

# Chapter 10

## Acceptance Testing

*What's done, is done.*  
 -- *Macbeth Act 3, scene 2, 8–12*

*If it isn't tested, it doesn't exist.*  
 -- *Anonymous agile Master*

*We adopted agile and all the testers quit*  
 -- *as narrated by Lisa Crispin*

---

### Chapter 10 Table of Contents

Why Write About Testing in an Agile Requirements Book? .....	10-2
Agile Testing Overview .....	10-3
What is Acceptance Testing? .....	10-5
Characteristics of Good Story Acceptance Tests .....	10-7
They Test Good User Stories .....	10-7
They are Relatively Unambiguous and Test All the Scenarios .....	10-8
They Persist .....	10-8
Acceptance Test-Driven-Development .....	10-9
Acceptance Test Template .....	10-10
Automated Acceptance Testing .....	10-12
Automated Acceptance Testing Example: The FIT Approach .....	10-12
Unit Testing .....	10-14
Unit Testing in the Course of the Iteration .....	10-16
Component Testing .....	10-17
Summary .....	10-18

---

## *Why Write About Testing in an Agile Requirements Book?*

As a sanity check in preparing for this Chapter, I went to my bookshelf and looked at a number of texts on software requirements management, including my own [Leffingwell 2003&2000], for guidance on testing whether an application meets its requirements. Of course, I knew he wouldn't find much on testing there, if for no other reason than I knew I hadn't written much, either. And I wasn't surprised that other requirements authors haven't written much of anything on testing either.

So the question naturally arises: why do we feel compelled to write about testing now, in a book on agile requirements? The question itself reflects a traditional view, that historically, software requirements were somehow independent of their implementation - they lived a separate life - you could get them (reasonably right) at some point, mostly up front - the developers could actually implement them as intended - - they would be tested somewhat independently to assure the system worked as intended - and the customers and users would be happy with the result. Of course it never really worked that way, but it sure was easier to write about it.

In thinking in lean and agile terms however, we must take a much more systemic and holistic view. We understand that stories (requirements), implementation (code) and validation (acceptance tests, unit tests, and others) are not separate activities, but a continuous refinement of a much deeper understanding – so our thinking is different:

*No matter what we thought back then, this functionality is what the user really needs, and it's now implemented, working, and tested in accordance with the continuous discussions and agreements we have forged during development.*

*Just as importantly, we have instrumented the system (with automated regression tests) such that we can assure this functionality will continue to work as we make future changes and enhancements to the system.*

*Then, and only then, can we declare that our work is complete for this increment and move onto the next.*

That is the reason that we have taken a much more systemic view of “requirements” in this book, discussing users, agile teams, agile process, roles, product owners, and whatever else is necessary for a team to develop an application that, in the end, actually solves the user's problem.

However, when describing requirements in book form, the subject is fuzzier and more tangible at the same time. It's *fuzzier* - because you can't really tell where a story ends and its acceptance test begins or ends. Are the range of data elements in a user entry field included in the story, are they implied requirements attached to the story? Are they really details left for the acceptance test? Or are they perhaps so fine-grained that they may be covered solely in the unit tests for the method that does it?

And yet, the subject is *more tangible*, because the precise answers to these questions aren't so important. What is important is that we worked through a cycle of incremental information discovery - we collaborated, we negotiated, refined, we compromised - and we ended up with something that actually works for the users. In addition, we have captured the details of system behavior in a set of tests that will persist for all the time the

software continues to provide value to its users. So the requirements are implemented, complete and tested.

In this Chapter, we'll provide guidance to these questions, and an overview of how we achieve quality in our agile requirements practices. In agile, we simply can't do that without a discussion of testing.

---

## *Agile Testing Overview*

Given that we are taking a systemic view to requirements, across the team, program, and portfolio, we must also take a broader view of agile testing in general, so we'll know the context in which a discussion of acceptance testing can make sense.

Brian Marick, an early XP proponent (and a signer of the Agile Manifesto) has provided much of the thought leadership in this area and has developed a framework that many agilists use to think about testing in an agile paradigm. His philosophy of agile testing is as follows<sup>1</sup>:

*Agile testing is a style of testing, one with lessened reliance on documentation, increased acceptance of change, and the notion that a project is an ongoing conversation about quality.*

He goes on to describe two main categories of testing: *business-facing* and *technology-facing* tests<sup>2</sup>:

*A **business-facing test** is one you could describe to a business expert in terms that would (or should) interest her. If you ... wanted to describe what questions the test answers, you would use words drawn from the business domain: "If you withdraw more money than you have in your account, does the system automatically extend you a loan for the difference?"*

*A **technology-facing test** is one you describe with words drawn from the domain of the programmers: "Different browsers implement Javascript differently, so we test whether our product works with the most important ones."*

He further categorizes tests, whether business-facing or technology-facing, that are used primarily to *support programming* or to *critique the product*.

