5 Agile Development, Quality, and Testing Practices

YOU MIGHT ASK: "Why does a project manager need to know something about development practices?" In the past, the role of a project manager might have been somewhat limited to a coordination function to integrate the efforts of different functional organizations that played a role in the project. In many cases, the actual direction for the different functions involved (development, test, etc.) came from the managers who were responsible for those functions themselves, and the role of the project manager in providing direction may have been limited. An agile environment is different:

- Instead of a relatively loosely knit team of people from a variety of functional departments who might only work on a particular project on a part-time basis, an agile team is typically dedicated to a project and should be much more tightly integrated.
- The methodology for an agile project is much more of one well-integrated project methodology with all the functions (development, test, etc.) working more collaboratively and concurrently. Any decisions about how development and testing is done need to be integrated with the overall project management approach.

Those factors require a much higher level of cross-functional leadership. Agile project managers can help provide that leadership if they have the cross-functional knowledge that is required, but it is more than a simple coordination function.

AGILE SOFTWARE DEVELOPMENT PRACTICES

This section provides an overview of some of the most important agile software development practices that an agile project manager should be familiar with.

Code refactoring

In order to improve the reliability and maintainability of the software, code refactoring involves:

- Removing redundancy
- Eliminating unused functionality
- Rejuvenating obsolete designs
- Improving the design of existing software

Agile approaches tend to emphasize quickly creating code to meet functional requirements *first* and then refactoring the code as necessary *later* to clean it up. Refactoring throughout the entire project life cycle saves time and increases the quality of the software.

In a traditional project, there is typically an emphasis on laying out the entire design and architecture of the project early in the project to avoid unnecessary rework later. In an agile project, that approach may not be very realistic, for a couple of reasons:

1. It's very difficult in an agile approach to define all the requirements in a sufficient level of detail at the beginning of the project to lay out the entire design up front.
2. An agile approach encourages changes as the project progresses in order to better adapt to user needs and requirements and attempting to define and control the design approach early in the project may restrict that ability to make changes.

The question then becomes—how do you avoid having code that is a complete mess and is unreliable and difficult to maintain as a result of so many unplanned changes? Fortunately, modern development tools have made it easier to implement code refactoring as the project progresses to more easily rework and clean up the code to make it more maintainable and reliable in an agile environment. The use of design patterns and coding standards can also make this approach more practical to implement.

The truth is that code refactoring is just as important in a traditional project, but it often hasn't been done adequately because it was assumed that the design of the code was defined and stabilized up front, and that just isn't a very realistic assumption in many cases, even in a non–agile environment. The strengths and benefits of a code refactoring approach are that it:

Here are the strengths and benefits of code refactoring:

- Encourages developers to put the primary focus on the functionality provided by the code first and clean it up later to make the code more well-structured and maintainable.
- Reduces the time required to produce functional code that can be available for prototyping and user validation.

There are some risks and limitations associated with code refactoring:

- The amount of rework of the code required might be significant.
- The structure of the code might not be optimized around the most desirable architectural approach.

- Time pressures to complete the iteration may short-circuit the code-refactoring effort and allow poorly organized code to be released.

Continuous integration

Continuous integration is the practice of frequently integrating new or changed software with the code repository and performing overall system integration testing throughout the project rather than deferring that effort until the end of the project. Continuous integration provides a way of early detection of problems that may occur when individual software developers are working on code changes that might conflict with each other. In many typical software development environments, integration might not be performed until the application is ready for final release.

I was involved in a large software development project in the early 1990s where a major electronics company invested over $150 million in the development of a very complex hardware/software switching system and the project had to be abandoned in the end because when it was time to integrate the project, the various components of the system could not be integrated and the overall system would not work reliably. That is a perfect example of the critical importance of continuous integration.

There are some strengths and benefits of continuous integration:

- Developers detect and fix integration problems continuously.
- Early warning of broken/incompatible code and of conflicting changes.
- Constant availability of a "current" build for testing, demo, or release purposes.
- Immediate feedback to developers on the quality, functionality, or system-wide impact of code they are writing.
- Frequent code check-in pushes developers to create modular, less-complex code.
- Metrics generated from automated testing and continuous integration focus developers on developing functional, quality code, and help develop momentum in a team.

There are also some risks and limitations associated with continuous integration:

- It might be difficult to implement on larger code development projects where the integration effort may be too complex to do as frequently.
- It requires a level of sophistication and close teamwork on the part of the team to make it work especially on large, complex projects.
- Resources and tools to automate the continuous integration, building, and testing process are essential and can be expensive.

