# Low-Tech ATDD

**Kevin Klinemeier Davisbase, LLC.**
**kklinemeier@gmail.com**

## Abstract

"The customer does not like what we made this week" is a bigger problem than "The customer does not like what we made this morning". But the natural inclination of programmers is to program until they are out of one of the following: time, features, or food & water. Much of the Acceptance Test Driven Development (ATDD) literature talks about tools for automation, for example Cucumber, Robot Framework, or RSpec. These tools can be a means to an end, but are not required and may be more of a hindrance than help especially when a team is just starting ATDD.

Instead of tools, focus on planning and task creation. Turn the planning of work items (stories) upside down by asking "where is the risk?" rather than "what needs to be built?" A set of standard starting questions that borrow from the traditional QA skillset of Risk Analysis gets the conversation started. These answers lead the planning of the work, starting from the areas with the most risk. Rather than build the feature in its entirety, programmers build just enough to test, based on this plan. No tools are needed, but sticky notes help.

In our team, identifying these testable subsets of work in advance allowed the testers to receive work every day, shortening the bug's life cycle and reducing the cost of their fix. It also allowed the team as a whole to demonstrate to the business approximately every other day, with similar benefits. When the team then adopted automated testing, this experience helped them identify where it would be most beneficial. This experience also translated to better estimates on the part of the testing work, as the habit of discussing risk and mitigation was carried to those activities.

This low-tech ATDD approach is especially applicable to two kinds of transitions a team may be in. The first are those who are taking the first steps to moving away from a big-handoff approach and toward a continuous testing approach. The second are teams that are attempting to work more closely with their business stakeholders. Generally, this approach is useable for both legacy projects as well as new-development. It is appropriate in both the Scrum and Lean paradigms, making no assumptions about when the planning takes place.

## *Biography*

*Kevin Klinemeier has almost 20 years of software development experience as developer, team lead, software architect, and agile technical consultant. As a consultant, he has used this approach with teams as small as three to as large as twenty. He has worked in industries including telecommunications, global logistics, healthcare, and finance.*

# 1 Introduction - IKIWISI and being Driven

The most exciting aspect of the creation of software is also its most difficult: Every time we set out to write software, we are solving a problem that's never been solved before. Because if there's a solution out there that already works, we will download it, copy it, license it, and use it.

Traditionally, we ask our customer "what do you want?" This may have different levels of formality, from simple descriptions, whiteboard mock-ups, or formal documents. All these paths lead to the same universal experience. When shown the result, our customer says, "Ah, now that I see it..." A common complaint among development team members is, "Our customer doesn't know what they want, but they know what they don't want when they see it."

It is tempting to respond to this uncertainty with even more detailed up-front analysis in an attempt to lock down the specification. However, it is not analysis that will help the customer discover what they really need, it is experience using the real, working software. They are essentially saying, because they are business or problem experts rather than software experts, I don't know what I want, but "I'll Know It When I See It" (IKIWISI)

Often this experience can be expressed as tests. If features are too uncertain to drive our work, then it can be tests. Tests now are the primary focus of planning sessions and status meetings. In our planning sessions we discuss which programming activities are necessary to complete a test, and in our status meetings ask, "Which tests have been completed?"

# 2 User Stories and Acceptance Criteria

Agile approaches to the problem of requirements with user stories and acceptance tests (Mike Cohn 2004 loc 378, 560, 1686). Instead of asking non-experts to describe the end state in detail, it asks them to describe what it is the user needs to do in customer terms. Often this takes the form of a story statement:

> As a <role>
> I want to <cause some effect>
> So that <some value to the user or business>

An example:

> **As an instructor,**
> **I want to post my slides online**
> **So that I do not have to email them to my students.**

Acceptance criteria build on this story statement to describe outcomes that are important to the customer, or potentially unexpected to the implementer. These, too, are written from the customer's point of view but are broader. Continuing the instructor example:

1. Uploading the slides should be a one-step process.
2. Only the students signed up for my class should have access to the slides.
3. Only the instructor should be able to post slides.
4. Only PowerPoint files (.ppt) should be accepted, in order to prevent me from uploading the wrong file.

