ESSENTIAL Acceptance Testing

Discussing agile acceptance testing techniques

Have traditional agile testing techniques become testing dogma? Have we adopted a cargo cult of testing ? Do these techniques really help build great products? Do they help us to get from concept to cash or are they holding us back?

Toby Weston

Essential Acceptance Testing

Discussing agile acceptance testing techniques

Toby Weston

This book is for sale at http://leanpub.com/essential_acceptance_testing

This version was published on 2013-10-30

Tweet This Book!

Please help Toby Weston by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I’m reading the #eat_book via @leanpub

The suggested hashtag for this book is #eat_book.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#eat_book

In memory of Shaun Bowden

Contents

What’s in the full version

The full version discusses the conventional acceptance testing strategies used by many agile teams today. In Part 1, it describes generally accepted strategies, their motivations, pitfalls and techniques to maximise success. It talks about how testing influences design and how to avoid the common problem of too many tests and specification overload.

If you want tips applying conventional acceptance testing strategies, Part 1 can help you get started and avoid common mistakes. The Introduction section and Typical Process Overview from Part 1 is included in this sample.

In Part 2, the book discusses why some of these techniques are fundamentally floored and tries to pose some difficult questions. Has acceptance testings techniques become dogma? Can stories really have business “value”? How can we test value? Can we run thousands of acceptance tests quickly? Part 2 also includes a detailed design section using example from a sample application built for the book.

If you’re interested in what’s beyond the canon, Part 2 may get you thinking. Inspired by real world frustrations and lean principles, Part 2 questions the de facto agile stance on testing.

Full table of contents

Part 2 - Discussion and alternatives

Problems acceptance testing can fix

- Communication barriers
- Lack of shared memory
- Lack of collective understanding of requirements
- Blurring the “what” with the “how”
- Ambiguous language
- Lack of structure and direction
- Team engagement

Problems acceptance testing can cause

- Communication crutch
- Hand off behaviour
- Technical overexposure
- Cargo cult
- Command and control structures
- Construct validity
- Artificial constraints

Business value (not yet written)

- What is “value”?
- Measuring “value”

Alternatives to acceptance tests

- Don’t write acceptance tests
- Use a ports and adapters architecture
- Don’t specify
- Measure don’t agree
- Log don’t specify

How design can influence testing

- Sample application
- Coupled architecture
- Decoupled architecture using ports and adapters
- Testing end-to-end (system tests)
- Summary of test coverage

- Benefits using ports and adapters
- Disadvantages using ports and adapters

Common pitfalls (not yet written)

- Features hit production that the customer didn't want
- Users describe solutions not problems
- Users can't tell how the system is supposed to behave
- Users can't tell if feature x is already implemented
- Tests repeat themselves
- The acceptance test suite takes forever to run
- Intermittent failures in tests

Q&A (not yet written)

- What do we mean when we say "acceptance testing"
- How do I manage large numbers of acceptance tests?
- How do you map acceptance tests to stories in say JIRA?
- How does applying acceptance testing techniques help us focus on reducing complexity?
- When would you not write stories? Acceptance criteria?
- How does agile acceptance testing differ from conventional style UAT?
- What are some other testing strategies? How does acceptance testing fit in?
- How do you layer various types of testing to maximise benefit?
- How does exception testing fit with unit and end-to-end tests?
- Aren't acceptance tests slow with high maintenance costs?
- What's the best way to leverage CI servers like TeamCity and Jenkins?
- Where does BDD fit in?
- Can I run acceptance tests in parallel?
- How can I run acceptance tests which exercise business processes than span multiple business days?
- How should I setup and tear down data?

Part 3 - Specification testing frameworks

The frameworks (not yet written)

- Concordion/.NET
- Yatspec
- Fit

Part 1 - An agile approach to acceptance testing

Part 1 introduces an acceptance testing approach centered around clearly defined requirements, customer authored acceptance criteria and the implementation of executable tests to exercise that criteria. This approach is used by many successful agile teams today.

Whilst this part describes the approach, Part 2 goes on to discuss why it's not always the best approach.

Introduction

What is an acceptance test?

Before we start, we should agree on some working definitions. Deciding on a definition of acceptance test can be contentious. Different people have different interpretations. So what is an acceptance test?

> An acceptance test is a set of examples or a specification that helps the customer "accept" that a system behaves as intended.

