Acceptance Test Driven Planning

Richard J. Watt[1], David Leigh-Fellows[2]

[1] Richard J. Watt, ThoughtWorks, UK
rwatt@thoughtworks.com

[2] David Leigh-Fellows, Egg, UK
David.Fellows@Egg.com

Abstract. The experience of writing and maintaining Acceptance Tests for many is not a successful one. Perhaps, like unit tests, their worth is misunderstood. When first introduced to acceptance testing many think of them merely as a form of black box testing and miss their greater significance as a multi-use form of requirements specification. We have found that by making acceptance tests not only central to the definition of a story but central to our process they can be used to drive the entire development. This paper describes an adaptation, or evolution to XP style planning that takes the existing planning practices (with some additions) and organises them in a way which we believe can lead to better planning and more predictable results.

1 Introduction: How Do We Know When We Are Done?

One of the defining questions for a development team is “do the developers know when they are done”. The question is a simple one but too often the answer is not as obvious as it should be. Whatever form the requirements are expressed in, it is important for the development team to be clear on what is to be implemented for them to know “when they are done”. Acceptance Tests (ATs) are an effective way of expressing requirements in a way that provides an unambiguous answer to this question. However, getting acceptance tests written and maintained is often difficult for a team. As we evolved our process over time there would be many bumps in the road to attend to but this seemed to be our biggest problem so we tackled it first.

Acceptance Tests are not only central to the definition of a story but by making them central to the process they can be used to drive the entire development [1].

2 And Then There Were Tests

There are many types of tests in software development projects. For our purposes we shall limit our discussion to unit, functional and acceptance style tests. We introduce these descriptions to make a distinction between these test types and to clarify our interpretation of ATs specifically.

2.1 Unit Tests

Unit Tests test units. In the context of XP style projects and Test-Driven Development (TDD) [2], unit tests serve two purposes:

- Unit Tests provide a means to automate the testing of code we develop at the level of methods, functions and procedures. In themselves they are useful but by assembling these individual tests into a suite of tests, we gain much more.

  As a suite of tests they provide feedback on the effect of our changes to the existing code base, thus helping control and verify the internal quality of the code. Refactoring, one of the most important techniques to emerge in the last decade, would arguably not have been adopted so widely if it were not for the rise of unit testing and the Xunit framework specifically.

- In the context of TDD, they serve to provide us with a framework we can use to direct the design of the systems we develop.

In [1] the authors describe a approach where ATs are used to drive development; and they argue further that the advantages of unit testing can be met, and in some cases surpassed, solely through the application of ATs. By using ATs to drive development we are able to assess the external quality of the system but not the internal quality of the design or code itself. In our experience this carries an unecessary risk and we would argue that the short (design) and long (maintenance) term benefits of unit testing outweigh their cost.

2.2 Functional Tests

Functional Tests[1] fill the middle ground between Unit Tests and ATs. They are often used by developers to verify that their unit-tested components work together as expected. Unlike unit tests where the objects under test commonly have the objects on which they depend mocked out, functional tests interact with the actual objects. Functional tests are more extensive in their coverage and usually involve some kind of end-to-end scenario.

2.3 Acceptance Tests

Acceptance Tests specify the acceptance criteria for the story. If the story is the promise of a conversation then agreeing the ATs mark its conclusion.

It is during the course of this conversation that questions will be asked, the customer requirement will evolve and ultimately a shared understanding of the requirement will

---

[1] Although originally the name given to what we now call acceptance tests [3], their meaning is now more commonly taken to be as described.

be reached which will shape what the developers implement for that story. Unlike unit or functional tests, the customer writes ATs.

It is common for these tests to be written during the course of the iteration, perhaps finalised midway through the iteration [3:1]. A disadvantage to this approach is the potential for developers to develop the wrong functionality in the interim period when the requirement is still not fully clear. This can result in subsequent rework to bring the code back in line once the requirement has been clarified.