# 3 The Goal: Feedback at the Right Time

If we cannot predict what we want the software to do, our best course of action is to make the software easy to change. There are volumes on approaches and activities to do this from a software design point of view, such as Object Oriented practices including Interface Segregation and Dependency Injection

(Robert C Martin 2009), methodology steps like Continuous Integration (Paul M. Duvall, Steve Matyas, Andrew Glover 2007), and of course the creation of small unit level tests in any language you might choose.

From a quality assurance process point of view, this is accomplished by providing feedback at exactly the right time. That right time is immediately after the programmers have made the change (due to bug or requirement change) but before they have built anything on top of it or forgotten how the work was done.

When we find changes, we want to avoid hearing, “That is an architectural change” or “What idiot wrote this code, and what liar put my name on it?” Instead, we want to hear “I can change that, I just wrote it.” or “We can change that, we have not started on it yet.”

# 4 Plan Differently: *ATDD* starts with *ATDP*

The pressure is on the Quality Assurance organization to demonstrate early and provide that feedback. But small demonstrable pieces of work are not an artifact of a QA plan alone. The *writing* of software must be planned differently. The natural inclination of developers is to build everything that is requested, assuming that “the spec” is correct and that they have not made mistakes.

Therefore, if you want Acceptance Test Driven Development, you must first start with Acceptance Test Driven Planning (ATDP).

ATDP activities occur during the planning phase of agile processes. In Scrum, this is the Sprint Planning ceremony. In Lean, it is likely the planning column on your Kanban board. Also, just as those process recommend, ATDP is best as a collaboration between all the roles involved, especially developers and testers.

Typically in a task creation session without ATDP developers are in the lead. They break the work down into client, service, data, and jargon layers, and commence discussing which flavor of jargon will be best. This has two outcomes: the tests are an afterthought, and any non-programming testers are left out of the process.

Instead, in an ATDP session, teams identify the tests that they will write first, before any development tasks have been created. There is no need to write the entire test in detail, a title is enough. And as the name implies, the user story approach gives you a starting place: the acceptance criteria.

# 5 Acceptance Criteria to Tests

If the acceptance criteria from the story are directly translatable to small tests, great! Count yourself lucky and move on. Most of the time there is some problem in the transition from criteria to test. Good tests are both informative and small.

## 5.1 Make Tests Informative

Acceptance tests’ primary purpose are to allow the product owner to form opinions about the feature being developed.

This means that the functionality under test should be complete enough to include a result or outcome that is at the customer’s level -- for non-technical customers, this will typically be a user interface. Unless a team’s product owner is comfortable reading XML, that outcome would be unacceptable.

Requiring tests to be *informative* requires the implementation to be done in small, vertical slices. Rather than create all of the data structures necessary for the whole feature set, the team creates only the data tables necessary for this acceptance criteria.

This feels inefficient, and programmers are apt to utter the most dangerous words in programming: “While I’m at it, I may as well…” Time spent on code and artifacts not needed to be informative is potentially wasted. If the demonstration causes the product to change, all those unrelated items will likely have to change as well.

> **Not Informative: Test that confirms a file is stored to the database**
> The product owner has no opinions about the database format. Until that data is put into the context in which it will be used, it does not build any new knowledge. Rewrite the test to include the use of some of the information.
>
> **Better solution: Upload the slides and see a confirmation message**
> This test helps the product owner determine if the user’s experience is correct. They may learn about the kind of confirmation message and details they would like to present. Perhaps the confirmation should include a link to the download, or the name of the class the slides were posted to. Should the filename appear on this page? Great questions arise from this test. Note that this test is specifically about a successful upload. Failure messages are a different test.