Pair programming

Pair programming is another agile development process that is sometimes used to improve the quality and reliability of software. Pair programming is analogous to the way a pilot and a copilot fly a

commercial airliner. In a commercial airliner, one of the pilots flies the airplane while the other pilot oversees the overall flying of the aircraft and provides support to the person who is actually doing the flying. Pair programming in a software development environment works in a similar fashion.

One developer typically writes the code and the other developer provides overall guidance and direction as well as support and possibly mentoring to the person writing the code. This technique might be used by two peer-level developers who rotate between the two roles, or it might be used by a senior-level developer to mentor a more junior-level developer. Pair programming is not as widely used as some other agile development practices because the economics of dedicating two programmers to work together on the same task can't always be justified. In some cases, code reviews are substituted for pair programming to achieve a similar effect.

Here are the strengths and benefits of pair programming:

- Design quality:
  - One developer observing the other person's work should result in better quality software with better designs and fewer bugs.
  - Any defects should also be caught much earlier in the development process as the code is being developed.
  - The cost of a second developer that is required may or may not be at least partially offset by productivity gains.
- Learning and training. Sharing knowledge about the system as the development progresses increases learning.
- Overcoming difficult problems. Pairs are able to more easily resolve difficult problems.

There are also some risks and limitations associated with pair programming:

- Work Preference
  - Some developers prefer to work alone.
  - A less-experienced, less-confident developer may feel intimidated when pairing with a more experienced developer and might participate less as a result.
  - Experienced developers may find it tedious to tutor a less-experienced developer.
- Costs
  - The productivity gains may not offset the additional costs of adding a second developer.

Test-driven development

Test-driven development is a somewhat widely used agile development practice. It is typically used in conjunction with unit testing by developers. Rather than writing some code first and then, as a second step, writing tests to validate the code, the developer actually starts by writing a test that the code

needs to pass to demonstrate that it does the functionality that was intended and then writes code to make that test succeed. Of course, the test fails initially until the functionality has been implemented, and the objective is to do just enough coding to make the test pass. Development is done incrementally in very small steps—one test and a small bit of corresponding functional code at a time.

Here are the strengths and benefits of test-driven development:

- It encourages the development of small, incremental modules of code that can be easily tested and integrated with a continuous integration process
- It is well suited to testing of the design as it progresses and provides immediate feedback to the developers on how well the design meets the requirements
- It also encourages developers to write only the minimal amount of code necessary to pass a given test

There are also risks and limitations associated with test-driven development:

- Test-driven development primarily addresses only unit testing of code modules—much more testing is typically needed at different levels of the application.
- Test-driven development emphasizes rapidly implementing software to provide a minimum level of required functionality and relies on later refactoring the code to clean it up to meet acceptable design standards. If that refactoring isn't done, the code might not be reliable and supportable.

Extreme programming (XP)

At one time, extreme programming (commonly referred to as XP) was a more important agile methodology, but XP is primarily just a collection of agile development practices consisting of:

- User stories
- Release planning
- Iteration planning
- Test-driven development
- Collective ownership of code
- Pair programming
- Continuous integration
- Ongoing process improvement

It doesn't provide a management framework for how an agile project should be managed, as Scrum does; it is more of a development process. Scrum does provide that management framework and has become much more dominant and more widely used agile process. Sometimes XP might be

used to define the development process used with Scrum, but more often, people take more of an à la carte approach to selecting the development processes that they use with Scrum.

AGILE QUALITY MANAGEMENT PRACTICES

For the same reason that cross-functional agile project managers need to understand development practices, they also need to understand quality management practices.

Key differences in agile quality management practices

Table 5.1 shows some of the key differences between the agile approach to quality management and the approach typically used for quality management in more traditional projects.

Definition of "done"

A very important concept in agile is the definition of "done." In agile, the goal of each sprint is to produce a "potentially shippable" product. Of course, *potentially shippable* can mean different things to different people, depending on the nature of the project. In a large, enterprise-level project, it might be impossible or impractical to really release something to production at the end of each sprint, for a lot of reasons. For example, the results might not be sufficiently complete to deliver a complete subset of functionality that is required, or additional integration work might be needed to integrate the software with other related applications before it is fully released to production.

TABLE 5.1 Key Differences between Agile and Traditional Quality Management

	Typical Traditional Plan-Driven/ Waterfall Quality Management Process	Typical Agile Quality Management Process
Integration of Testing with Development	Testing is typically done sequentially with development and is done by a separate organization.	Testing is done more concurrently with development and is well-integrated into the development team.
Testing Approach	There is a more reactive approach to finding and correcting defects.	There is a more proactive approach to preventing defects.
Responsibility for Quality	Responsibility for the quality of the software is perceived as on the QA organization.	The overall team has responsibility for the quality of the software.
Regression Testing	Because testing is deferred until the end of the project and code has stabilized, there may not be a need for a complicated regression testing approach.	Because testing is done concurrently with development and the code is constantly changing, it is more essential to do regression testing as the project progresses.