In Planning Extreme Programming [5], Kent Beck and Martin Fowler remark on the idea of getting the customer to write the ATs in preparation for the iteration planning session, but go on to say that “it seems to be impossible to get the acceptance tests at the beginning of the iteration”. Brian Marick used the term “Coaching Tests” [6] to describe this scenario, although the name does not seem entirely appropriate. Alternatively, Joshua Kerievsky, has chosen to call ATs “Customer Tests” [7] but their meaning is essentially the same either way – only the name has changed.

The idea and practice of writing ATs definitions before the coding even begins has both obvious and subtle benefits that we will discuss in the next section. What are the obstacles to writing ATs before the iteration planning begins and how can they be overcome? We have sought to find answers to these questions in the last year and in doing so have developed an adaptation of iteration planning that has some key benefits over the traditional approach.

3. Why Acceptance Testing Is So Important

If a story serves as a description of the requirement then the ATs form the ultimate definition of the requirement. When we describe the acceptance criteria for a story we are defining the bounds of that story, helping our developers understand what to build and helping them know when they’re done. The ATs define the interface contract between what the customer wants and what the developers must implement.

ATs also provide us with a means to define acceptable external quality [8]. For example, in my application, if I click on a submit button and somewhere in the depths of the application a call is made to an unavailable sub-system, does the application go bang and display a generic error, does it elegantly attempt to recover and retry, or perhaps give me the offer of some alternate service? There are different costs associated with each option; each will result in a different user experience; and each will effect the user’s perception of external quality.

We rely on our automated unit tests to tell us if we have broken anything, and the same is true for ATs. If unit tests, as a side effect, provide a means to monitor and maintain internal quality then ATs provide the same function for external quality. Both are equally important but the external quality is what the customer sees and experiences, and therefore values most.

And how should we run our tests? We could run our ATs manually every time we add or change some functionality. Unfortunately this tends to be both resource and time intensive and often results in tests not being run as frequently as they should. To enjoy the same benefits we take for granted when writing unit tests, we should automate our ATs too.

4. Why Acceptance Testing Is Hard.

The case for using ATs would seem to be a strong one and yet their adoption is far from universal. Why is this so?

Perhaps, like unit tests, their worth is misunderstood. When first introduced to TDD many developers perceive unit testing as purely a testing technique and miss its greater benefit as a design technique. The same can be said for acceptance testing. When first introduced to acceptance testing many think of them merely as a form of black box testing and miss their greater significance as a multi-use form of requirements specification.

Ultimately, there are many reasons why ATs are not always written and maintained and each of these constraints/issues must be addressed for acceptance testing to be successful:

- Customer participation: Unlike unit tests, which are written and maintained by the developers, the customer writes the ATs (although often with the help of development and/or QA) [3:2]. For those customers used to more traditional forms of development the demands of an agile project can initially be difficult and the responsibilities unclear. However the benefits are soon apparent. On one of our recent projects, our customers commented: "they're [ATs] hard to write at first but helped me to feel comfortable about what I needed and what would be delivered" [9].

- Management buy-in: Quite often the reason the assigned customer finds the role a strain is because their duties are myriad and the project is just one of their many responsibilities. Management must be aware and supportive of the investment required to perform the role of a customer on an XP style project.

- QA availability: The purpose and process of writing unit tests is quite different from writing ATs and requires a different set of skills. QA people have the necessary skills to help the customer articulate the acceptance criteria for a story and their contribution is essential to the overall process.

- Our developers are already writing unit tests and the introduction of ATs means even more tests. "We've already got all these units tests to do and now we've got these ATs to implement as well" (sic) - nobody said it was going to be easy. For every useful test we write there is a benefit, and equally for every test we keep there is a maintenance cost. Our code hygiene

habits must be applied consistently to both our production and our test code if we are to remain agile.