### 5.2 Keep It Small: One to Two Days

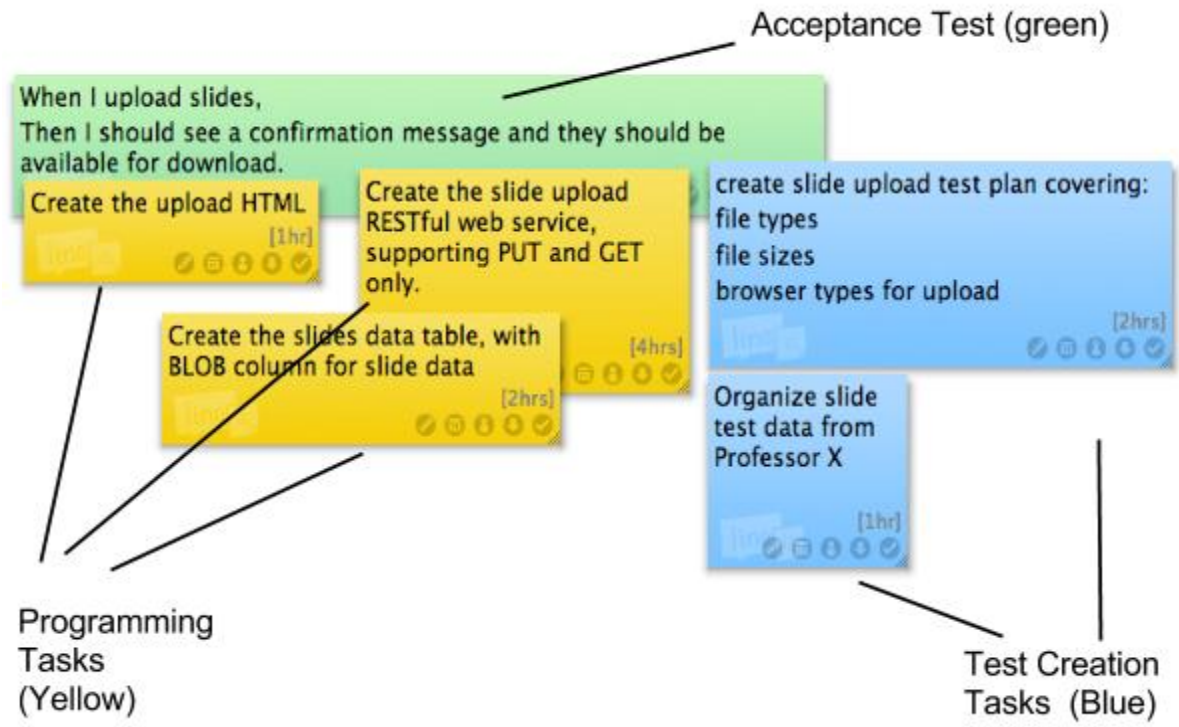
Being able to form an opinion does not imply that the work is completely finished and polished. It should still be rough -- not production ready. The time taken to make it ready for production will move us away from “I just wrote it” and into territory where programmers say “I have no memory of this code.”

> **Not Small:** Instructor can sign up, log in, reset password, and upload slides.
> There are many features of a login process that are expected in a production-ready product: password resets, multiple attempt lockouts, password rules, etc. If this work is large, such as in a brand new system or complicated enterprise environment, it should be divided into additional tests.
>
> **Better Solution:** Start with one case: “Signed-in instructor can upload slides, students and assistants cannot.”
> Registrations can be simplified, and the work around password resets or lockout durations is moved elsewhere, allowing us to focus on one aspect of the work.

## 6 From Tests to Code

Once you have identified one or more tests, write each one on a card (sticky note, virtual task card -- something visual). Then challenge your development teammates to write only the programming and design tasks that are *absolutely required* to enable that test. Place those tasks visually below the test, marking dependencies physically. Add your testing tasks as well, which can be the creation of detailed test plans, or broader tasks such as exploratory testing.