*Tests that **support programming** means that the programmers use them as an integral part of the act of programming. For example, some programmers write a test to tell them what code to write next. ... Running the test after the [code] change reassures them that they changed what they wanted. Running all the other tests reassures them that they didn't change behavior they intended to leave alone.*

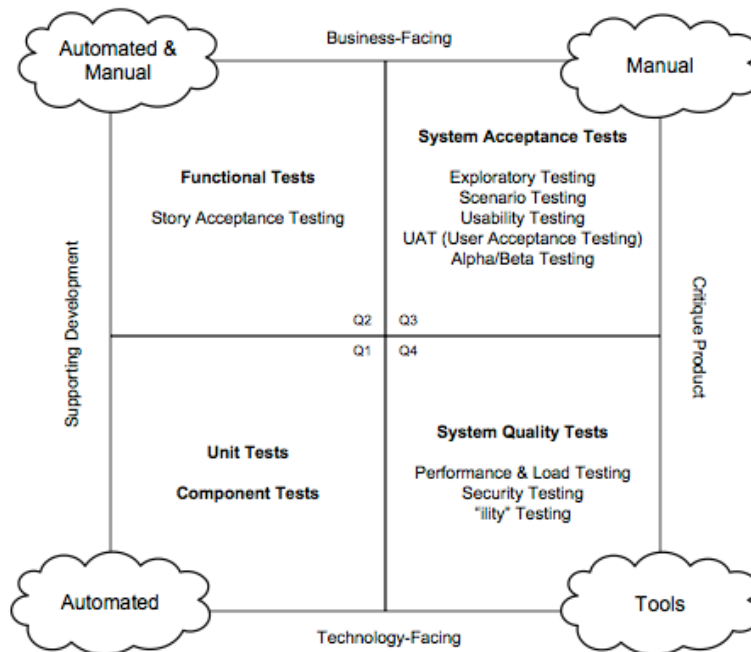
---

<sup>1</sup> [agilemanifesto.org/authors.html](http://agilemanifesto.org/authors.html)

<sup>2</sup> [www.exampler.com/old-blog/2003/08/21/](http://www.exampler.com/old-blog/2003/08/21/)

*Tests that **critique the product** are not focused on the act of programming. Instead, they look at a finished product with the intent of discovering inadequacies.*

In the text, *Agile Testing*, [Crispin 2009] Crispin and Gregory developed these concepts further. With a few minor adaptations for our context, we find an agile testing matrix in Figure 10-1:



**Figure 10-1** The agile testing matrix

In **Quadrant 1**, we find *unit tests* and *component tests*, which are the tests generally written by developers to test whether or not the system does what they intended it to do. As indicated on the matrix, these tests are primarily *automated*, as they can be implemented in the unit testing environment of choice.

In **Quadrant 2**, we find *functional* tests, in our case, consisting primarily of story level acceptance testing that the teams use to validate that each new story works the way their product owner, customer, and user intended. Many of these tests can be automated, as we'll see later, but some of these tests are likely to be manual.

In **Quadrant 3**, we find larger scale, systematic acceptance testing which determines whether the aggregate behavior of the system meets its usability and functionality requirements, including variations (scenarios) which may be encountered in actual use. These tests are largely manual in nature, as they involve users and testers using the system in actual or simulated deployment and usage scenarios.

In **Quadrant 4**, we find tests designed to determine whether the system meets all its other qualities and performs as reliably as needed. These are the nonfunctional requirements, or quality requirements, of the system. These tests are generally supported by a class of testing tools, such as load and performance testing tools, which are designed specifically to put various types of loads on the system to see where the system breaks down under load.

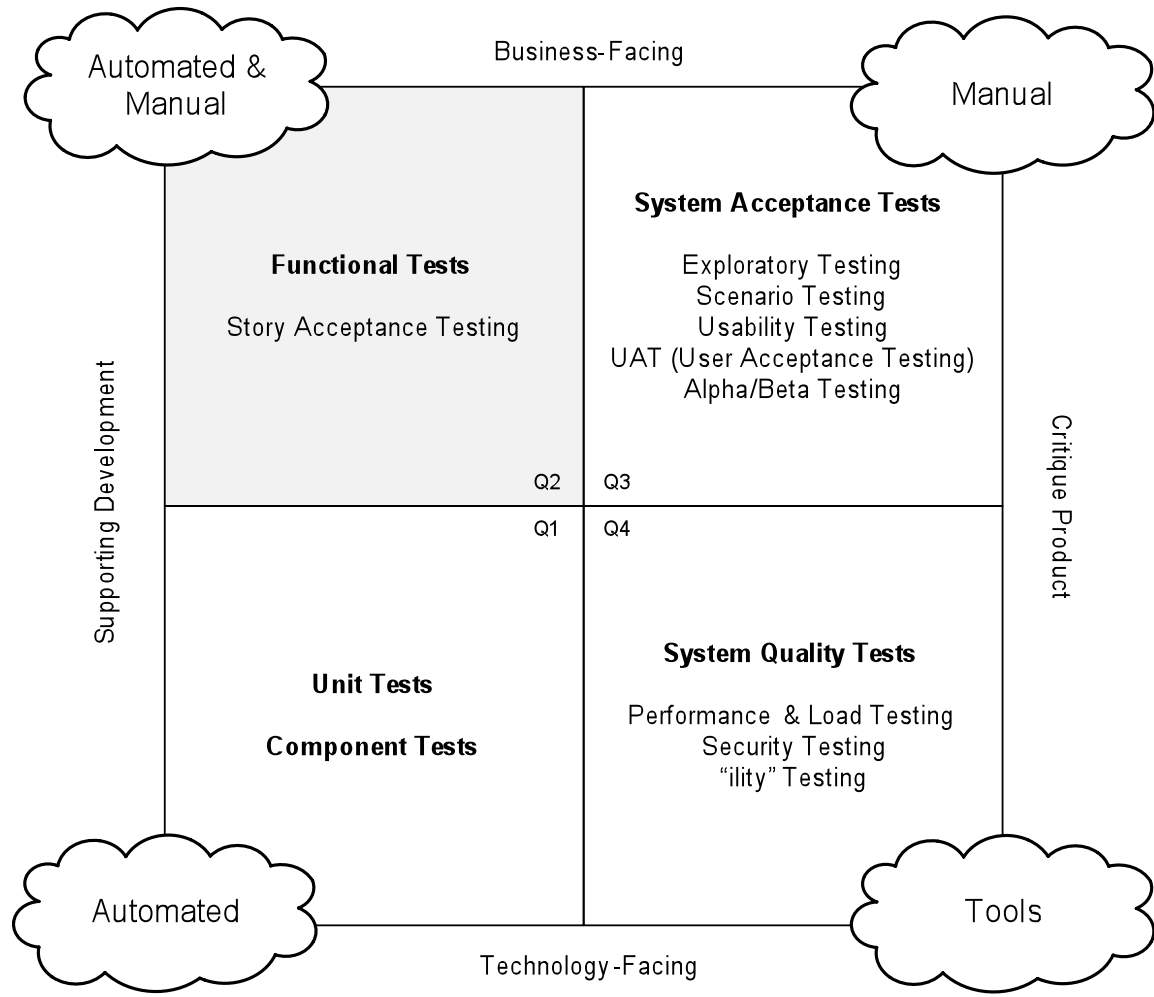
With this perspective, we have a way to think about the different types of testing we'll need to do to assure that a system performs as expected.

