacts as a result.

2. **Expect Incompleteness Specifications**

Models, documents, and source code evolve over time. The only time when agilists can honestly say that something is done is when they are ready to release it into production. For example, they won't have a finalized requirements document from which to develop test cases until the very end of the project. The implication is that if QA team are going to be

involved with agile software projects then they need to get used to working with in-progress artifacts.

3. **Improve Your Skill set**

A critical concept is that agilists need to move away from being narrowly focused specialists to become more of generalizing specialists. A generalizing specialist is someone with one or more technical specialties who actively seeks to gain new skills in his or her existing specialties as well as in other areas. A person should have more than one specialty so he or she can add value to the team, but one needs a common understanding of the software process and of the business domain so he or she can interact successfully with others on the team. With evolutionary approaches to development there is no longer room to invest several hours, let alone days or weeks, on a given activity. One does a little bit of requirements elicitation, then some analysis and design, then some implementation.

Quality Assurance Comparison In Traditional & Agile Models

If we compare the difference between agile quality assurance activities and waterfall SQA from three aspects:

1. ***The SQA activities start early and the frequency is much greater than in waterfall model***.
   The agile process has many small releases and each release can be considered to be similar to a tiny waterfall release. Clearly analyzing how quality can be achieved in each agile release will help us understand how agile processes achieve quality. This will allow us to identify which parts of agile development add the most quality to our software.

2. ***Agile methods have fewer static quality assurance techniques when compared with waterfall development***.
   This can be explained by their background. Many of the static techniques in waterfall are people-intensive; these cost time and resource. Agile methods are used when we have market pressure and budget limitation so people-intensive techniques are not acceptable.

   The second reason is that waterfall normally begins with static requirement documents. Hence static techniques are suitable. Agile methods, on the other hand, begin with poor and violate requirements. This makes static methods unsuitable at this stage.

Conclusion

The emphasis of agile development model on faster, evolving requirement modeling and implementation not only leads to better quality implementation but also lessens the quality assurance team's involvement. The Agilist developers ensure their code's good design, quality and functionality through refactoring, TDD and other practices known through best practices of agile model.

In reality, the quality of software in true agile framework is even better than traditional framework as the quality assurance and testing and bug fixing activities are more frequent and start at a very early stage in the process as compared to the traditional framework.

Achieving a true agile SQA process flow is not easy, as it requires consistent coordination between all the stakeholders to ensure that the resulting product qualifies at aspect.

Acceptance testing, usability testing and investigation testing routines further test the product's quality from the point of view of external users.

The application of agile SQA is still a new topic and much research is required to fully implement it.

References

1. Ambler, S. W. 2003c. Model reviews: Best practice or process smell? http://www.agilemodeling.com/essays/modelReviews.htm.
2. Ming Huo, June Verner, Muhammad Ali Babar, Liming Zhu, How does agility ensure quality?
3. Continuous Integration,http://www.martinfowler.com/articles/continuousIntegration.html.
4. Ambler, S. W. Quality in an Agile World
5. "Refactoring Mercilessly" http://www.extremeprogramming.org
6. Ambler, S. W. 2006. "Agile Testing Strategies", Dr.Dobb'sPortal, http://wwwddj.com