If you use physical cards, you will end up expressing dependencies very naturally. You physically cannot pick up an underlying task without picking up all the other tasks that need doing first, because the dependent tasks are in the way.




Since each task is estimated in hours, we can calculate how long it will be take to go from start to finish on a single test and evaluate whether we have met the criteria of tests being small. Note: the one to two days goal includes the programming *and* the testing time! In the example above, we have planned a total of ten hours of work, which meets our "small" criteria handily.

If the programming size/effort for your test will take longer than the one to two day guideline, collaborate with your development teammates to break it down into small testable pieces. Here are some options:

- Break the data or interface into separate tests.
  Example: User Setup has 25 fields, start with a test for username, first name, last name, and email.
- Build only the immediate results, implement the delayed results as part of the test for their display.
  Example: In the upload test above, do not store the file -- save that work for the test that displays the stored file instead.
- Break out different outcomes by their results
  Example: build the error-checking as a separate test. Initial implementation for success is to display "error: success not implemented"

# 7 Organization and Prioritization

Great, you have a lot of tests, and work to be done for each one. Where to start? The Product Owner may shrug and say, "It all has to get done." If we leave it up to the programmers, they will often choose whichever piece has the most interesting coding toys associated with it.

- Likely to change due to uncertainty in our Product Owner or stakeholders.
- Has regulatory or auditing expectations.
- Uses tools or technology we have never used before.
- Integrates with a technology we know to have bugs.

- Integrates with systems we have not integrated with before.
- Likely to have performance problems.

Features with one or more of these attributes are then high candidates for problems during development. For each of these features identify the acceptance test that will test for the related risk. These risk-related acceptance tests should be prioritized first so that we discover any problems as soon as possible. The earlier a problem is discovered, the easier it is to handle.

### 7.1 This is a different role for QA!

This approach changes the role of quality assurance from gatekeeper to communicator -- always searching for how to provide fast feedback. In ATDD, QA leads the team in asking, “How can I show this to someone and find out if it is what we want?”

Many of these testing tasks are not simply for QA to execute, but for QA to facilitate a conversation or demonstration, which in turn leads to more information. Standup and status meetings are good opportunities to ask these pertinent questions:

- Are we working to enable the highest priority test?
- Why are we not working on this task?
- How many demonstrations have we done?
- What percentage of our features have we demonstrated?
- What high priority tests are remaining?

## 8 Making it fun

This approach is compatible with a kind of gamification, or at least a lighthearted approach to execution. Each acceptance test can be treated like achievements in video games. Doing the minimum possible development work to “unlock that achievement” becomes the metaphor for the daily standup.

One example might be that when the Upload test in our example above is accepted by our product owner, we could put a disk-drive sticker on the user story card, or a shield sticker when the security test is accepted.

## 9 Tracking the plan

During the execution of the sprint, testing and development tasks can be used to track progress in the usual way (burndown charts or hours remaining).

In addition, two major metrics can be visible:

- Burndown of acceptance tests remaining
- Number of days since a new test was unlocked

Let’s look at what each item may be communicating, what good and bad looks like, and how to correct our course.