In this Chapter, we'll focus on two of these quadrants, Q2 and Q1. In later Chapters, we'll revisit the testing process to look at testing practices that support the tests indicated in Quadrants 3 and 4.

---

### *What is Acceptance Testing?*

The language around testing is as overloaded as any other domain in software development, so the words *acceptance testing* mean many different things to different people. However, in the context of agile development, we can see that there are two different uses of the term in the testing matrix. In Quadrant 2, we see functional, or *story acceptance tests* (SAT) as shown in Figure 10-2.

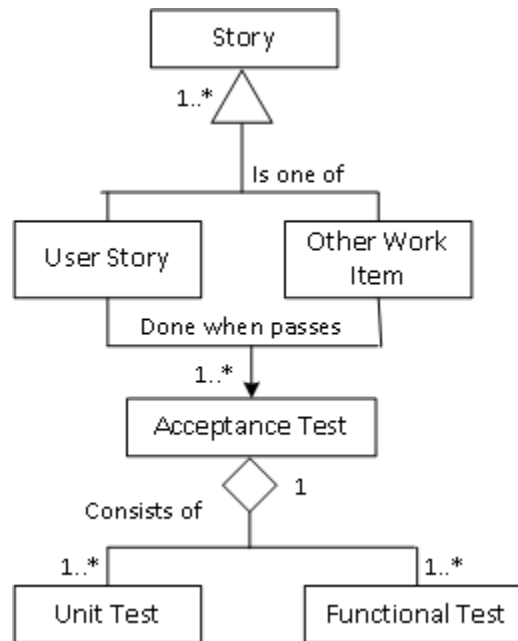


**Figure 10-2** Quadrant 2 Testing

SATs are high-level tests intended to assure the completeness of a user story. User story acceptance tests are written to assure that each increment of software is delivering value, and that the project is progressing in a way that will ultimately satisfy the needs of the business owner. Generally:

- They are written in the language of the business domain (they are business facing tests from quadrant 2)
- They are developed in a conversation between the developers, testers, and product owner
- While anyone can write tests, the product owner, as business owner/customer proxy, is the primary owner of the tests
- They are *black box* tests in that they only verify that the outputs of the system meet the conditions of satisfaction without concern for how the result is achieved
- They are carried out during the course of the iteration in which the story is implemented

In our agile requirements model, these acceptance tests appear as illustrated in Figure 10-3.



**Figure 10-3** A story cannot be considered done until it passes one or more acceptance tests

This means that new acceptance tests are developed for every new story. If a story does not pass its test, the teams get no credit for the story and the story is carried over into the next iteration, where the code or the test, or both, are reworked until the test passes.

---

### *Characteristics of Good Story Acceptance Tests*

If user stories are the workhorse of agile development, the key artifact that carries the value stream to the customer, then story acceptance tests are the workhorse of agile testing, and teams must spend much time defining, refining, and negotiating the details of these tests. Good story acceptance tests exhibit the characteristics described in the following sections:

#### **They Test Good User Stories**

An attribute of a good SAT is that it is associated with a good user story. In Chapter 6, we described the INVEST model for user stories and the quality of our acceptance tests are fully dependent on the native quality of the story itself. In particular, the user story to which a test is associated has to be Independent, Small, and Testable. If the development of an acceptance test illustrates that the story is otherwise, then the story itself must be refactored until it meets these criteria. So, to get a good acceptance test, we may need to refactor the story first. For example:

*As a consumer, I am always aware of my current energy costs*

becomes

As a consumer, I always see current energy pricing reflected on my portal, so that I know that my energy usage costs are accurate and reflect any utility pricing changes

### **They are Relatively Unambiguous and Test All the Scenarios**

The user story itself is a lightweight, expression of intent. The acceptance test carries the detailed requirements for the story. As such, there should be relatively little ambiguity about the details of the acceptance test. In addition, the acceptance test must test all the scenarios implied by the story, otherwise, the team won't know when the story is sufficiently complete in order to be able to be presented to the product owner for acceptance. For example:

Story: As a consumer, I always see current energy pricing reflected on my portal, so that I know that my energy usage costs are accurate and reflect any utility pricing changes

Acceptance test:

1. Verify the current pricing is always used and the calculated numbers are displayed correctly on the portal (see attachment for format)
2. Verify the pricing and the calculated numbers are updated correctly when the price changes
3. Verify the "current price" field itself is updated according the scheduled time
4. Verify the info/error messages when there is a fault in the pricing. (see approved error messages attached)

### **They Persist**

For those considering agile, one of the mysteries about agile development, and indeed a key impediment to adoption, is a common sense question: "If developers don't document much, and there are no software requirements specifications as such, how are we supposed to keep track of what the system actually does? After all, we are the ones responsible for assuring that it actually works, now and in the future. Isn't that something we have to know, not just once, but in perpetuity?"

The answer is yes, indeed, we do have to know how it works, and we have to routinely regression test it to make sure it continues to work. We do that primarily by persisting and automating (wherever possible) acceptance tests and unit tests (discussed later).

User stories can be safely thrown away after implementation, that keeps them lightweight, team friendly, and fosters negotiation, but acceptance tests persist for the life of the application. We have to know that the current price field didn't just get updated once when we tested it, but that it gets updated every time the price changes, even when the application itself has been modified.

A clarification is needed here. Of course, we don't imagine that life would be sweet if someone was given a pile of code and tests and then told to go figure out what the entire system does! Reasoning backwards from the tests and code is usually a hopeless proposition (although it has happened before and will happen again, remember Y2K?)

The missing link to understanding the legacy and functionality of a system is the user documentation such as user guides, online help, etc. Armed with such user-oriented material, and the code, and the acceptance tests, a system developer has a fighting chance at understanding the system well enough to make improvements, fix bugs, add features, etc.

---

### *Acceptance Test-Driven-Development*

Beck [2003] and others have defined a set of XP-compatible practices for agility described under the umbrella label of Test Driven Development, or TDD. In TDD, the focus is on writing the unit test before writing the code. For many, TDD is an assumed part of agile development and is straightforward in principle:

1. Write the test first. Writing the test first forces the developer to understand exactly what required behavior of this new code is to be as well how the system will be tested.
2. Run the test and watch it fail. Because there is as yet no code to be tested, this may seem silly initially, but this accomplishes two useful objectives: (a) it tests the test itself and any test harnesses that hold the test in place, and (b) it illustrates how the system will fail if the code is incorrect.
3. Write the minimum amount of code that is necessary to pass the test. If the test fails, refactor as necessary until a module is created that routinely passes the test.

In XP, this practice was primarily designed to operate in the context of unit tests, which are developer-written tests (also code) that test the classes and methods that are used.

However, the philosophy of TDD applies equally well to story acceptance testing as it does to unit testing. This is called *acceptance test driven development*, and whether it is adopted formally or informally, many teams write the acceptance test first, before developing the code. The acceptance tests serve to record the decisions made in the conversation (Card, Conversation, Confirmation) so that the team understands the specifics of the behavior the card represents. The code follows logically.

Proponents of this model argue that doing so is lean thinking that reduces waste, and substantially increases the productivity of the team. This was illustrated best in the simple equation that Amir Kolskey, of Net Objectives, showed on a whiteboard. As shown in Figure 10-4, Amir wrote<sup>3</sup>:

---

<sup>3</sup> Personal interaction between Kolskey and Leffingwell

Where:

$C_t$  is the time to write code

$T_t$  is the time to write test

$H_t$  is the time to hook test

Time to complete story if you write test first

$$= T_t + C_t + H_t$$

Time to complete the story if you don't

$$= C_t + T_t + H_t + R_t$$

Where  $R_t$  is the rework time necessary to pass the test once the test is understood and available.

**Figure 10-4** The simple math behind acceptance test driven development

If  $R_t$  is 0, of course, then there is no savings. However, we all know that  $R_t$  isn't always zero, and we have to write the test anyway, so why not write the test first, just to be sure?

---

### Acceptance Test Template

At each iteration boundary, or whenever a story is to be implemented, it comes as no surprise to the team that they need to create an acceptance test that further refines the details of a new story, and defines the conditions of satisfaction which will tell the team when the story is ready for acceptance by the product owner. In addition, in the context of a team and a current iteration, the domain of the story is pretty well established and certain patterns of activities result which can guide the team to the work necessary to get the story accepted into the baseline.

To assist in this process, it can be convenient to the team to have a checklist – a simple list of things to consider – to fill out, review, and discuss each time a new story appears. Crispin [Crispin 2009] provides an example of such a story acceptance testing checklist. Based on our experience using this checklist, we provide an example of a modified *acceptance testing template* in Table 10- 1.

