More so than other methodologies, agile depends on testing not only for verification and validation, but also for design and development.

Managing Test

A companion white paper to Agile Project Management, Making it Work in the Enterprise, Second Edition

John Goodpasture, PMP

Managing Test

Agile methods rigorously test to prove quality is designed-in.

All truths are easy to understand once they are discovered; the point is to discover them.

Galileo

To discover errors, demonstrate quality, and prove compliance—these are the reasons to test. It seems like an obvious statement, but the point of agile methods is to deliver working product at every iteration.

Principles and Practices Guide Testing-in Quality

Customer satisfaction is the ultimate goal of agile product development. For the agile practitioner quality is not a property to be controlled or assured by outsiders. Quality is a daily goal of every agile team member—an achievement to be demonstrated every day by automated tests that prove red failures can become green passes.

Quality has many dimensions, and many of those dimensions can be verified and validated by testing. Testing—to include user validation—is valued by agile methodologists as the one sure way to close any gap between what was asked for and what is to be delivered; testing is less to prove compliance to a specification and more to prove a satisfactory outcome for beneficiaries.

Principles and Practices

To that end, rules are needed that bind behavior but otherwise allow the self-direction of the development team to determine and modify day-to-day tactics.[1] Five principles to guide testing—in effect, the top-level rules—are summarized thus:

1. Don't kill the messenger
2. Every requirement must be testable; every test must relate to a requirement
3. Anticipate some failure and make room for correction and retest
4. Embrace learning
5. Avoid isolation

Test-driven Development Is the Starting Point

Test-driven development (TDD), is a practice that was proven first in XP.[2] The general TDD cycle is to:

"Write a test; . . . Make it run; . . . Make it right".[3]

- Write a test is a design task
- Make it run is an iterative design-development-test task
- Make it right is a regression test and refactor task

However, TDD is not restricted to the XP methodology, and as a concept, not even restricted to software.

Main Idea

Here are the main ideas of TDD, and they are a mind-bender for the traditionalist accustomed to having the test scripts come later in the project:

- Requirements are documented in the form of test scripts, and
- Test scripts are the beginning point for product (object, process, feature) design, and
- When the object's test passes, at least the minimum function, feature, and process requirements have been satisfied.
- Refactoring to coding and product standards may still be required

How TDD works

TDD works this way: detailed design and development, after consideration for architecture, is a matter of three significant steps:

Step 1: Document development requirements with test scripts. Beginning with functional requirements in the iteration backlog, and in collaboration with users, the developer writes technical design requirements in the form of a test script, and writes the script in a form to be run with test-automating tools. Automated tools enable quick turnaround. Automated tests run fast, run on-demand, run repeatedly the same way, and run under various data and system conditions.

Step 2: Run the test, modifying the object design until it passes. If the test fails, as it often will, the developer implements the quickest and simplest solution that will likely pass. Step 2 is repeated until the object's test passes. ("object" is a surrogate for whatever is being tested)

Step 3: Refine the detail design of the object. After the object passes the test, the developer iterates the detail design to be compliant with quality standards. In doing so, the internal detail is upgraded and improved. Usually, there is minimum change to the object's external characteristics. To verify continued functionality, the modified object is retested.

	TDD, unit tests, and acceptance tests
Project management tip	TDD was invented as a design practice; it's commonly applied to the lowest level design units "TDD's origins were a desire to get strong automatic regression testing that supported evolutionary design"[4] Unit tests, as distinct from TDD tests, are post-implementation design verification tests. In most respects, unit tests and TDD tests are very similar, but the purposes are very different: one drives design and the other verifies implementation. The concept of automated tests is applicable to higher level tests, like product integration tests and user acceptance tests, but these are not for the purpose of driving design. At higher levels, designs that pass unit tests are being tested in a larger context and by independent testers, including end-users

Beginning with a test script is to begin with the end in mind, an idea very much like Steven Covey's second habit for success.[5] Beginning with a test is, for all intents, beginning with the end-state features and functions written in the form of a script. The developer must think about how the object will work when it is finished, and write that working description into the script. To be as accurate as possible, the user must be present to converse with the developer during the course of the scripting. In fact, the dialogue begins before scripting with discussion of user stories in the iteration backlog.

User stories are vignettes of functional needs as told by expert users.[6] User stories themselves are not specific or technical enough to be actionable by developers. Thus, as seen in Step 1 just described there is interaction between developer and user to synthesize design requirements from user stories.