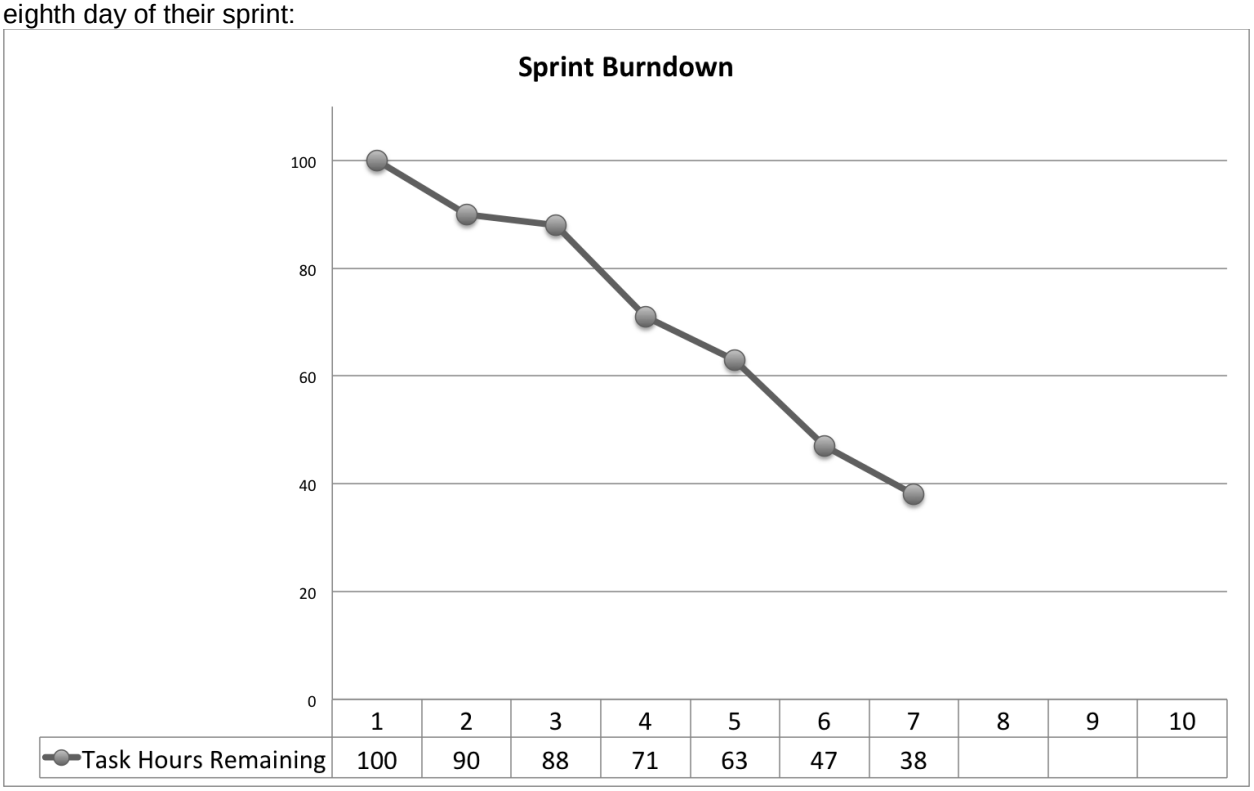
### 9.1 Burndown of acceptance tests remaining

This burndown helps us answer the big question: are we going to finish? Unlike task-level burndowns, simply finishing easy tasks across many different stories does not show as progress. Teams that struggle with story-level burndown charts not being granular enough often find that this measure of sprint completion is very accurate.

This metric, much like a regular burndown, should be updated daily, and reviewed (at least implicitly) as part of the daily standup. Good Scrum Masters and coaches will be sure to point out when our burndown indicates that there is a problem, and translate that into questions for the team during the post-standup "parking lot" session. In this case, the questions are the same as any other burndown projection:

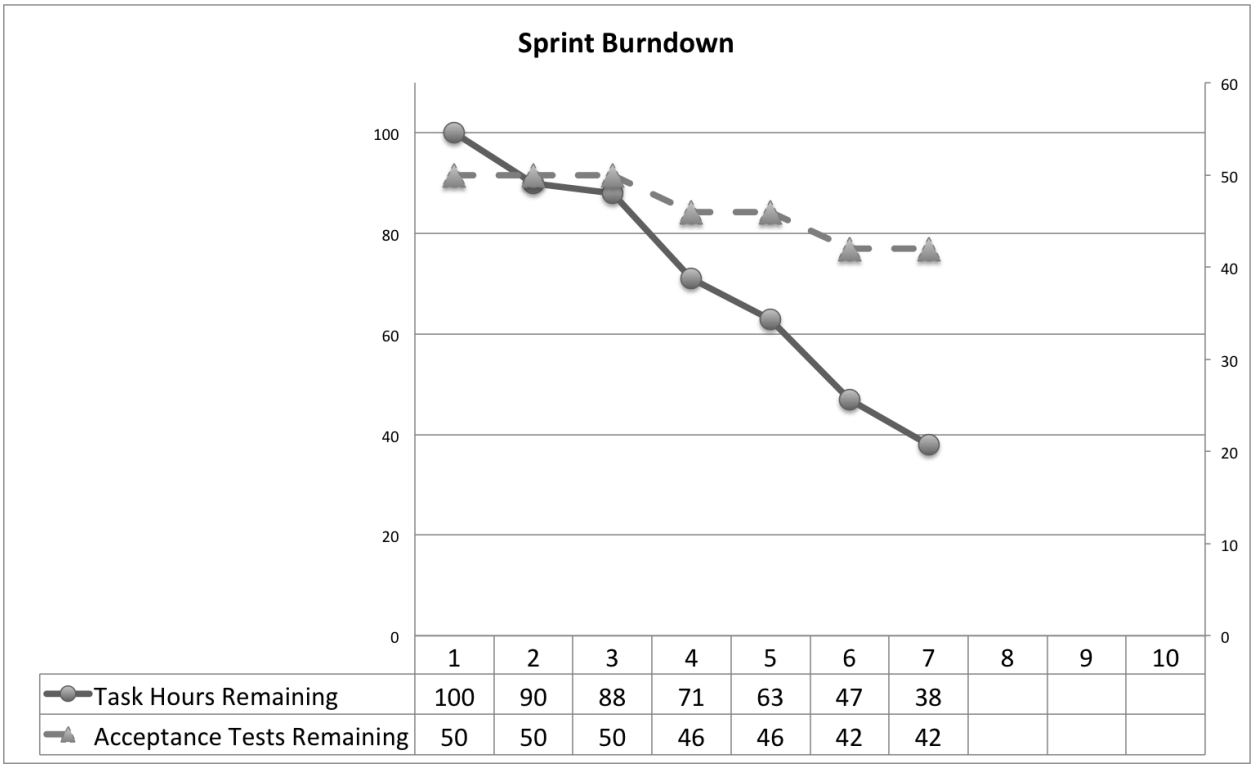
- What are our impediments?
- Can we be sure to finish the most important items?
- Can anything be negotiated out of scope?
- If we do not finish everything, have we reviewed the new scope with our stakeholders?

Let's look at an example. Consider the burndown example below, which represents a team entering the eighth day of their sprint:




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Task Hours Remaining | 100 | 90 | 88 | 71 | 63 | 47 | 38 | | | |

This team looks pretty good -- on track to complete all the tasks. But completing the tasks is not the same as making customers satisfied. Let's look at an example that also tracks Acceptance Tests Remaining.




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Task Hours Remaining | 100 | 90 | 88 | 71 | 63 | 47 | 38 | | | |
| Acceptance Tests Remaining | 50 | 50 | 50 | 46 | 46 | 42 | 42 | | | |

This second chart makes it clear that while this team has completed a lot of work, not much of it has been tested. Perhaps they did not build it in testable pieces or there may be an environmental issue that is keeping them from performing their tests. Whatever the cause the second chart shows that there is a lot of testing that has not been done. It will need to be done in a rush right at the end of the sprint, leaving very little time for fixing bugs or adjusting things when their customer inevitably says, “Now that I see it, I don’t like it.”