Story		
Story ID: US123	As a consumer, I always see current energy pricing reflected on my portal, so that I know that my energy usage costs are accurate and reflect any utility pricing changes	
Conditions of Satisfaction		
1. Verify the current pricing is always used and the calculated numbers are displayed correctly on the portal (see attachment for format) 2. Verify the pricing and the calculated numbers are updated correctly when the price changes 3. Verify the "current price" field itself is updated according the scheduled time 4. Verify the info/error messages when there is a fault in the pricing. (see approved error messages attached)		
Modules Impacted		
Pricing RESTlet API	Impact: Amend protocol to allow pricing data	
In Home Display	Impact: Refactor pricing schedule to support pricing programs to display on the In Home Display	
Portal	Impact: Refactor pricing schedule to support pricing programs to display on the portal	
Documents Impacted		
User Guide	Impact: Add new section on pricing	
Online Help	Impact: Update online help to reflect pricing programs	
Release Notes	Impact: Document defects in release notes	
Utility Guide	Impact: Document pricing schedule changes	
Test Case Outline		
Test ID: <input checked="" type="checkbox"/> Manual <input type="checkbox"/> Automatic	Outline: 1. Check Pricing -When there is no pricing info for a user 2. Change of Pricing <ul style="list-style-type: none"> <li>• When there is a pricing change in all allowed ways</li> <li>• Effective in the future</li> <li>• Effective in the past before the current pricing</li> <li>• Effective in the past but later than the current pricing</li> </ul> 3. Our current release does not support pricing change in the middle of a billing cycle. 4. Check the dashboard Billing Period Consumption and the Current Bill to Date.	
Communications		
Internal	Involved Parties: Marketing, Sales, Product Management	Message: This is a new marketable feature
External	Involved Parties: Utilities	Message: This is a new feature to support new programs

**Table 10- 1** An acceptance testing template

Since each team is a different context, their templates will differ, but the simple act of creating a template as a reminder of all the things to think about, benefits the team and

increases the velocity with which they can further elaborate and acceptance test a new story.

---

### *Automated Acceptance Testing*

Because acceptance tests run at a level above the code, there are a variety of approaches to executing these tests, including manual tests. However, manual tests pile up very quickly, (the faster you go, the faster they grow) and eventually, the number of manual tests required to run a regression slows down the team and introduces delays in the value stream.

To avoid this problem, most teams know that they have to automate most of their acceptance tests. They use a variety of tools to do so, including database-driven tests, Web UI testing tools, and automated tools for record and playback. However, many agile teams have discovered that some of these methods are labor-intensive and can be somewhat brittle and difficult to maintain because they often couple tightly to the implementation.

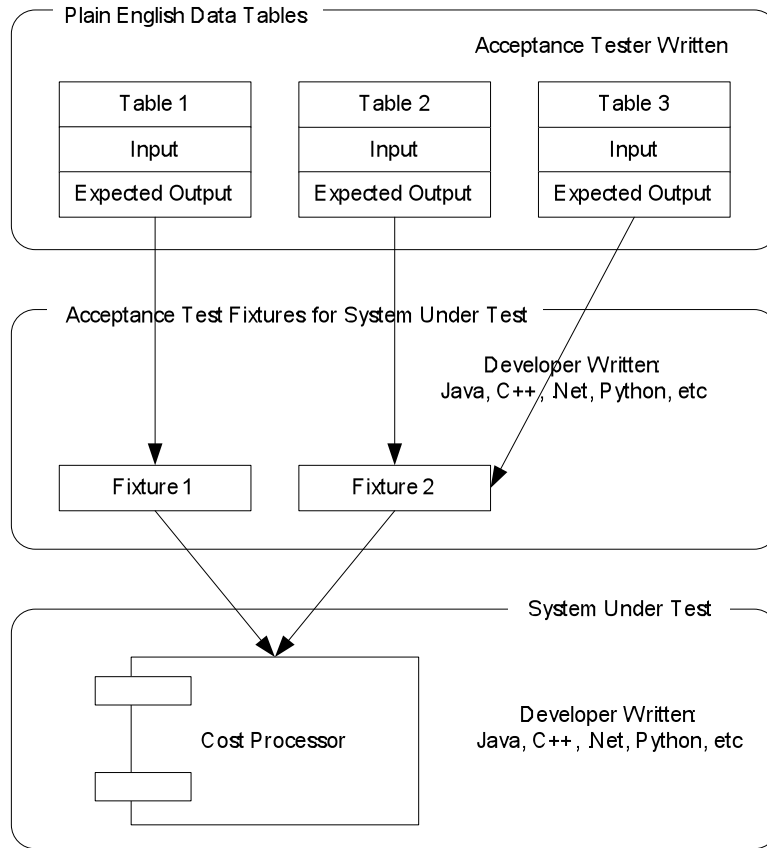
A more beneficial approach is to take a higher level of abstraction that works directly against the business logic of the application and one that is not encumbered by the presentation layer or other implementation details.

#### **Automated Acceptance Testing Example: The FIT Approach**

One such method is the FIT (Framework for Integrated Tests) method created by Ward Cunningham [Cunningham and Mugridge 2005]. This open source framework was designed to help with the automation of acceptance testing in a fast-moving agile context.

The FIT approach mirrors the unit testing approach in that the tests are created and run against the system under test, but they are not part of the system itself. FIT is a scriptable framework that supports tests being written in table form (any text tool will work) and saved for input as HTML. Therefore, these tests can be constructed in the business language (input and expected results) and can be written by developers, product owners, testers or anyone on the team capable of building the necessary scripts.

Another open source component, FitNesse, is a wiki/Web-based front end for creating text tables for FIT that also provides some test management capability. FIT uses data-driven tables for individual tests, coupled with fixtures or methods written by the developers to drive the system under test, as Figure 10-5 illustrates.



