Using ATDD To Build Customers That Care

SAMUEL HOTOP, Thoughtworks, Inc.
LAV PATHAK, Thoughtworks, Inc.
JEFFREY DAVIDSON, Davisbase, LLC

In this experience report, we present our implementation of Acceptance Test Driven Development in a complex, data-driven domain. We spent over a year as consultants in the Oil and Gas industry, engineering Natural Gas Pipeline management software that would manage the movement of gas across a number of pipeline systems in North America. We used ATDD to better engage product owners in the development process, build a ubiquitous language across a distributed team, promote faster feedback, and create an enjoyable team culture in which engineers and product owners alike were deeply invested in the success of the product.

1. INTRODUCTION

Samuel Hotop has made a career of seeking out and eliminating weaknesses in software. He spends his professional life on the road, embedded in delivery teams across a number of domains, collaborating with engineers to bake quality into their ideas, processes, and software. Lav Pathak is a passionate software developer with experience delivering high quality applications. Well versed in agile practices such as TDD, CI and extreme programming, Lav believes in teams where every member directly contributes toward code quality. Jeffrey Davidson is an acknowledged expert on the use of BDD by Business Analysts. He is at the forefront of change with a career spanning more than 2 decades helping organizations hone their business processes by improving their business analysis and product management practices.

2. CONTEXT

Our time on this project took place over the course of 18 months. We began in early 2012 and ended with our transition to an in-house team late in 2013. This was a greenfield project staffed by ThoughtWorks consultants and two primary client developers. We were tasked with building a natural gas pipeline management system that would move gas across a series of seven pipes that stretched from Calgary, Alberta down to the Gulf of Mexico. The domain was marked by its data complexity. We wrote complex equations and algorithms to determine pipe capacities. We implemented shortest path algorithms to make the movement of gas between two locations efficient. On certain pipes, there were thousands of gas shippers making requests to move gas in overlapping cycles throughout the gas day. These shippers competed for spots on the pipe, which had finite space, and in order to determine which shipper's gas would be moved, we had to take into account the shippers rank in relation to all other shippers who were requesting to move gas at that moment at that mile marker. The rank had to be derived from various types of contracts the shipper had set up with the pipeline owner. After a shipper's gas was on the pipe, we had to account for fuel cost, based on distance traveled, and had to facilitate the agreement between pipeline operators at upstream and downstream locations to confirm quantities that would be sent and would be received. We were focused on building multiple pipes that operated in this nature, each with their own unique business rules, and we had to allow for new pipes to be added in the future. The domain was new to everyone but our clients, and as consultants, we were expected to have working software up and running quickly.

Communication presented its own challenges. Our team was distributed between Recife, Brazil and Houston, TX. There was a technical team in Brazil that we shared the codebase with, in addition to business analysts and iteration managers that we interacted with daily. We took steps to mitigate the geographical distance, setting up a 24/7 live video feed between the US and Brazil that could be used for on-the-fly conversations, and we pair-programmed using a tool called TeamViewer. While the language / culture barrier wasn't really a problem, we faced real challenges up front disseminating information about the domain to our teammates in Brazil. The product owners, through whom the entire team was expected to learn the domain, were all housed full time in Houston. This was our first time working together as a team.

Author's address: Samuel Hotop, Chicago, IL; email: shotop@thoughtworks.com
Second author's address: Lav Pathak, San Francisco, CA; email: lpathak@thoughtworks.com
Third author's address: Jeffrey Davidson, Carrollton, TX; email: jeffrey.davidson@davisbase.com

Copyright 2014 is held by the author(s).

3. FRESH IDEAS

While the domain was complex, and working in a distributed team presented its own unique challenges, we did some key things early on that gave us a set of tools for success. One of the things we love to do with our clients is to try to find ways to inject fresh ideas into our thinking about the domain. This can happen in a number of ways. Many teams like brown-bag lunches, lunch and learns, etc. We think that it's good to try to break our thinking out of the status quo from time to time in order to reveal new paths previously untraveled. On our pipeline project, we decided to start a simple book club where we voted on books of interest, read a few chapters a week, and then over lunch, had individuals present the material to the team. One of the books we chose to study was *Domain Driven Design* by Eric Evans. *Domain Driven Design* [Evans] presented us with a series of ideas around tackling complexity. In the midst of our own knowledge crunch, a concept Evans covers, we needed a focused approach to defining our model and creating a model-based ubiquitous language that would help everyone communicate the domain. We needed domain concepts not only to solidify in our minds and in our speech, but to consistently permeate all layers of our code, from domain objects themselves, up through our tests. *Domain Driven Design* goes into great depth on how to make this possible and was essential reading for our team.