The important thing is that the team should have a clearly defined definition of what "done" means so that it is unambiguous and well understood by everyone on the team. One of the biggest difficulties on a software project is that there could be different interpretations of what "done" means.

- Is it when the coding is complete?
- Is it when the coding and testing are complete?
- Does it depend on user acceptance?

The more clearly and crisply this is defined removes a lot of subjectivity of making an assessment of whether something is "done" or not and makes it clear what the team is committing to when it makes a commitment to complete some number of stories in a sprint.

The definition of "done" typically includes:

- Code review
- Tested by developers and QA
- Accepted by the product owner and other stakeholders if necessary

The important thing is that the team takes complete responsibility for the quality of the software it develops and defines its own standards of completeness—quality isn't someone else's responsibility.

The role of QA testing in an agile project

That leads right into the question of: "What is the role of QA in an agile project?" As with many things in an agile environment, there is not necessarily a single right/wrong way of doing things, and there are different schools of thought on this:

- There's a somewhat idealistic view that everyone on the team should be capable of doing anything required (development or testing), and all developers should be totally responsible for the quality of the work that they produce rather than relying on someone else on the team to test it and provide feedback.
- In actual practice, that view is very difficult to implement, and there is value in having people with focused skills within a project team. There is also value in having an independent tester look at software other than the person who developed it.

Some people would say that having people focused on specific tasks and skills within a team inhibits teamwork. I don't agree with that point of view. The analogy I like to use is of an athletic team—on a high-performance athletic team you have people who are highly trained and skilled around playing particular positions on the team, but that doesn't inhibit them working together as a cohesive team.

A good team should be cross-functional and collaborative, and having people on the team representing different perspectives will ultimately strengthen the team. QA is a discipline that needs focus

and training to do it well. To give it to a developer as just another thing for them to do might not be the best use of resources, and there's always value in having an independent observer look at someone else's software. A trained and specialized QA tester will typically see things that the primary developer might have taken for granted or overlooked.

The important points are that:

- The team, as a whole, owns responsibility for quality—*it is not someone else's responsibility*.
- QA should be an integral part of the team, not a separate organization external to the team.
- QA testing is an important skill, and it is often worthwhile to have people on the team who are specialized and skilled in QA testing who are dedicated to planning and executing QA tests.

AGILE TESTING PRACTICES

Concurrent testing

The biggest difference in agile testing practices is that testing is integrated with development in each sprint in a very collaborative process where testers and developers take joint responsibility for the quality of the product that they produce. In actual practice, there are a number of ways that can be done. For example, instead of waiting until the end of the sprint to turn over all stories to QA for testing, a better practice would be to have the testers begin testing the software as soon as it is sufficiently complete for testing. That approach allows more concurrency of development and testing and avoids any last-minute surprises at the end of the sprint.

Acceptance test driven development

Acceptance test-driven development is an excellent approach that is used in agile projects. It is very similar to test-driven development but it is at a higher level of functionality. Test-driven development is done at the level of unit testing to validate the implementation of code, while acceptance test-driven development is done at a higher level of functional testing and tests the features and behaviors of the system that are observable by the user.

Acceptance test-driven development involves writing the acceptance tests for the functionality that must be provided in each story prior to starting development. This is usually done as part of the writing or grooming of the stories, and it helps to build a more concise, common understanding of exactly what needs to be done to satisfy the user need for each story.

A big advantage of acceptance test-driven development is that it can eliminate the requirement for writing detailed functional specifications, it simplifies the requirements definition process, and it keeps everyone on the team clearly focused on the functionality that must be provided to satisfy the business need.

Repeatable tests and automated regression testing

Because testing is done concurrently with development in an agile project and new functionality is being continuously added, it is essential for tests to be repeatable in an agile project to ensure that new functionality or other development activity has not broken something that was already tested and completed. As any item of new functionality is added, a regression test is needed to repeat the testing of any items of functionality that were previously tested to ensure that nothing has been inadvertently broken by adding the new functionality.

As agile projects get larger, it becomes impractical and perhaps even impossible to repeatedly manually run a very large suite of regression tests that could be in the hundreds or thousands of tests so there is a big advantage to automating regression tests so that they can be run repeatedly and efficiently on a frequent basis. In an ideal case, automated regression tests can be run nightly to check the entire system as new functionality is being added. It would be very difficult, if not impossible, to do that frequently without automation for a complex system with a large number of individual regression tests to be run. And, of course, automating the regression testing frees up QA test resources to focus on more value-added tasks and also ensures that it is done consistently each time it is run.