What would help a customer "accept" a system works as intended? The customer gains confidence if they're able to define requirements and see those requirements manifest as behaviour in a running system. For example, the customer might define their expectations and verify these against running code in a demo. The important point here is that the customer's criteria for acceptance are recorded and verified against a running system.

Recording the runtime behaviour of a system after the fact is only half the story. Deciding *what* that behaviour should be before starting work is the tricky part. Getting input from customers before building out the system is useful in ensuring we build the *right* system. Acceptance tests are a great vehicle for discussing and formalising these requirements.

> An acceptance test is a set of examples or a specification that helps the customer "accept" that a system behaves as intended. The test is used to specify required behaviour and to verify that behaviour against a running system.

Many teams emphasize that acceptance tests should be customer authored. If we write software that nobody wants, there's not much point in writing it. The customer can express their requirements in the form of acceptance criteria; a specification against which the system can be verified.

In Agile Testing, Lisa Crispin and Janet Gregory describe acceptance tests as

> "Tests that define the business value each story must deliver. They may verify functional requirements or non-functional requirements such as performance or reliability … Acceptance test is a broad term that may include both business facing and technology facing tests."

An important addendum to our definition then should be that acceptance tests don't have to be about just business behaviour, they can also be about broader system qualities such as non-functional requirements and usability. It's still about customer confidence.

Unfortunately, Crispin's definition talks specifically about *stories* and business *value*. I say unfortunate because acceptance tests may or may not have anything to do with user stories. We'll talk about stories later but in our definition, we're talking generally about system behaviour and brushing over the idea that stories can describe that behaviour. A story isn't necessarily a prerequisite for an acceptance test. Similarly, business *value* is a tricky thing to quantify, so measuring it in a test can be a challenge. We'll talk more about that later too.

Applying the addendum to our definition gives the following.

> An acceptance test is a set of examples or a specification that helps the customer "accept" that a system behaves as intended. The test is used to specify required behaviour and to verify that behaviour against a running system. Acceptance tests are not limited to confirming business behaviour but can also verify that broader, non-functional objectives have been met.

What are acceptance criteria?

We'll often use the terms acceptance criteria and acceptance test interchangeably but really they're distinct.

Acceptance criteria are the set of criteria that, when verified against a running system, give confidence to the customer that the system behaves as intended. They represent the requirements or specification for a small set of functionality and are written in such a way as to be quantifiable. They're typically defined when doing analysis and since they're mostly concerned with business requirements, the customer is best placed to define them. Non-functional requirements, despite affecting the customer, usually end up being championed by technical stakeholders.

Defining the criteria is a useful step in really understanding what's required. It helps us define the scope of a feature so developers know when to stop. Importantly, it also helps the team to drive out a shared understanding of the requirements. Criteria should be implementation independent and written at a high level. We then *implement* the criteria in terms of one or more acceptance tests.

A single criterion ("the total basket value is displayed correctly") may require multiple examples to be comprehensive (what *exactly* does it mean to "display correctly"?). That's where implementing the acceptance *criteria* as executable acceptance *tests* comes in.

Referring back to our definition, we're emphasising that tests are executable and criteria are not. An acceptance test is a physical test artifact. It may be a xUnit test written in the language of choice, a test script that requires a human to step through, a record-replay style UI test or even a checklist on a scrap of paper.

Adding this dimension to our definition gives us the following.

> An acceptance test is a set of *executable* criteria that helps the customer "accept" that a system behaves as intended. The test is used to specify required behaviour and to verify that behaviour against a running system. Acceptance tests are not limited to confirming business behaviour but can also verify that broader, non-functional objectives have been met.

Acceptance criteria *become* acceptance tests. Attributes that describe acceptance tests also describe acceptance criteria with the additional fact that tests should be *executable*. Executing acceptance tests verify that the acceptance criteria have been met.

Attributes of acceptance criteria and tests

Acceptance Criteria	Acceptance Tests
Document behaviour	Document behaviour
Are specific	Are specific
Are illustrative	Are illustrative
Are quantitative	Are quantitative
Require discussion	Require discussion
Are agreed	Are agreed
	Are executable

What is a story?

Acceptance criteria are usually discussed in terms of *user stories* so it's worthwhile making sure we have a common understanding of what makes up a story. Agile processes often focus on stories as a way of gathering requirements and organising them into deliverable chunks that have *business value*. In Planning Extreme Programming, Kent Beck and Martin Fowler describe a user story as "a chunk of functionality that is of value to the customer". It's common to associate acceptance tests with stories. Once the tests are passing, a story is considered finished. There's a close relationship between stories and acceptance testing.