Another activity that injected a lot of new ideas into our thinking was a seminar the testers and analysts attended called *Specification By Example* [Adzic]. This was a concept created by Gojko Adzic that suggests a collaborative approach to creating tests, in which the team analyzes and generates executable specs that define the behavior of a new feature before the feature goes into development. The developers are then tasked with performing the same Red, Green, Refactor cycle they would for unit tests, but for higher-level acceptance tests.

The team also structured itself in a way that made sharing new ideas easy. Our approach to roles was unique. In many ways, we jettisoned the standard expectations for dev, tester, and business analyst (BA). As is the case with nearly all ThoughtWorks projects, we pair program. We think pairing creates an environment where it is easy to share knowledge, build skills, and create a higher quality product. This project stands out in that all engineers, despite their titles, paired. And that's how we've characterized our approach to roles: you are an engineer first. Do what you can with the skills you have to help the team at the moment you see the opportunity. As your skills grow over time, you should be capable of doing more to help. On the ground, this simple directive meant that testers would routinely pair with BAs and product owners to drive out executable specs and check them in before development. It meant that devs would routinely perform analysis on upcoming stories, looking for bugs in our thinking, or pair with testers on new feature development, looking to 'bake quality in'. It meant that BAs would pair with devs to debug test code, write queries, and do analysis of the domain code itself. This cross-pollination of expertise throughout our 2-week iterations made us a stronger team and helped us to create a better product. It also prepared our testers to write and maintain our non-standard acceptance tests.

Each of our three testers came into the project with a good amount of test automation experience. This experience, however, was limited to Selenium Webdriver type tests that would target elements on rendered html. Outside of the automation skillset, the testers had a good mix of accessory skills that we think is worth mentioning. One tester was especially proficient in blackbox testing and was able to achieve very deep domain knowledge that drove out a lot of elusive bugs. The other two were more technical and could handle devops tasks, like configuring new builds, as well as test framework maintenance. The mix of technical and analytical testers on the team proved quite effective for us. Out of necessity, though, each tester would need to expand his or her knowledge to include a deeper understanding of the Model View Controller pattern. In particular, they needed to be able to build up requests that would be passed to controller actions and to be able to query the responses for desired data. This request response paradigm would be the basis of all of our step definitions and would become fundamental to the implementation of our acceptance tests. The building up of this particular skillset occurred as a result of pairing with developers on a daily basis.

4. KNOWLEDGE CRUNCH

Eric Evans says, "Effective domain modelers are knowledge crunchers. They take a torrent of information and probe for the relevant trickle." We think this adequately describes what many of us face when beginning new projects. There are often so many new domain terms and new people who speak about concepts in similar, but often variable ways. With so much pressure on us to ramp up quickly and start hitting velocity numbers for our first few iterations, we had to be active participants in the process of separating out all the little nuggets of truth. We had to remain alert for inconsistencies and immediately question those who expressed concepts that

were contrary to the team's current understanding of our model. One of the major challenges arose from the fact that we were building the system from the ground up, which meant that we had to spend the first few months building out the most fundamental or basic concepts. The product owners we were working with had never worked with Agile teams before and we had to work with them to provide us with information relevant to the model as it existed at that moment, not as it would exist in a final product a year and a half down the road.

One of the most fundamental pieces of pipeline management is the Nomination - which is basically an order for gas. It took us three iterations of this term to finally whittle down a meaning, because it seemed to mean different things to different people. From varied use of the word Nomination, we eventually derived three different individual domain concepts: the nomination, the nomination transaction, and the nomination request. When we arrived, our clients were expressing these terms interchangeably and it was the result of constant questioning, of constant knowledge crunching, that we drove out the team's shared understanding and eventually its language.

It took approximately three months to get from the inception of the project to the point where we comfortable with our basic understanding of the gas day. Understanding the gas day was always an ah-ha moment for newcomers to the team, because it meant that you knew how the gas cycles overlapped and how cuts would be made at receipt and delivery locations based on the rankings of the shippers and how much gas they were shipping. Midway through our fifth month, we had a working Capacity Model coded that could determine cuts at any location on two of our seven pipes, the first of which was the most complex pipe. This was a major victory for the team.