The Step 1-2-3 sequence is tagged as the *Red-Green-Refactor* sequence because the first test results are often *red* meaning failure, then *green* meaning a success, and finally *refactor* meaning the design is refined and brought compliant with quality standards. Refactoring usually directed at the internal detail of the object; the object's external appearance, functionality, and performance at its interfaces are often unchanged by internal refactoring.[7]

Of course, at Step 1, the test may pass; passing is a good thing when it happens. Depending on circumstances, if the first test passes before any design is done, then a reasonable conclusion is that the system already satisfies the requirement and additional design is not needed. A first-test success avoids redundancy and avoids adding unnecessary complexity to the product base.

	TDD works best with new products
Project management tip	Legacy complexities can greatly complicate a new project, making testing very time consuming and complex in order to prove, regressively, that no harm is done with new functionality. Legacy testing scope often dilutes much of the advantages of practices like TDD if automated tests are impractical. More conventional requirements documentation may be a better approach if it proves too complex to write the requirements as a test when modifying a legacy system Legacy tools and product base may not support the tools and frameworks for efficient TDD practices that depend on automated tests

Advantages of TDD

TDD addresses six big problems in achieving quality outcomes:[8]

1. Scope creep: Agile methodologists value simplicity.[9] In this sense, simplicity does not mean the absence of complexity; it means the absence of unnecessary design or unnecessary scope. Avoid design anticipation—so- called hooks—for requirements that ultimately never materialize. Avoid unnecessary redundancy and do not duplicate preexisting capabilities
2. Coupling and cohesion: Coupling and cohesion are outcomes of structured design, and are largely credited to the work of Larry Constantine, a distinguished researcher at MIT and IBM.

Constantine formulated the metrics of coupling and cohesion in the late 1960s, presenting his work in 1968 to the National Symposium on Modular Programming.[12] Coupling and cohesion are properties of architecture. System coupling—loose or tight—refers to interdependencies among components—the degree to which a change in state, function, or feature in one module affects the functioning or performance of another. Cohesion—low or high—is about consistency and similarity that promotes affinity. Systems with cohesion have common and consistent properties between modules that enable things to stay together and work together smoothly. Affinity could be procedural, temporal, logical, sequential, etc.

Testing readily identifies these properties. The fact is, it is much easier to write a test and get it to pass when the coupling is loose. A sure sign of tight coupling is that it is hard to write a test that passes!

3. Trust: Design and implementations that work are trusted by the team; so also is the designer or implementer. Trust in product development reduces nonvalue inspections and drives integrity in the product base. Testing establishes trust. The principle is: *no addition to the product base is made until the test passes.* Constant integration, frequent system builds, and rigorous regression testing guards the testing quality. Feedback corrects ineffective protocols.
4. Rhythm: Red-Green-Refactor sets up a pattern of activity with a certain cadence that maintains a pace and productivity, and most importantly, guards against wandering off the track. Rhythm promotes progressive successes.
5. Requirements management: Test scripts document technical requirements derived from user stories and other business documents that envision project outcomes. At the lowest level, the script is the persistent record of the requirement, logging information such as who created, accessed, updated, or deleted the script requirement, as well as the dates of each transaction. But there is more to managing requirements than just keeping track. As already described, tests reveal unnecessary and redundant requirements, dependencies that can be eliminated or minimized, and features or functions that can be eliminated to reduce complexity and overhead. And, tests reveal inconsistent requirements that might disturb cohesion.
6. Regression assurance: One of the motivations for a test-oriented development is to improve the likelihood of successful integration of a new capability into the product base.

Testing with Asserts

Among the things that make TDD work is the testing concept called *assert.*

The Meaning of Assert
Assert as a testing concept was invented and described by C. A. R. Hoare, a British scientist, and described by him in a widely read 1969 paper.[10]
The word is taken from the verb assert, meaning in effect, to make a claim, state a position, or insist on a reality. Assert is a condition imposed on the object under test.
An assert can be a precondition, a post-condition, or both.
As commonly implemented, an assert is a condition that returns a value of true or false, but other logic is also employed, such as null or not null, equals, and same or not same.[11]

If the object is working correctly, the condition should always be true at the point where the assert is placed.

The Red-Refactor-Green TDD cycle iteratively operates on a test object until it works according to the functional needs of the user stories.

Testing with scripted asserts involves a three-step protocol.