**Figure 10-5** Acceptance test framework for the costing processor

During the course of each iteration, new acceptance tests are developed and validated for each new story in the iteration, and these tests are then added to the regression test inventory. These suites of acceptance tests can be run automatically against the system under test at any time to assure that the build is not broken by the new code.

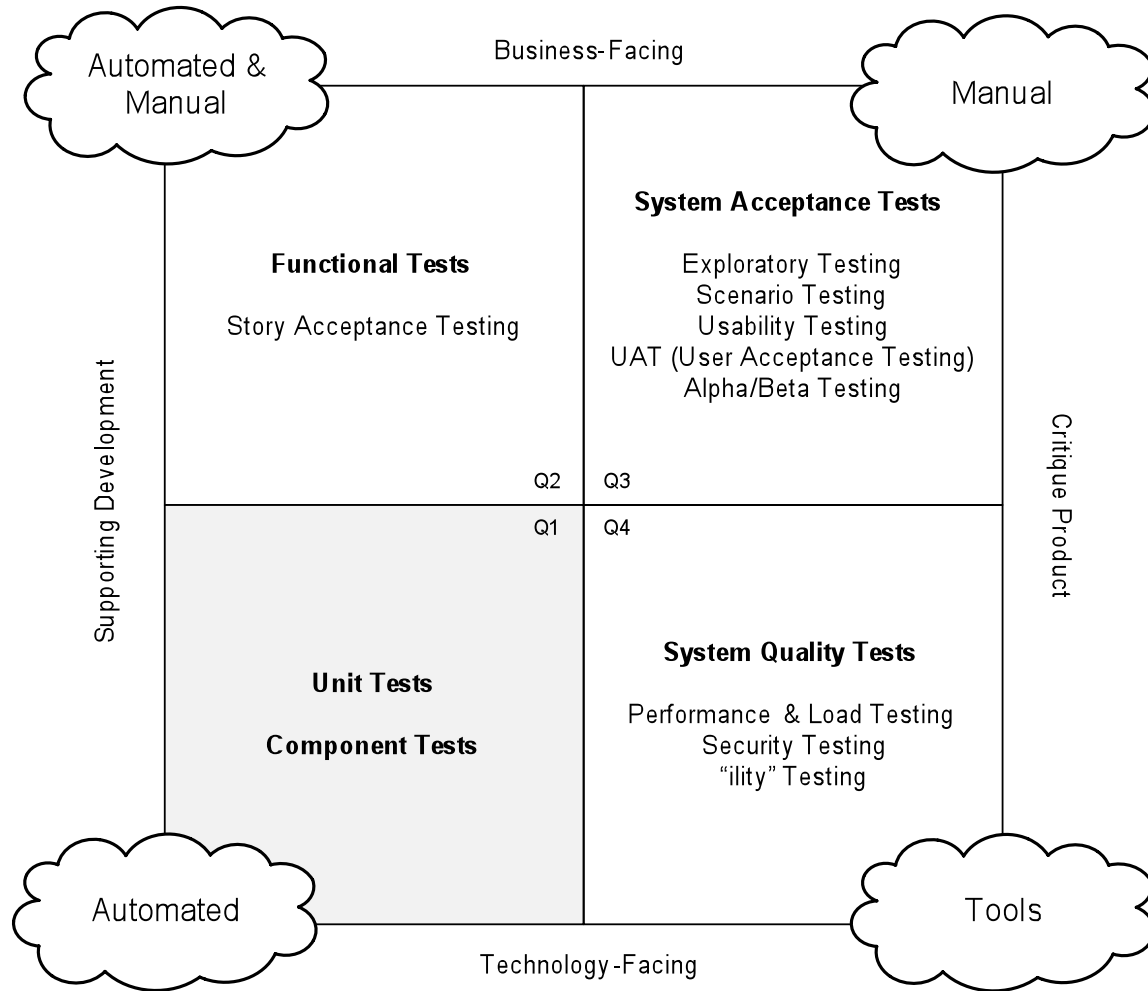
As with unit tests, the FIT approach requires continuous involvement by the developers in establishing the acceptance test strategy and then writing the fixtures to support the test, illustrating yet again that the define/build/test team is the necessary structure for effective agile development. As always, the team’s goal is to develop and automate tests within the course of the iteration in which the new functionality is introduced.

In some application domains, FIT does not provide an appropriately configurable framework. Sometimes, teams often build custom frameworks that mirror this approach, but work in the technologies of their implementation.

In any case, whatever can’t be automated eventually slows the team down, so continuous investments in testing automation infrastructure are routine items on a team’s backlog, and like any other backlog item, they must be appropriately prioritized by the product owner to achieve sustainable high velocity.

## Unit Testing

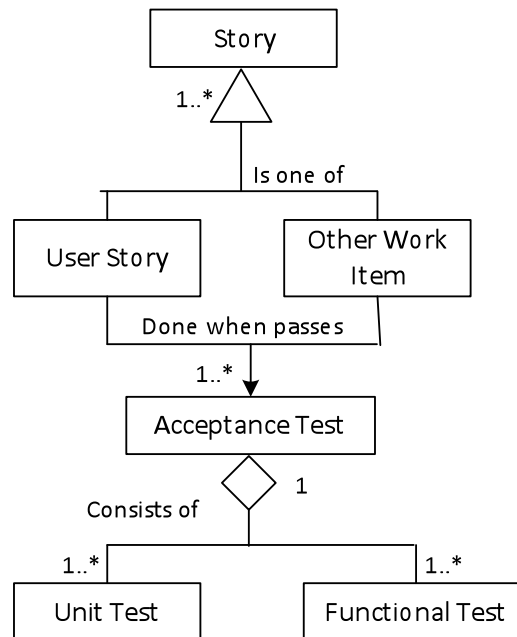
Before we leave this Chapter, we must drill down one more level into testing and discuss Quadrant 1 in the agile testing picture. In Quadrant 1, we see both *unit tests* and *component tests*, and together, they complete the *Support Development* side of the agile testing picture.