Value-driven and risk-based testing

Another consideration in planning the testing strategy in any project is to recognize that it is impossible to test 100 percent of everything you could possibly test, and some method should be developed to determine what testing should be done and what testing can be minimized or eliminated. Fortunately, an agile project makes that easier to do because there is a lot more direct communications with the users to determine what is important and what is not important, and the user is directly engaged in testing the functionality of the software as it is being developed.

SUMMARY OF KEY POINTS

It's important for an agile project manager to have a broad, cross-functional view of all aspects of a project including development and testing practices to allow him/her to ensure that the people, process, and tools in the project are very well-integrated to maximize the overall efficiency and flow of the project and that they are also well-aligned with achieving the business goals the project is intended to accomplish.

Agile Software Development Practices

1. Continuous Integration and Code Refactoring

   An agile project calls for a different approach for managing the development process that should be well-integrated and well-aligned with the overall agile project management approach. It is not often possible to completely lay out and stabilize the design of a project early in

the project. A more dynamic approach for evolving the design as the project progresses is needed. Because the design is potentially changing throughout the project, a rigorous approach is needed to manage how changes are merged into the design, emphasizing continuous integration and code refactoring to ensure that the overall reliability of the design is still maintained.

2. Pair Programming

   Pair programming is sometimes used in agile projects to coach and mentor less-experienced developers and to provide higher levels of code quality and reliability. However, it is difficult to justify the economics of pair programming on all agile projects, and sometimes a more limited form of code reviews can accomplish some of the same goals.

3. Test-Driven Development

   Test-driven development is often used on agile project to simplify the development effort and to improve the reliability and quality of the software.

4. Extreme Programming

   Sometimes XP might be used to define the development process used with Scrum, but more often, people take more of an à la carte approach to selecting the development processes that they use with Scrum.

Agile Quality Management Practices

1. Key Differences in Agile Quality Management Practices

   The philosophy and approach for managing quality in an agile project is very different from a traditional project. The most important differences are that testing is an integral part of the development effort and is done concurrently with the development effort. The team, as a whole, needs to own responsibility for quality—it is not someone else's responsibility. The orientation is also more proactive towards preventing defects rather than a typical reactive approach of finding defects after the fact.

2. Definition of "Done"

   The definition of "done" is very important in an agile project. It provides a very well-defined standard of completeness that the team commits to. If it is properly done, it removes a lot of subjective judgment of whether a development effort is really complete or not.

3. The Role of QA Testing

   QA Testing plays a different role in an agile project. QA should be an integral part of the team and not a separate organization external to the team; however, that doesn't necessarily mean that developers should do their own testing. QA testing is an important skill and it is many-times worthwhile to have people on the team who are specialized and skilled in QA testing who are dedicated to planning and executing QA tests.

Agile Testing Practices

1. Concurrent Testing

   Instead of waiting until the end of the sprint to turn over all stories to QA for testing, a better practice would be to have the testers begin testing the software as soon as it is sufficiently complete for testing. That approach allows more concurrency of development and testing and avoids any last-minute surprises at the end of the sprint.

2. Acceptance Test-Driven Development

   Acceptance test-driven development is very similar to test-driven development but it is at a higher level of functionality. A big advantage of acceptance test-driven development is that it can eliminate the requirement for writing detailed functional specifications, it simplifies the requirements definition process, and it keeps everyone on the team clearly focused on the functionality that must be provided to satisfy the business need.

3. Repeatable Tests and Automated Regression Testing

   Because testing is done concurrently with development in an agile project and new functionality is being continuously added, it is essential for tests to be repeatable in an agile project to ensure that new functionality or other development activity has not broken something that was already tested and completed. As agile projects get larger, there is a big advantage to automating regression tests so that they can be run repeatedly and efficiently on a frequent basis. In an ideal case, automated regression tests can be run nightly to check the entire system as new functionality is being added.

4. Value-driven and Risk-based Testing

   It is impossible to test everything. Fortunately, an agile project makes that easier to determine what is important, and the user is directly engaged in testing the functionality of the software as it is being developed.

DISCUSSION TOPICS

Agile Software Development Practices

1. Why is it important for a project manager to have some understanding of development practices in an agile project?
2. What is continuous integration, and why is it important in an agile project?
3. What is test driven-development, and how is it different from acceptance test-driven development?

Agile Quality Management Practices

4. What is the biggest advantage of an agile quality management approach over a conventional non–agile testing approach? How is it different? What do you think it would take to make it work effectively?
5. What does the definition of "done" mean, and what is its importance in an agile project? Provide an example of a definition of done.
6. What is the role of a QA tester in an agile project? How is it different from a conventional, non–agile project?

Agile Testing Practices

7. Why is it important that tests be repeatable in an agile project?
8. Explain what automated regression testing is and why it is important in an agile project.