1. Assert a precondition
2. Operate and execute the object procedures, then check the post-condition assert
3. Assert a result or post-condition—if the logic returns the expected result, the test passes

Architecture Impacts TDD

Architecture is a prerequisite for TDD. On one level, it is a tool to organize and coordinate business requirements among the many product components, revealing, at a high level, redundancy, inconsistency, and problems of coupling and cohesion. On another level, architecture is the top-level specification of the structure, performance, appearance, relationships, and the interactivity among elements acting as a system.[13] It describes the external behaviors, the look and feel, and the connection dependencies of the elements. Architecture provides the overall road map to the finished product, although there is not always a green field for architecture. It may be constrained by a legacy product base, distribution system, or manufacturing and supply chain system.

Architecture, but not too much architecture

Albert Einstein once said, *"Everything should be made as simple as possible, but not simpler."* As applied to architecture, Einstein provided good advice.

As ***simple as possible*** means avoiding fortuitous complexity, unnecessary redundancy, and couplings that create unnecessary dependencies.

In agile speak, *as simple as possible* means designing for the present requirement and not second guessing where the customer may next find value.

The phrase ***but not simpler*** means do not decompose to such small units such as tiny user stories so that essential cohesion is missing and the overall understanding is lost in a forest of details.

Test Planning Is Essential to Good Test Metrics

British Field Marshal Sir Bernard L. Montgomery, a military leader of World War II, once famously told his staff when planning for the invasion of Europe, "I don't read papers," referring to the heavy weight of the memoranda his planners were trying to foist on him.[14] Perhaps Montgomery was the lean thinker of his day! His statement did not mean there was not a need for planning or plans, or for others to read them; his statement simply meant that beyond a point, the fine print obscures. Test planning is a bit like that: to some level of detail, it is necessary to put plans down in black and white; beyond that, more detail probably detracts from effectiveness. Emergent protocols should be allowed to fill in the gaps.

Test Planning Essentials

The primary application of test planning is for unit tests, integration tests, and acceptance testing, not TDD. TDD, as already noted, is a rapid-fire design rhythm for which the team should develop design

metrics. Test metrics, on the other hand, require a test plan for context. Test planning requires standard definitions so that metrics convey the same information to all concerned. In other words, common definitions provide cohesion among planning elements.

Planning Test Flow

Here is an example planning exercise in terms of who will do what to see how all this fits together. The business and the developers or testers work collaboratively, step-by-step, on stories, conditions, and scenarios:

- Business users select functional stories and identify the business conditions for scripts to be written.
- Testers plan the number of scripts based on the business user's stories and business conditions.
- Testers group the conditions according to their natural affinity and plan how conditions will be applied to scripts to make unique scenarios.
- Testers evaluate coupling. Some changes in affinity grouping may be required to loosen the coupling among testing conditions.
- Project management and the team leader develop test plans, making assumptions about the pass and fail rates of scenarios, calling upon experience or other benchmarks.

Pass and fail rates are estimated by assuming a test-fix-test flow of "run the test . . . measure results, and fix observed failures. Then, rerun the test . . . fix any new failures, and rerun again if necessary". In this flow plan, all scripts are run once to see which pass. Those that fail are run again after diagnosis and repair; if any failures remain, the failed scripts are run yet again after fixes are applied.

Planning the Test-fix-test sequence

The test-fix-test sequence, which we call an attempt sequence, is relatively easy to plan. This attempt sequence is particularly appealing to agile managers because it is outcome oriented—the focus is on passes and failures and eventually getting all to green.

To illustrate the idea, and as a working assumption, we will plan a 30-50-20 attempt sequence, so named because each number is an assumed-for-planning-purposes passing percentage. The 30-50-20 sequence is explained in the following panel:

Planning the 30-50-20 Attempt sequence	
Attempt situation	Outcomes
30% of all scenarios pass on the first attempt	Three of 10 pass, seven of 10 fail, and those seven must be retested in the second attempt. The passing rate is 30 percent of all 10 scenarios
50% percent of all scenarios pass on the second attempt.	Seven first-attempt failures are rerun after fixes are applied; Plan for five of seven to pass; and two of seven to fail, an attempt passing rate of five of all the 10 scenarios, or 50 percent.
The remaining 20 percent of all scenarios pass on the	The two remaining failures are rerun after more fixes are applied;

third attempt.	The two pass and none fail on the third attempt. The attempt passing rate of two of the original 10 pass on this third attempt, or 20 percent

When combined, note that the 30-50-20 percentages equal 100 percent of the scenarios. In other words, everything eventually passes. Everything must work—in other words, pass the test—is one of the principles of agile methods. See the Appendix at the end of this

Estimating hours by parameters

To estimate hours for the test plan, more parameters are required:

Complexity parameter: For each object-under test, grade the complexity of its test script. Complexity, as always, is a judgment call based on experience and benchmarks. Although a complexity scale is often some ordinal rating like low, medium, high, or very high, a numeric scale is more useful for planning.