**Figure 10-6** Unit testing in Quadrant 1

Unit testing is the white box testing whereby developers write code to test the code they developed for the system. In so doing, the understanding of the user story is further refined and additional details about a user story can be found in the unit tests that accompany the code. For example, the presentation syntax and range of legal values for current price field can likely be found in the unit tests, rather than the acceptance tests, as otherwise the acceptance tests are long, unwieldy, and cause attention to the wrong level of detail.

Unit tests are depicted alongside functional tests, in our requirements model as shown in Figure 10-7:



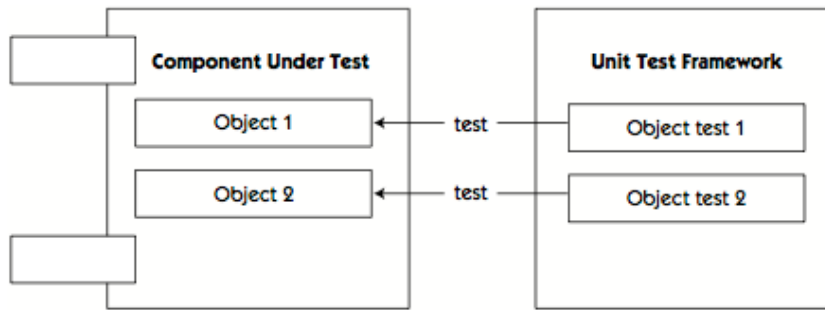
**Figure 10-7** The role of unit tests in the meta-model

A comprehensive unit test strategy prevents QA and test personnel from spending most of their time finding and reporting on code-level bugs and allows the team to move its focus to more system-level testing challenges. Indeed, for many agile teams, the addition of a comprehensive unit test strategy is a key pivot point in their move toward true agility—and one that delivers the “best bang for the buck” in determining overall system quality.

The history of agile unit testing closely follows the development of XP because XP’s test-first practices drove developers to create low-level tests for their code prior to, or concurrent with, the development of the code itself. The open source community has built unit testing frameworks to cover most forms of testing, including Java, C, C++, XML, HTTP, and Python, so there are unit testing frameworks for most languages and coding constructs an agile developer is likely to encounter.

These frameworks provide a harness for the development and maintenance of unit tests and for automatically executing unit tests against the system under development. Unit testing the energy cost calculator might be a matter of writing unit tests against every object in the component, as Figure 10-8 illustrates.

The unit tests themselves are not part of the system under test and therefore do not affect the performance of the query processor at runtime.



**Figure 10-8** Unit testing the energy cost calculator

### Unit Testing in the Course of the Iteration

Because the unit tests are written before or concurrently with the code and because the unit testing frameworks include test execution automation, all unit testing can be naturally accomplished within the iteration. Moreover, the unit test frameworks hold and manage the accumulated unit tests, so regression testing automation for unit tests is largely free for the team. For these and other reasons, unit testing is a cornerstone practice of software agility, and any investments a team makes toward more comprehensive unit testing will be well rewarded in quality and productivity.

Figure 10-9 shows a typical small unit test for Tendril’s energy costing module that takes the consumer’s time-based pricing structure and runs a small sample that will exercise many of the costing module’s pathways<sup>4</sup>. This test ensures that the costing algorithm effectively accommodates price changes that may occur at times other than standard day boundaries. Note that many more unit tests will be required before the costing module can be considered to be *done*. This is just one of many unit tests using the JUnit testing platform. This unit test example is one of many associated with calculating current costing but it is automated and is an integral part of the regression testing package. The sample shows another useful feature in that there are comments embedded in the test to explain what the test results are supposed to be. The follows an old adage, “If you don’t comment the program, how will the computer know what you want?”