In our experience we have found that once introduced to the benefits of TDD few developers abandon such a self-rewarding practice. The benefits of ATs are often not initially as compelling to developers as they are for unit testing. However, their value is soon acknowledged as the likelihood of misunderstanding the requirement is lessoned and thus rework avoided.

- ATs have traditionally been difficult to automate due to limited framework support. A framework helps reduce the cost of ownership by reducing the effort it takes to write and maintain tests. Of course, things have changed with the advent of functional testing frameworks like FIT [10] and the more recent FAT [11], but their popularity is still some way off that of the ubiquitous Xunit unit testing framework.

5. Now To Planning

Our experience of iteration planning was perhaps typical. As fully signed up members to the agile development cause, we all agreed that this form of planning was better than what we did before, and yet in our shared experience the process was never wholly successful and not without some pain. Maybe we were doing something wrong but the more people we spoke to the more we realised our experience of planning was at least not uncommon.

As an illustration of our plight, below is a description of a typical iteration planning session drawn from our shared experience:

All interested parties gather into a room which for us meant ‘n’ developers (where ‘n’ <= 12), our customer, our QA, often an interaction designer, our project manager, iteration manager and coach – all empowered to contribute to the planning process.

Our customer would describe each of the stories in turn. The developers would ask questions and discuss alternate solutions before producing a list of tasks for each story and an updated estimate figure. This process of discussion could take a long time and was once described by a customer as feeling more like a “techno babble” session than a planning one. From a developer perspective the customer could not provide the information they needed in order to produce confident estimates in the meeting. This resulted in best-guess estimates on the incomplete information. Our customer thought they had all the information to hand before they went into the session but even with their best efforts they couldn’t anticipate all the questions that were going to be asked.

What seemed like a good idea to start with did not feel so good 8 hours later as the team emerged tired and not so sure about this whole “empowerment” thing.

The real pain came when we measured our velocity at the end of the iteration and discovered the discrepancy between what we had signed up for and what we had

achieved. There are, of course, many reasons why this could have happened but once we had investigated underlying the causes we discovered that the main problem was that we hadn't identified a complete set of tasks for each story. Next time there was much less enthusiasm about iteration planning. We could all see that the developers were working really hard and the results produced were good but in hindsight it was if they had started a 400m race but didn't know where the finishing line was and just kept going round the track. Our understanding of the work was incomplete which meant it was unclear when we were done, which meant everyone was very tired and nobody was very happy. So we knew we had to do something.

6. Getting Our Stories Straight

One of the biggest problems we had in our planning sessions was that our customer felt that no matter what they did to prepare there were still many questions they could not answer without time for further analysis or/and investigation. Our solution was simple: a day or two before the end of the iteration the customer would sit down with the QA and a developer to begin the process of writing ATs for the upcoming stories, or in our terms, "Getting our Stories Straight". In short, we still ended up asking the same questions but by doing this activity before the planning session we were able to find more answers earlier and thus be better prepared for when we did gather the full team into a room. This is not a return to "big up front analysis" but an acknowledgement that as we approach the end of an iteration we already have a good idea of what is going to be completed in this iteration and the customer has a good idea of what stories they want to include in the next.

Our principal aim at this stage was simply to better prepare the customer for our planning session so that developers could get more of the answers they needed at the time they needed them, but we soon discovered there were many more benefits. In particular we have identified the following benefits:

- **Are our tests any good?** The QA helps the customer make sure the acceptance criteria specify the expected behavior and external quality. The developer is on hand to make sure that there is no technical reason why the functionality required to pass the ATs cannot be implemented. If we liken this to UML use cases, a story becomes the title of the use case and the ATs become the use case itself detailing the main success scenario, alternate and error scenarios. [12].

- **Are our stories too big?** One of the positive side effects of this process is that it often highlights if a story is likely to be too big for the development team to estimate before we even get to the planning session. The simple correlation between the size and number of ATs and the size of the story has proven to be a reliable predictive indicator.