Mike Cohen describes user stories as follows.

> "User stories are short, simple description[s] of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system."

He goes on to describe the typical template developed by Connextra

> As a <type of user>, I want <some goal> so that <some reason>

Cohen's description is the generally accepted definition of a user story and is so commonly discussed in terms of the Connextra structure that the two have become almost synonymous. In practice however, teams tend to settle on their own style of writing stories loosely based on the combination. Sometimes a short description is not enough. Some teams stick to the template whilst others expand it to write requirements out long hand. Some teams abandon it completely and just write abbreviated notes.

Settling on the appropriate scope for a story is also something teams can struggle with. Stories should be short enough to be achievable but still provide some level of business value. Stories help set the rhythm of development and help orientate the team. However, it's easy to get confused by the difference between tasks and stories. It is useful to capture discrete tasks, things like "pay the suppliers" or "talk to Bob in Commodities about their new API", but if these don't add business value, chances are they're not really stories.

The reason this matters is because "business value" is supposed to enable an opportunity for profit. If we're not clear about the definition of the term story, it's easy to create and focus development on tasks which don't add value and so don't contribute towards profit. You can think of the idea of "business value" as simply "cash" or "profit". That way, you can ask yourself "would this story contribute to the bottom line?". To get the most out of acceptance testing, a link from story to acceptance criteria needs to be established. That way, you can demonstrate when real value has been added to the system and elaborate on the details of exactly how.

To capture and track progress, some teams write stories on index cards, others write tasks or work items on post-its. Others still write up analysis in their issue tracking software or wiki.

Tasks vs stories

It's easy to get confused with the difference between tasks and stories. The team lead role can suffer from this especially but keeping track of todo items needn't muddy the waters when it comes to planning story delivery.

It's important to realise that tasks or todo items fit more appropriately under project or team management activities and not story planning. That way, it's easier to pick the right tool to manage them. David Allen's Getting Things Done is a great way to manage your todo pile.

Bringing it all together

Why is this discussion important? How we interpret the definition of user stories has a knock on effect on how we choose to implement our acceptance testing approach. Sticking to the letter of Cohen's story definition above can lead to ambiguous requirements. We need to think a little harder. His definition should encourage us to think about requirements from the customer's perspective, clearly articulate the goal and solidify *why* it's important (the *so that* clause).

It's also important to recognise that the Connextra template is not a literal mantra. Articulating the goal will likely take more than a single sentence on an index card. That's where defining unambiguous acceptance criteria comes in.

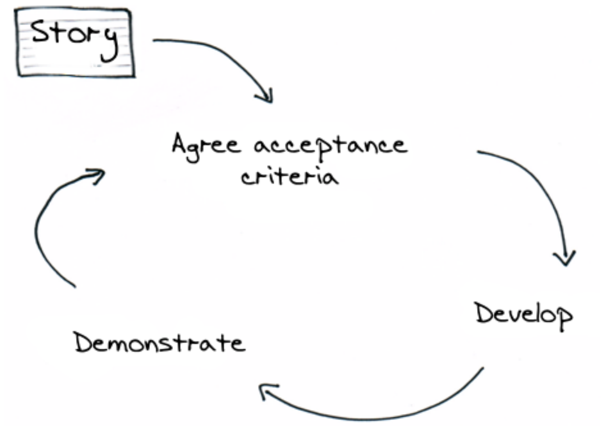
If the story definition is vague, it's difficult to define concise acceptance criteria. Without clear acceptance criteria, it's difficult to be confident about what we're supposed to develop. Without understanding what we're supposed to develop, it's difficult to know when we've actually built it.

Working from story definition through defining acceptance criteria to delivery is something David Peterson calls the Story delivery lifecycle. It brings together the ideas of stories, acceptance criteria and tests with a framework for iterative development that underpins common agile processes. We'll take a closer look at it next.

Typical process overview

The story delivery lifecycle

A typical agile process used by many teams today revolves around the following steps often referred to as the *story delivery lifecycle*. The steps act as a guide to ensuring you're working on incremental value, to highligh problems or misunderstanding early and give you the chance to adjust.



Story delivery lifecycle
1. Pick a story
2. Agree acceptance criteria
3. Develop functionality
4. Demonstrate and sign off
6. Repeat

Let's go through these steps in more detail.