<sup>4</sup> Thanks to Ben Hoyt of Tendril Networks Inc for this example.

```

@Test
public void testDailyCost_MultiplePrices() {
    List<ConsumptionValue> consumptionValues = new ArrayList<ConsumptionValue>();
    consumptionValues.add(new ConsumptionValue(TUESDAY_NOON, new Double(100)));
    consumptionValues.add(new ConsumptionValue(THURSDAY_NOON, new Double(200)));

    List<TemporalPrice> prices = new ArrayList<TemporalPrice>();
    prices.add(new TemporalPrice(new BigDecimal(".10"), SUNDAY, FIXED_PRICE));
    prices.add(new TemporalPrice(new BigDecimal(".25"), WEDNESDAY_NOON, FIXED_PRICE));

    DailyConsumptionHistory dailyConsumptionHistory =
        new DailyConsumptionHistory(new DayRange(SUNDAY, 7), consumptionValues,
prices);

    DailyCost dailyCost = dailyConsumptionHistory.getDailyCost(new
Day(WEDNESDAY_NOON));
    assertNotNull(dailyCost);

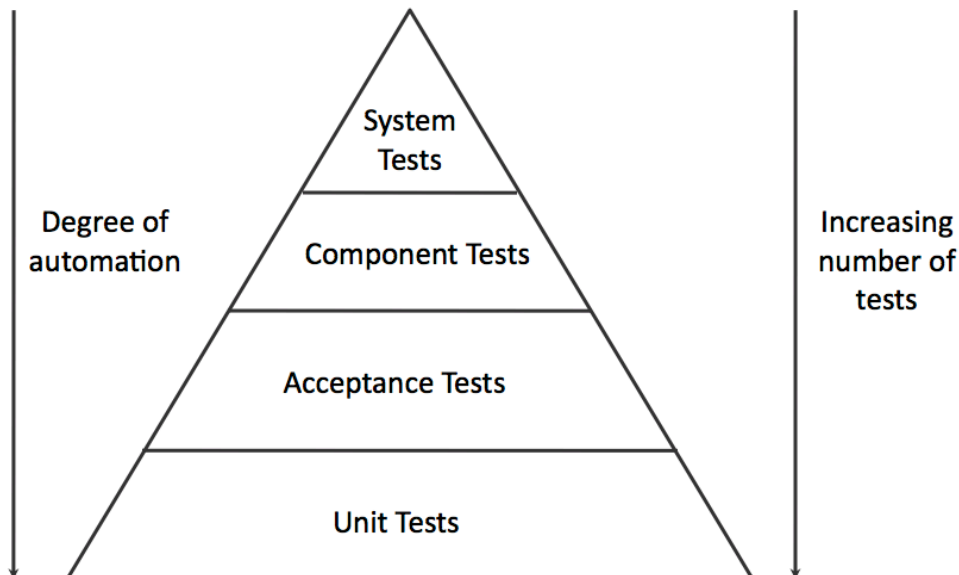
    /* First half of Wednesday is .10 / kWh, second half is .25 / kWh */
    /* 50 kWh burned that day = 25 * .10 + 25 * .25 = 2.50 + 6.25 = 8.75 */
    assertEquals(new BigDecimal("8.75"), dailyCost.getCost());
}

```

**Figure 10-9** A sample unit test

## Component Testing

In *Scaling Software Agility* [Leffingwell 2007], Leffingwell described agile testing in four layers, which can be represented in the pyramid shown in Figure 10-10<sup>5</sup>.



**Figure 10-10** Agile test and testing automation pyramids

In this figure, the unit tests we have just discussed are at the base of the pyramid. The shape of the pyramid is not accidental, as there are far more unit tests in agile

<sup>5</sup> Similar pyramids occur in Cohn [200x] and Crispin [2009].

development, and a greater investment, than there are at other levels of the pyramid. Fortunately, automation is a big help, because the degree of “automatability” also is higher at the bottom of the pyramid, than it is at the top where, more manual, exploratory, and periodic tool-based testing must be applied.

At the next level, we find the acceptance tests that have been the primary subject of this Chapter. The last form of testing we need to describe here is the component testing level of the pyramid.

Component testing is used to test larger scale components of the system, which may exist along architectural layers, that are simply services that provide value to new features or other components.

Testing tools and practices for implementing component tests vary according to the nature of the component. For example, unit testing frameworks can hold arbitrarily complex tests written in the framework language (Java, C, etc.), so many teams use their unit testing frameworks to build component tests. Acceptance testing frameworks, especially those at the level of **http Unit** and **XML Unit**, are also employed. In other cases, developers may use testing tools or write fully custom tests in any language or environment that is most productive for them.

---

## Summary

In this Chapter, we’ve described acceptance testing as an integral part of agile requirements management. For if a requirement (story) is not tested, then unless its simply work in process, which we indeed try to minimize, it doesn’t really have any value to the team or to the user. With this discussion, we described the agile testing approach to assure that each new story works as intended, *as* it is implemented. This covers Quadrant 2, Functional Testing (story acceptance testing), of the agile testing matrix.

We also describe the testing necessary to assure systematic quality from the perspective of Quadrant 1, Unit and Component tests. Unit tests are the lowest level of tests written by the developer to assure that the actual code (methods, classes, and functions) works as intended. Component tests are higher-level tests that are written by the team to assure that the larger components of the system, which aggregate functionality along architectural boundaries, also works as intended. We also described how the team must endeavor to automate all the testing that is possible, or else they will simply build a pile of manual regression tests that will eventually decrease velocity and slow down value delivery.

Form an overall testing perspective, we have yet to cover Quadrant 3, System Acceptance tests, and Quadrant 4, System Quality (Nonfunctional requirements) Tests, we’ll cover those in Chapters **XX** and **YY** respectively.

## References

- Beck, K., (2003). *Test-Driven Development*. Boston: Addison-Wesley
- Crispin, L., & Gregory, J. (2009). *Agile Testing: a Practical Guide for Testers and Agile Teams*. Reading: Addison-Wesley Professional.
- Cohn, M., (2004). *User Stories Applied*. Boston: Addison-Wesley.
- Leffingwell, D., & Widrig, D. (2000). *Managing Software Requirements: A Unified Approach*. Boston: Addison-Wesley.
- Leffingwell, D., & Widrig, D. (2003). *Managing Software Requirements: A Use-Case Approach*. Boston: Addison-Wesley
- Leffingwell, D., (2007). *Scaling Software Agility*. Boston: Addison-Wesley
- Mugridge, R., & Cunningham, W. (2005). *Fit for Developing Software*. Upper Saddle River: Prentice Hall Professional Technical Reference.