- **The chance to improve our estimates**: For a story that appears too large we may choose at this stage to break it into two or more smaller stories. In our

experience, smaller stories are easier to estimate accurately. We strive to have stories that can be ideally completed in 1 to 3 days.

- **Customer requirements vs. QA requirements**: The testing interests of the QA may go beyond that of what is needed to define the requirements for the story, but over time we have found that the difference is much smaller than we expected.

The last point is an important one. Using ATs as our "single source of truth" means the needs of both the customer and the QA function are served by a single artefact. We would later see how the same ATs could be used to improve our task breakdown and estimates. The key to these additional benefits was our decision to choose ATs as our focal point for all planning and development activities. With our stories "straight" the next thing to improve was how we ran our planning sessions.

7. The Planning Workshop

In Acceptance Test Driven Development the typical XP planning session is refactored into something we call the Planning Workshop. The objectives remain the same as the XP planning session (updates to our story estimates with a clearer picture of their dependencies), but we find the quality of the information provided is much improved.

The Planning Workshop is split into two phases, "Task Identification" and "Present and Challenge".

7.1 Task Identification.

We have a set of candidate stories for the next iteration, their ATs, some existing high level estimates and a vague understanding of what dependencies exist between stories.

The Planning Game Workshop begins with the customer presenting the candidate stories to the rest of the team. Each candidate story is described along with its ATs.

The developers are then grouped into triples and each mini team signs up to a subset of the stories for further analysis. We organise ourselves into triples because we've found that 3 is the optimum number of people to have in a mini team; we find that 3 people are able to fit around a pair station and the fact that we have an odd number always results in a mediator in case of disagreement.

Once each candidate story has a triple assigned we move into the phase designed to drive out the tasks required to pass the ATs. Each triple heads back to a workstation and pulls up the detailed ATs for their candidate stories. The customer and QA float between the triples answering any questions that may arise to make sure that everyone

is clear on what is required in the story. At this stage the existing ATs may be altered or (more rarely) added to.

The ATs provide a framework to help the developers think about what needs to be done; the developers consider each statement in turn and ask what functionality must be implemented to satisfy each statement; the answers to these questions become our tasks. It is useful and illustrative to consider what we did before and the reasons that led us to this process of task identification. In previous iterations, it became obvious to us that there was a discrepancy between the tasks identified for a story and the actual work required to complete a story. It was common to find a developer saying that they had finished the tasks for a story and now only had the ATs to implement. The problem was that that it took the same time to implement the ATs as it had taken them to complete the tasks. Their original task breakdown, with all the best intentions, was incomplete. We quickly found that by using the ATs as the focus of the task identification exercise we were able to produce a list of tasks which was much more representative of the work that we would need to do to complete a given story. With the increase in confidence we gained from a more structured approach to task identification, we also found our estimates improved too.

7.2 Present and Challenge.

The next phase of our planning workshop is called “Present and Challenge”. The idea is that each triple presents their solution for a given story back to the rest of the team, and the rest of the team are then given the opportunity to challenge the list of tasks to see if we can refine the task list and the estimates given. This works well on two levels as it is a useful tool for exploring (at a high level) the proposed solution with the whole team present, whilst also acting as a double-check ensure we have considered all of the tasks.

In addition, this process gives the team members an opportunity to practice their presentation and debating skills. We have also found it can be useful in establishing team rapport – the sessions are designed to be challenging and we manage them carefully to avoid challenge turning into conflict. Finally, the process is time-boxed and takes 2-3 hours as compared to our previous and much less effective 6-8 hour marathon session.

8. Putting It All Together

We have spoken specifically about “Getting Our Stories Straight” and “The Planning Workshop”, but where do these activities fit into the larger picture?