Of course, we couldn’t have consumed this knowledge and started modeling it with the speed that we did without the help of some extraordinary product owners. They were passionate about their work in the energy industry. They made themselves available any time we needed them. And they remained open to learning and trying new things as part of our development processes. There were approximately 60+ years collective experience in the natural gas industry shared between the product owners, but what we quickly realized is that all that knowledge with regard to the inner workings of their existing pipeline systems was housed in code form in a faulty legacy codebase that we didn't have access to and, more significantly, in their heads. No one had ever really documented the processes necessary to move gas from point A to point B in a way that was accessible and maintainable. There was no living documentation, not even in the form of test cases.

The legacy system itself was a web of fixes that included traps like the “death spiral.” It wasn't quite clear how the death spiral was triggered, but it had been seen in the wild and many emergency levers and buttons needed to be pulled and pressed to stop it, if it was detected in time. Effectively, it was an infinite loop that would cut all nominations of gas on the pipe down to zero even though there was available capacity. We knew it was bad. And we wanted to avoid it. We were also fortunate enough to have had some understanding of what caused major failures in previous attempts at rebuilding the system. Rounding was one particular Achilles heel for the team that was in place immediately prior to us. Due to a lack of more acute testing and traceability, fractions of dekatherms of gas were not being rounded correctly, compounding over time and in such quantities that the issue was reason enough for a complete write-off of the work that had been done. The development team could not pinpoint the root-cause of the issue. These two pitfalls really illustrated the need for a testing approach that drove the design of the system. We needed living documentation that flushed out these pitfalls and that would allow us to explore new paths and scenarios.

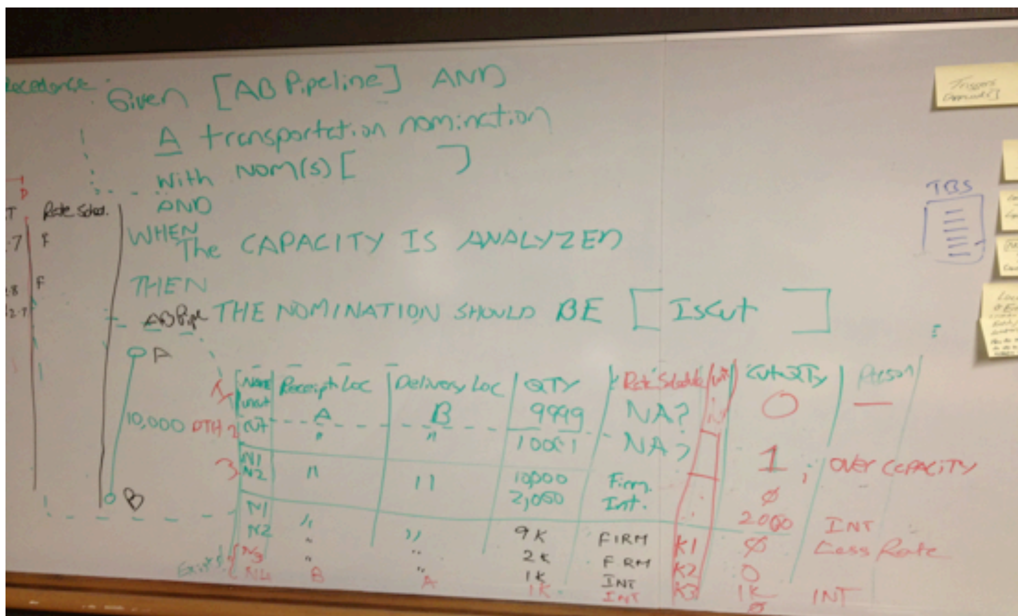
5. EARLY STAGES

So, we started with what felt like puzzles. The team would sit around the projector with basic inputs and domain concepts displayed on the white board and someone would moderate. "Ok if we nom 70,000 dekatherms from Fayetteville Express down to Moulton, and we run the capacity algorithm, based on the operational capacities of our downstream locations, what do we expect to be cut at Shorewood?" These full team exercises were important for a few reasons. First, they allowed us to find any inconsistencies in our model and narrow down our language. They were a haven for knowledge crunching. With everyone in the room together, product owners included, and Brazil on video, inconsistencies in language and meaning bubbled up quite quickly. It was through these sessions that we refined the meaning of nomination that we talked about earlier.