Complexity scale factor: Choose a scale such as a numerical binary sequence corresponding to the ordinals low, medium, high, and very high; choose a scale that provides reasonable range and separation between the ordinal values, like: 1, 2, 4, and 8, respectively.

Baseline Rank parameter: Select a script that will serve as the baseline "low-complexity" benchmark. Rank all scripts by relative complexity compared to the baseline benchmark. In any particular group of scripts, it is not unusual to find that not all complexities are represented. For instance, a particular group of scripts might have all complexities ranked as high (a numerical rating of 4) compared to the benchmark.

Tasks and hours: For a low-complexity script, make an absolute estimate of the hours and skills required to set up and test the script. This low-complexity absolute estimate will be the baseline. For scripts ranked higher than "low-complexity", multiply the baseline hours by the complexity scale factor.

By example, a very high complexity script would be planned for 8 times the resources as the baseline, using the 1-2-4-8 scale factor.

See the Appendix for a worked example.

Pipeline Grid

A tool for showing the test plan is the so-called pipeline grid. There are three elements of information:

1. Baseline
2. Day-to-day working plan
3. Actual performance

Week-by-week and cumulative-to-date data are often shown in grid format. Calendar time is shown horizontally as columns. The three plans, as above, are each a row in the grid.

- Effective hours, after adjustment for labor loss, are planned in the baseline and estimated for the working plan.

- Variances between the working plan and baseline require a strategy to converge the plans to zero variance.
- Actual hours are the hours recorded by participants for the work task.

However, solely looking at hours is only looking at resource consumption, a so-called input-input view that does not measure outcomes. Agile methodologists always focus on outcomes by putting the most attention on what flows out of the pipeline.

Planning the Scorecard

The testing scorecard is much like a pipeline monitor, watching the flow of things as they go by. A pipeline requires a temporal dimension, so plan the baseline according to a calendar. As an example to illustrate the scorecard planning, assume the following project test case:

Test case

- *Scripts:* ten scripts, all subject to the same 30-50-20 passing sequence
- *Script complexity:* two low, five medium, three high, and zero very high
- *Scenarios:* five scenarios for each of 10 scripts, 50 scenarios total (10 low, 25 medium, 15 high)

The 20-50-30 sequence is planned for the low, medium, and high scenarios; in effect, three different plans. Each plan is scaled for resources according to the complexity scale factor for that plan.

See the appendix for a worked example.

Summary and Takeaway Points

Our theme is that agile methods rigorously test to prove quality is designed-in. Demonstrating quality through testing builds trust with the sponsor, the stakeholders, and the customer. Quality means the product will work and satisfy when put into production.

In the XP methodology, a recommended practice is test-driven development (TDD). TDD is applicable to any agile methodology. TDD is a means to document requirements, begin the design effort, and establish a Red-Green-Refactor rhythm.

Good test metrics for unit, integration, and acceptance testing come from good test planning. Test planning is about estimating the number of scripts, scenarios, and instances and estimating their pass and failure rates. Scorecards keep track of actual pass and failures compared to baseline forecasts.

Appendix:

Scenario test case

Set-up: Suppose for a given script with all conditions taken into account, the team plans 25 scenarios.

The plan: Assume the 3-attempt *30-50-20* sequence is a reasonable expectation.

- First attempt*:* Plan to run the 25 instances and expect seven passes (0.3 X 25 = 7) on the first attempt, rounding down to the nearest integer, leaving 18 scenarios failed.
- Second attempt*:* Plan to run the 18 scenarios, expecting to get to a cumulative 80 percent passing rate (30 + 50 = 80 percent). This means 50 percent of the original 25, or 12.5, are

forecast to pass on the second attempt.
Rounding up to compensate for rounding down in the first attempt, expect 13 passes in the second attempt.
Cumulatively, after the second attempt, there are 20 passes (7 + 13 = 20) which is 80 percent of the original 25.
Five remain to pass.