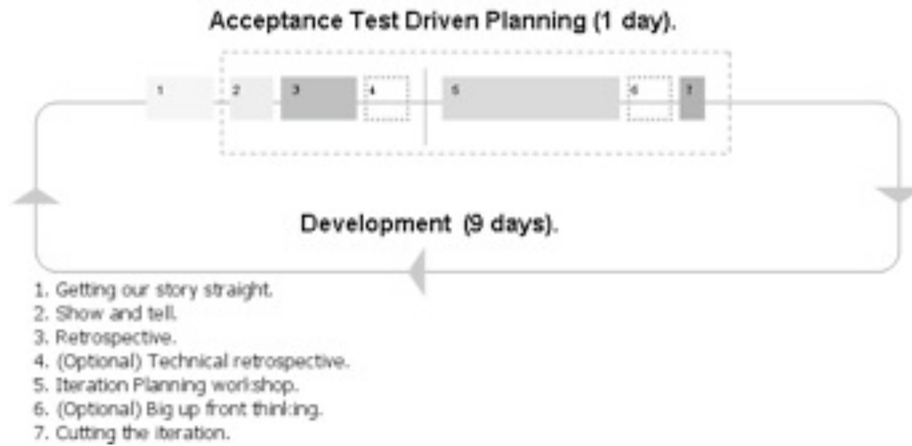

Figure 1. ATDD Timeline

In a two week iteration we are now able to limit the impact of planning activities down to a single day for nearly all the developers.

The morning marks the end of the existing iteration and begins with a show-and-tell session to allow interested (and paying) parties to come along and see what the team has achieved in the iteration. There are usually few surprises here as the customer and QA will have seen each story as it was completed during the iteration, but it is a useful opportunity for the team to appreciate all that has been achieved.

We then follow this with a short retrospective [13] where we discuss those things that went well, not so well, things that still puzzle us and any actions we need to take to improve.

An optional addition to the morning programme is a technical retrospective which we have used on occasion when there have been significant changes to the code base which not all members of the team have been directly involved in. This of course is a "smell" of sorts but, regardless, is often a useful way to get everyone back on the same page before we begin the next iteration.

A break for lunch, a new iteration begins and we are in the "Planning Workshop" for most of the afternoon.

A "Big Up Front Thinking" session at the end of the day is also optional but is sometimes used by the team to draw broad brush impressions (class and simple interaction diagrams mostly) of how the code might evolve for the set of stories we are just about to begin working on. Big up front design? Not really. It is just a way to share ideas and get clear in people's minds current ideas on where we might go next.

The day is over and it is time to go home and look forward to another iteration.

9. Conclusion

Kent Beck has always been keen to stress that none of the guidelines or practices are new in themselves but that the organisation and order of their application is different. When practiced together they produce a benefit greater than the sum of their parts. The 12 plus[2] practices can broadly be seen to fall into either one of two camps: planning or development type activities. We would argue that development is emphasized over planning yet in practice successful planning is more difficult to achieve.

The familiar mantra of test-code-refactor, combined with continuous integration and source control has given us a development environment with all the right ingredients and a fairly prescriptive recipe on how they should be practiced. From a developer standpoint these practices are self-serving and self-satisfying: we do them because they make our lives easier, more productive and fun. We continue to refactor not because we are told to but because we know that if we don't keep the code base clean we (or some other poor soul) will have to pay the cumulative price of working with our code debt. We write tests first because they drive out the functionality of the application. As a bonus they provide us with a suite of tests that signal when a change results in breaking something else in the existing application.

The planning activities, which make up the other half of XP, are much less prescriptive in how they should be applied. XP style planning, especially for those used to more traditional heavyweight processes, can initially appear almost too simple to work. Ultimately planning is about capturing requirements in some shape or form, prioritising which are most important, estimating the effort it will take to implement those requirements and delivering demonstrable business value on a regular basis. If only it was always this simple: The customer does not always know what they want when we need them to express their requirement: the requirement is not always clear to the developers; the developers are not clear when they are done; and task breakdown is incomplete, resulting in volatile velocity.