Early in the project, we were having these sessions daily, sometimes multiple sessions a day. They would generally last from 30 minutes to an hour, or however long it took to solidify the team’s understanding of a new concept or scenario. The product owners really seemed to get a kick out of this, which is the second

important point. These sessions helped to create those personal and professional relationships between the technology team and the product owners that we believe were an essential part of our success. We were a team of consultants brought in to work with a team of product owners that had never worked this closely with a development team before and had never taken such an active role in building software before. As operators of natural gas pipes, they were generally far removed from the software development lifecycle in their day-to-day work. We think at the outset they were a bit standoffish due to all the new technical jargon being thrown around. And we too were intimidated by the complexity of their work. These whiteboard sessions really gave the product owners a chance to flex their professional intellectual muscles and to show us what they knew. It got them engaged in our work and set the building blocks in place upon which we would implement our version of ATDD.

The final significant benefit of these sessions was that they acted as an incognito introduction to BDD, to acceptance tests, to Gherkin, and really to testing in general. We didn't bring everyone into a room and start showing slides about SpecFlow [SpecFlow]. We didn't go though trainings on Gherkin syntax or step definitions or table formatting. We didn't talk about any of this up front. We simply started writing our scenarios on the white board. Photo [A1] is from one of our sessions.



[A1]Team discussions around the whiteboard using Acceptance Tests (Photo by Sam Hotop)

You'll observe the Given, When, Then steps, the variable inputs, the table format for executing multiple examples. The technology team was well aware that this would eventually turn into an executable acceptance test, but the product team saw it simply as an intuitive way to run through our what-if pipeline scenarios and to continue driving out our collective knowledge of the domain. This was important, because when the product owners were originally asked to get involved in writing acceptance tests, they pushed back. They were still under the impression that test writing was a purely technical task and that they didn't have the time or expertise to learn the tools necessary for doing so. It took some convincing on our part that this was something that they could do and that we could provide them a way to really influence the team and define what would be built through these tests.

Eventually, they agreed to pair on one of our more complex capacity tests and were surprised at how similar the process was to our whiteboard sessions. We explained that all tests could be implemented this way and that if they needed new steps to define new functionality, they simply needed to create the English statement and we would implement the functionality. The lesson we learned here is that the introduction to a new approach, new idea, even a new tool, doesn't have to be formal and top-heavy. The team whiteboard sessions provided a ton of value, including ramping product owners up on a tool they didn't even know they'd be using. And all that was needed was a whiteboard and a dry-erase marker.

In the end, those scenarios became executable tests running with every new change to the system. And this was the moment that, looking back, we started really doing acceptance test DRIVEN development, roughly 3 months from the inception of the project. Now it wasn't that we weren't testing up until then. It really just took this long for the dust to settle, to the point that we were comfortable with our base knowledge of the system and that its early implementation was stable enough to start really building upon. It was also at this point that we made a concerted effort to get the product owners involved up front, so that BAs and testers could check in failing specs before the development process began.

The process moving forward from this point was fairly straightforward. The business would prioritize new features to be built in each two-week iteration with input from the development team. The team would perform analysis on these new stories, which included input from dev, test, and analysis roles, in conjunction with product owners. Stories would be presented during our iteration planning meetings so that the entire team could help knowledge crunch and drive out any inconsistencies. Once the analysis was driven out, a BA or a tester would work with the product owner to drive out SpecFlow tests. SpecFlow, sometimes referred to as "Cucumber for .Net," is a BDD framework that seeks to bridge the communication gap between domain experts and engineers by binding business readable tests to their underlying implementations.

Now for anyone who doesn't run the build locally, creating new SpecFlow specs could seem impossible. These tests live in the codebase. We needed our existing step definitions and existing tests to be accessible to everyone who would be writing new ones, and this included product owners and BAs who weren't running the build on their machines. We got the level of accessibility we needed from a tool called Pickles [Pickles]. Pickles is a .Net tool that can be used in conjunction with SpecFlow to create living documentation in a very accessible html format. We added Pickles to our project and had it generate an artifact after each commit build in TeamCity. We then simply showed the business analysts and product owners how to access that Team City artifact. The artifact was a pretty, searchable html document that showed every running test for that particular commit, so it was always up to date as long as you grabbed the latest one. Example [A2] shows some pickles output that would typically be used to compose new tests.