Pick a story

Picking the next story to play should be as simple as taking the next highest priority story from the list of options. Creating the option list or backlog is a little more interesting. Ideally, there should be ongoing work to identify concepts that, if realised, would help achieve business goals (cash). In the corporate environment, it's typically the responsibility of the business analysts to come up with candidate features for a project.

What's a Story?

There is often some debate about the definition of the term "story". For the purpose of this discussion, lets

assume that a story is just a way to decompose the requirements into achievable chunks that, if implemented, would add *business value*.

It's common to physically write the story description on an index card. Teams might then use this as a token on the team's project board to visualise it's life, moving from left to right to indicate progress.

It would usually fall to the iteration planning of a Scrum process to move a set of stories from the backlog to the planned iteration. The team would then attempt to deliver these stories when starting the iteration. In a Kanban process, the backlog becomes the pool of candidate stories that are drawn from at any given time; it becomes the input queue for subsequent activities (such as agreeing acceptance criteria). In both cases, it helps if when picking up a story to work on, there is a good understanding of the business objective it realises. In other words, what's the *real value* this story would deliver.

The next step is to express this value in the form of acceptance criteria.

Agree acceptance criteria

Inputs and outputs

Inputs:

- Story
- Additional conversations, analysis and information

Outputs:

- List of examples and scenarios with expectations (the acceptance criteria)
- Agreement from everyone that these demonstrate desired functionality
- Additional documentation or context in whatever form is appropriate

Avoid:

- Implementation details

It may be that the story you pick up lacks sufficient detail to answer the question "how do we know it's done?". To figure this out, you define the acceptance criteria and in doing so, better understand what's really needed. You might explore example scenarios, edge cases and outcomes to help. The idea here is to describe the requirements not in terms of a series of instructions to follow (a traditional test script) but as an english description of the business goals. When you can prove these have been met, you know the story is done.

When you describe the high level business scenarios like this, you implicitly create a specification accessible to business and technical staff. You're not concerned with the details of how things will be implemented. It's a chance to focus on the business intent and make sure everyone involved understands what's expected, the terminology and the business context.

This is best done with the business or customer. Get together and make sure everyone has a common understanding of what's needed. Expect lots of discussion about the specifics and come up with concrete examples and scenarios. What questions, if answered, will convince the customer that the system is behaving as they specified?

Do we need a meeting to agree acceptance criteria?

No. You don't need a meeting to define acceptance criteria, in fact, its great if you can keep discussing and clarifying them as you need to. You may find it useful however, to have a kick-off meeting where you can discuss a story's background, context and goals.

Note that I'm not talking about an *iteration planning* meeting here, more like a story definition meeting. The difference is that we're not trying to *plan* which stories are being played in an iteration. Instead we want to work up the team's understanding of a story. To be most efficient, some analysis work should have already been undertaken.

Once you've written the criteria down, the next step is to formally agree them with interested parties. We'll brush over how best to physically record the criteria but aim to have them in a format that will help you later when it comes to converting them into executable tests. You might record these on a Wiki, HTML pages stored with the source code or just on the back of the story card. The point is that after this step, everyone involved has agreed that the list of criteria is a reasonable effort at documenting the intent. It's not set in stone but it captures the current understanding.

Remember that all this is done before writing any production code.

Tick off these items as you go through the Agree acceptance criteria phase.

- Business analysis has been undertaken
- Developers understand the business background, context and goals for a story
- There is no ambiguity about business terms and everyone agrees on their definition
- Acceptance criteria have been discussed and documented
- Agreement has been reached on the acceptance criteria

Develop

Inputs and outputs

Inputs:

- Story and context
- Agreed acceptance criteria

Outputs:

- Implemented (and potentially deployable) story functionality
- Acceptance tests (converted from acceptance criteria)

Avoid:

- Implementing anything unrelated to the story

As well as implementing the underlying features, the developers should be converting acceptance criteria into runnable tests during this phase. It may be that tests are written early in the development phase, before any real work has gone on and they'll continue to fail until the story is completed. Or, it may be that the majority of development is undertaken before work on implementing the acceptance tests start.

Which approach you choose has interesting affects on developer testing. For example, if the coarse grained acceptance test is left until the end, you might focus on unit tests and use TDD to drive out design. TDD in this sense is a tool to aid *design*. When it comes to implementing the acceptance test, there'll be very little left unknown. It can be a bit like test-confirm where you back fill the details to get a green test.