If we were tracking this all through the sprint, our Scrum Master (or other lean leader) should really begin highlighting this potential impediment starting on day four. This would allow the team to have a conversation about how to adjust the plan in order to execute some of these tests earlier, getting the feedback earlier when changes are easier to make.

## 9.2 Number of Tests In-Progress

This is a measure of those tests that have no dependencies and can be executed, but have not yet moved into the success state. This metric helps teams discover when the bottleneck may be in the testing effort itself. If tests are not completed but new tests are being enabled, this implies that it is taking longer and longer to provide feedback and fixes.

This metric can be used within the sprint as a conversation trigger. Track this metric daily during the sprint, and whenever the team has more than three tests in progress, ask them:

- What are the impediments for finishing the highest-priority test?
- How can others help to test? (Programmers not excluded!)
- Who is working on unlocking yet another test? What can they do instead?

### 9.3 Number of days since new test unlocked

This metric helps teams transition from a mindset of a big handoff at the end by reminding them that testing should start as early as the second day of the iteration.
If it has not started by the third day, ask:

- What are the impediments to finishing the work for the highest priority test?
- How can we get more help on the tasks for that highest priority test?
- What is working and testable right now?
- Did we discover that the test is too big? How can we break it up?

This metric can be used as a conversation trigger as above, but it is also useful in the retrospective to identify habits and behavior patterns.

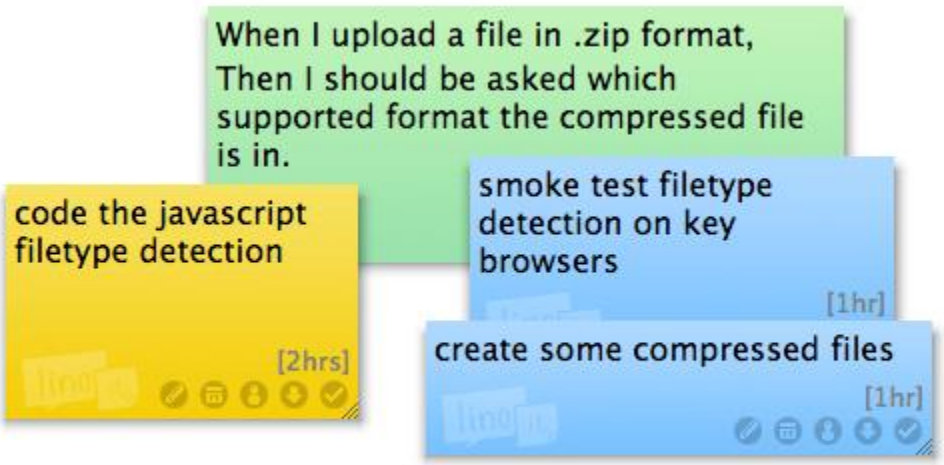
# 10 Changing the plan

Often, the results of a completed acceptance test will be a change to the plan. This change is reflected by changing the acceptance criteria and acceptance tests in the story. That test change is then used by Development to identify what changes are necessary.

Like the initial planning, this is not intended to be a waterfall-style set of handoffs. Teams will find that the acceptance criteria may alter a test so it cannot be completed in a timely fashion. Programmers will propose a modified version of the test. In this way the team optimizes the building of software, the collection of feedback, and the delivery of value.

**Example:**

In our slide sharing user story above, we begin to test the acceptance criteria around uploading only supported file types. We even get to demo the software to a potential user. He uploads a compressed file, in .zip format, and is surprised to see it rejected. The Product Owner asks the team how difficult verifying the contents of the compressed file might be. A conversation about security concerns of expanding files ensues, and the Product Owner accepts a simpler solution that simply asks the user what format (keynote or PowerPoint) compressed files are in. This new acceptance test is added to the board.



The programmers add their coding task (javascript file type detection), and QA adds their detailed testing tasks as well, layering them on top of the main Acceptance Test.