This paper describes an adaptation, or evolution to XP style planning which takes the existing planning practices (with some additions) and organises them in a way which we believe can lead to better planning and more predictable results. In developing the process, we have aimed, at every juncture, to find the least amount of work we could do and still make informed, quality decisions. Experience has taught us that this balance point is not easily found.

Over time we have recognised that Acceptance Test Driven Planning appears to have a 'Goldilocks' like quality – any more process and we feel the overhead, and any less and the process begins to fail. We do, of course, continue to look for ways in which we can improve the process but after many months and many projects, the balance we have found by planning and driving the development in this simple way feels "just right".

---

[2] For example, retrospectives are a commonplace practice on many XP style projects.

Biographies of the authors

Richard Watt: Richard is a coach and developer working with ThoughtWorks in the UK. In the last three years Richard has spent a large part of his time coaching and mentoring teams in XP and other Agile development methods. He has been a practitioner of XP since late '99 and can't remember how he developed software before then but is sure the results weren't as good.

David Leigh-Fellows: David Leigh-Fellows is a software architect and iteration manager. He has run half a dozen XP software projects since he came across Kent Beck three years ago. Dave has 15 years experience of software development and is a test driven everything convert.

Key Phrases

Extreme programming, acceptance testing, planning, unit testing.

Acknowledgements

To Owen Rogers and Ivan Moore for their contribution to the early development of some of the ideas presented in the paper; for being smart; and being great fun to work with.

To Alan Francis, Areiel Wolanow, and Ally Stokes for their constructive feedback in the earlier drafts; to Rachel Davies, Alex Howson, and Stuart Blair for their time and support in helping get the paper into a fit and proper state as the deadline loomed.

To Mary Poppendieck for her valued feedback and the ongoing support of our ideas.

And finally, to my colleague Rebecca Parsons for her hard work and encouragement in helping us make sure our work was best represented.

Bibliography

1. Roy Miller, Acceptance Testing, http://www.xpuniverse.com/2001/pdfs/Testing05.pdf

2. Lisa Crispin, Tip House, Testing Extreme Programming. Addison Weseley; 2002; ISBN 0321113551

References

1. Johan Andersson, Geoff Bache, Peter Sutton. XP with Acceptance-Test Driven Development: A rewrite project for a resource optimization system. http://www.carmen.se/research_development/articles/ctrt0302.pdf

2. Kent Beck, Test-Driven Development. By Example. Addison Wesley, 2003; ISBN 0321146530.

3. Kent Beck, Extreme Programming Explained: Embrace Change, p180, p118. Addison Wesley, 1999; ISBN 0201616416.

4. Ronald E. Jeffries, Ann Anderson, Chet Hendrickson, Extreme Programming Installed, p32. Addison Wesley, 1999; 0201708426.

5. Kent Beck, Martin Fowler, Planning Extreme Programming, p88. Addison Wesley, 2001; ISBN 0201710919.

6. Brian Marick, http://fit.c2.com/wiki.cgi?CoachingTests.

7. Joshua Kerievsky, http://fit.c2.com/wiki.cgi?CustomerTests.

8. Lisa Crispin Senior Consultant Boldtech Systems Denver, CO USA 1.303.722.7964, lisa.crispin@att.net, Is Quality Negotiable?, www.xpuniverse.com/2001/pdfs/Special02.pdf

9. Jacqueline Mitchell, Customer, Egg Bank Plc.

10. http://fit.c2.com/wiki.cgi?WelcomeVisitors

11. http://sourceforge.net/projects/fat/

12. Alistair Cockburn, Writing Effective Use Cases, Addison-Wesley, 2000; ISBN 0201702258

13. Norm Kerth, Project Retrospectives: A Handbook for Team Reviews, Dorset House Publishing Co, 2001; ISBN 0932633447