In contrast, when developers start with failing acceptance tests, the focus shifts. Acceptance tests are used to drive out coarse grained behaviour like unit tests drive out low level design choices. The acceptance tests themselves may change more frequently with more discussion with the customer taking place. The emphasis is on requirements (the story) and in this sense, the tests become more of a tool to help build *requirements*. This ATDD approach puts acceptance tests in the position that TDD puts unit tests in; at the beginning.

There's an argument in claiming that unit tests may not even be needed with sufficient acceptance tests in place. The meaning of "sufficient" here is open for debate. If the edge cases that unit tests would have picked up are unlikely to materialise and if the cost of fixing these once in production is low, then why test upfront? Especially, if this upfront cost affects the time to delivery (and by extension your profits). To put the argument in the extreme, you could say that you're not really doing ATDD if you're still writing unit tests.

Unit tests give confidence at a low level but if the overall system behaviour doesn't deliver the business proposition, you still won't be making any money. The right answer to the wrong question is still the wrong answer. Acceptance tests should demonstrate the business proposition and so have monetary value whereas unit tests have design value.

Your team's experiences and preferences will influence which approach you choose. Sometimes, the two compliment each other, other times they get in each others way and duplicate effort. Judicious testing takes care and practice.

Demonstrate

This step is about proving to customers that their requirements have been realised. It's about giving them confidence. You can do this in whatever way works best for your team. A common approach is to run through a manual demo and walk through the passing acceptance tests huddled around a desk.

After the demo, if everyone agrees the implementation does what's expected, the story can be marked as done and you can go round the cycle again with the next story. If there is disagreement or if new issues come up, it's ok to go round the loop again with the same story. Agree, develop, demo and deliver.

If you discover *incremental* improvements could be made, you might create a new story and go round the loop with that. This is different from going through the cycle again with the same story which would be more *iterative*. Think of it like incrementally adding value rather than iteratively building value piece by piece before finally releasing. It's like tweaking an already selling product in order to sell more (incremental improvement) as opposed to building a product out until the point it can start to sell (iterating).

A note on manual testing

If you're lucky, there's plenty of people on hand willing to perform some exploratory testing. This doesn't have to wait until the end of an iteration, it can start as soon as a story is stable enough to test. Usually this would be when the story is finished and acceptance tests are passing. It's useful to have a stable deployment environment for this.

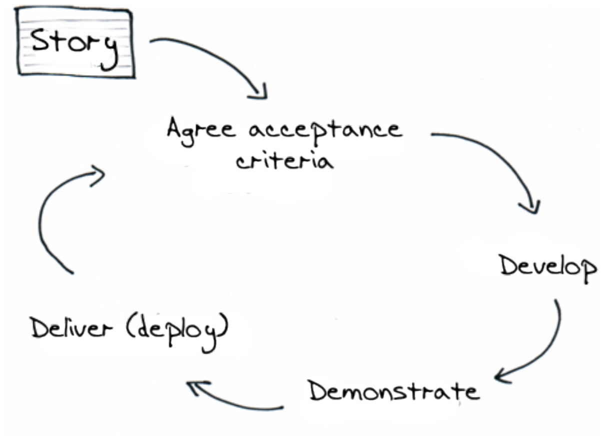
Acceptance testing doesn't negate the need for manual, exploratory style testing. Lisa Crispin calls this kind of testing critiquing the product. Some critiquing can be achieved using acceptance testing whilst more may require a manual approach or specialist tools. We'll look more at this later when we talk about Brian Marick's testing matrix and Crispin's elaboration.

To some degree, acceptance test suites address the need for regression testing. That is to say that derivation of behaviour hasn't been introduced if the acceptance tests still pass.

Deliver

When a story is finished, you may go round the loop again with a new story or choose to deliver the functionality directly to your production environment. This often gets less emphasis because it usually happens when multiple stories are batched up and deployed together, for example, at the end of an iteration. It's actually a crucial step though as it is only after this point that potential story value can be realised.

It can be incorporated into the story delivery lifecycle when continuous delivery ideas are applied with the aim to deploy individual stories as they're ready. It's a fairly sophisticated position to take and requires careful crafting of stories so that they add demonstrable value. It also implies a well groomed and automated build and deployment procedure.



The expanded story delivery lifecycle