### 10.1 Refactoring

If we change the plan after we have started writing code, then we will necessarily have to change the code. In Waterfall projects, both QA and Business have learned to fear refactoring because changes late in a project are expensive and risky. In ATDD projects, we have addressed the cost and risk by moving the changes from the end of the project to immediately after the code is written. Now our goal is to enable as much refactoring as we can because, when it happens early, it is small in scope and effort.

The practices we have discussed so far support these goals: Write only small pieces of functionality, so that we have little code to change. Execute tests as close in time to code completion as possible, so that we catch faults quickly, and build very little on top of the existing work. This fast feedback approach changes refactoring from fear to necessity.

This can also provide some excellent feedback to teams attempting to take on agile design principles. Want to know if your dependency injection strategy makes code easier to change? Did we segregate the right interfaces? By trying to make changes during the sprint rather than during a maintenance phase, we get fast feedback on the maintainability and extensibility of the design as well.

## 11 What success looks like

One of the most important questions in adopting any kind of process change is, “What does success look like?” An energized, excited team may summarize the myriad benefits of the change into “fixes all problems, and provides running water, Coke, Pepsi, and Spaten in every room.” That is a high bar!

Some post-sprint metrics that may help identify when you are getting the benefits, or guide retrospective conversations on how to change include:

- Number of demos given during a period (higher is better)
- Mean time between starting new tests (lower is better -- we want many small tests).
- Time to first successful acceptance test of a story (lower is better)
- Number of acceptance tests changed during a story (higher is better*)

Counterintuitively, changed tests are generally good. This process captures changes that could only be effectively identified *during* the sprint. When tests change, that means the team is applying what they have learned from testing. This metric also exemplifies the need to have open and honest communication.

For teams that adopt this practice as a way to move away from siloed, big-handoff, waterfall development, this is often a hard behavior to change. The team’s habitual behavior is often to resist change, or be ashamed of it -- help the team learn to celebrate that change, and the work we did to simplify it. Often, the change is so easy to make that it gets ignored. To keep teams on track and motivated, periodically ask, “how would this change have gone in the old system? When would we have caught that before?”

## 12 The reward for good work is...

As the old saying goes, the reward for good work is more work! Once your team is comfortable using acceptance tests to plan and execute the work, then it is natural to introduce ATDD tools such as Cucumber, Fitnesse, RSpec, etc. Your team will be able to use experience rather than speculation in answering some of these questions that teams ask as they evaluate ATDD tools:

- Are the tests written by non-technical people? (Cucumber does this well)
- Do we have lots of tabular data? (Fitnesse does this well)
- Are our features too different for a standard test template (RSpec has more flexibility)
- Are the tests read by non-technical people? (Trick question – that is what an acceptance test is!)

# 13 Conclusion

It is hard to build software for solutions that have never been solved before, and since the cost of copying data is essentially zero, that is what we are usually asked to do! Respond to the "I Know It When I See It" syndrome by demonstrating working software rather than performing detailed analysis.

Demonstrable software does not come from tools, but from communication. Identifying valuable tests from the start, and developing the software in such a way that those tests are completed early enough to get feedback and make changes will result in higher quality and higher productivity.

Once a team is comfortable using tests to organize their communication within the team and to product owners and stakeholders, ATDD tools can be used to further improve feedback, reduce re-work, share learning, and increase productivity.

# References

Cohn, Mike. 2004. User Stories Applied. Addison-Wesley Professional. Kindle Edition.

Martin, Robert C. 2009. Clean Code: A Handbook of Agile Software Craftsmanship. Pearson Education, Inc.

Paul M. Duvall, Steve Matyas, Andrew Glover. 2007. Continuous Integration: Improving Software Quality and Reducing Risk. Pearson Education, Inc.

Carr, Marvin et al. Taxonomy Based Risk Identification. Technical Report CMU/SEI-93-TR-6, ESC-TR-93-183. June 1993 http://www.sei.cmu.edu/reports/93tr006.pdf