- Third attempt: There are only five scenarios left that require yet a third attempt. Plan to run these five scenarios; the model forecasts all five to pass on the third attempt.
  At this point, all 25 of the original 25 have passed.

Summary: To summarize what has transpired, one script with 25 conditions (25 scenarios) expands to a 48-instance testing scope:

- One script with 25 scenarios will produce 48 instances (25 + 18 + 5 = 48) when rounded to a whole number.
- This expansion is 1.92 times the number of scenarios that began the test.
- The 30-50-20 sequence is an effort and schedule multiplier on scenarios.
- The multiplier effect is actually 1.9, disregarding any rounding errors:

Multiplier Effect

Multiplier of first instances + multiplier of second instances + multiplier of third instances

= Total multiplier for instances in three attempts
1 + (1 - 0.3) + (1 - 0.3 - .5) = 1 + 0.7 + 0.2 = 1.9
For 10 scenarios
10 + 7 + 2 = 19

If there are three pass rates corresponding to three attempts, then the multiplier effect is calculated:

1 + (1 - Rate 1) + (1 - Rate 1 - Rate 2)

- The pass rate is an effort multiplier
- The test plan anticipates a pass rate spread over a number of attempts.
- The consequence of different pass rates and multiple attempts is that the testing effort is multiplied.

About the author

John C. Goodpasture, PMP is a program manager, instructor, author, and project consultant specializing in technology projects

For many years, he has been one of the instructors for an online distance learning course in Agile project management. He was project director of an E-Business application development unit at Lanier Professional Services where his team delivered a number of successful projects using agile principles and practices.

He is the author and contributing of four other technical books in project management, numerous magazine and web journal articles in the field of project management, and has been an invited speaker at many professional project management events.

After graduating with a master's degree in engineering, John was a system engineer and program manager in the U.S. Department of Defense leading high technology programs. Subsequently, he managed numerous defense software programs while at Harris Corporation in Melbourne, FL., eventually finishing his corporate career as operations vice president for a document imaging and storage company.

He has coached many technology teams in new product development and functional process improvement, both in the United States and abroad, in industries as diverse as semi-conductor manufacturing and retail mortgages.

For more on the subject of project management and Agile methods, check out these websites: John blogs at johngoodpasture.com, and his work products are found in the library at www.sqpegconsulting.com.

Many of his presentations on agile methods are found at www.slideshare.net/jgoodpas. John maintains a professional profile at www.linkedin.com/in/johngoodpasture

Endnotes

1. Having a set of general rules that frame day-to-day tactics that adapt to the situation is an idea that comes from the study of *emergence.* Emergent processes and systems are those that can make changes in the input-to-output transformation so that outcomes are more in line with expectations based on input conditions. Emergence requires feedback from output to input and a means to incorporate the feedback to affect a better outcome. See Anderson, *Agile Management for Software Engineering,* 11-12.
2. Kent Beck, the father of XP, is the innovator most associated with TDD, although there are others—Ward Cunningham, Ron Jefferies, and Martin Fowler— who have been instrumental promoting TDD with developers and project managers alike.
3. See Beck, Test-Driven Development: By example, 11. See also Fowler, Patterns of Enterprise Application Architecture, 95.
4. Martin, *Mocks Aren't Stubs,* MartinFowler.com, January, 2007. For additional insight, see Walther, "TDD Tests are not Unit Tests," StephenWalther.com.
5. Covey, *7 Habits of Highly Effective People,* 95, Habit 2: Begin with the End in Mind.
6. User stories are short, small business scenarios that are functional descriptions of a user's need. In the iteration planning meeting, participating users tell the stories. These stories form a backlog for the development iterations. See Cohn, *User Stories Applied: For Agile Software Development,* 4.
7. Beck, Test-Driven Development: By Example, x.
8. Kent Beck has written about the first four problems discussed in the text. See Beck with Andres, *Extreme Programming Explained: Embrace Change,* 2nd ed. 50-51.
9. See the XP values defined by Kent Beck in Beck with Andres. The reader is reminded that system, product, application, deliverables, and outcomes are all used interchangeably to give generality to the discussion.
10. See Hoare, An Axiomatic Basis for Computer Programming.
11. Astels, Test-Driven Development—A Practical Guide, 62.
12. Stevens, Myers, and Constantine, *Structured Design,* 13 (2), 115-139.
13. Paulish, Architecture-Centric Software Project Management, 5.
14. D'Este, Decision in Normandy.