Home
NominationDataLoad
1
Create RFSRevolution
Accept Contract
Add Rates ANR FTS 1
Add Rates ANR FTS 3
Add Rates GTN FTS 1
Add Rates NB T 1
Add Rates NORTH BAJA FTS 1
Add Routes ANR FTS 1
Add Routes ANR FTS 3
Add Routes GTN FTS 1

PSV Bumping CounterFlow With Fuel

As a scheduler, I want to view details of the "Capacity Model" with the correct columns at a specific location so I can see nomination details as well as cuts details for the gas day.

PSV and Bumpable Quantities should be correct when there are Counter flow nominations

When I change operational information:

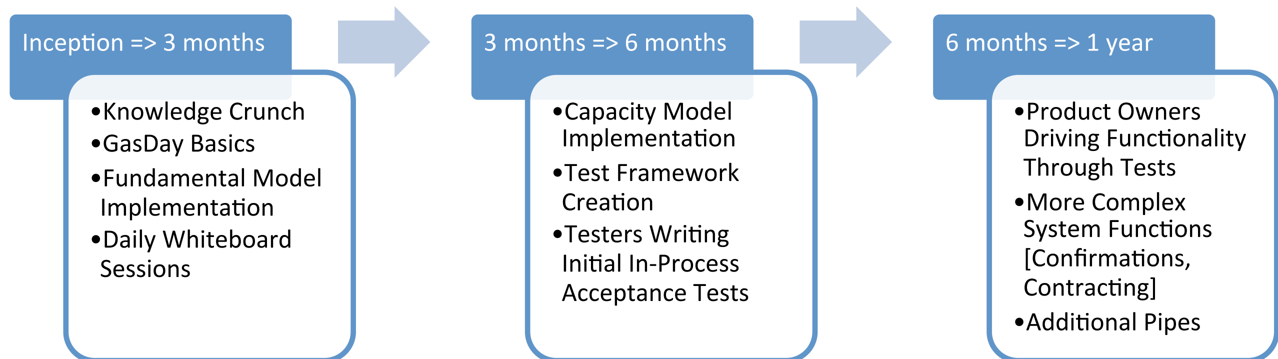
Location	Effective Date	Effective Cycle	Operational Capacity	Eval Direction	Constraint Location	Nomable
FAYETTEVILLE EXPRESS	01/01/2013	Timely	1,375,000	Receipt	Yes	Receipt
MOULTON	01/01/2013	Timely	3,100	Delivery	Yes	Delivery
EGAN (RECEIPT) INT	01/01/2013	Timely	3,100	Receipt	No	Receipt

[A2] Standard Pickles output used for test coverage transparency

A business analyst or product owner could access the artifact and click through the navigation links on the left to see what tests were running under each area of functionality. When composing new tests, they'd simply copy and paste combinations of existing steps or new English statements into an excel file, and sometimes even into a formatted SpecFlow file. These would then either be checked into the codebase on the spot and ignored until development began, or they'd be attached to the story cards to be checked in later. The ramp-up time necessary to get everyone on the same page with this was short. We just gave them the path to the artifact in the Continuous Integration server. Understanding what steps already existed and when to request new ones wasn't quite as simple and took more time.

At this point we were in approximately the 4th month of the project. We had built up a base set of steps to describe the system's most basic functionality, which the product owners didn't have as much a hand in creating. It took a little time for everyone to gain a clear understanding of what steps were already in existence. However, moving into the 7th and 8th months of the project, as new features were built out, new steps would

be delivered as part of Analysis sessions between the BA and product owner. This is really what allowed the product owners to drive out new functionality. Below is a **timeline** of the major events discussed thus far:



The product owners were energized in knowing that they had a clear role in creating new functionality and that they had a contract by which they could hold us accountable in the form of these tests. When a new feature would go up for approval, they would first look to see that the tests implemented up front were unignored and passing. This approach also freed up testers to do more exploratory testing for less obvious bugs, because they didn't need to spend time writing test cases or doing regression on the fundamental requirements for each feature. The documentation of each feature were the tests themselves and everyone had access to them.

6. IMPLEMENTATION DETAILS

We mentioned earlier that the time spent pairing between dev and tester was essential in that it prepared our testers to be able to write and maintain our tests. The tests were not traditional functional tests. We consider the traditional approach to be one that executes tests out of process against a running instance of the application. Often these tests will instantiate a driver that will scrape rendered html for desired elements and manipulate or query them in order to test assertions. Rendering was important to us, but we needed more focus on testing our algorithms and data heavy modules. These were the crux of our application and that's where our product owners would be able to help us the most, given their inherent knowledge of the domain. Also, it was clear up front that there would be many tests necessary to test all of the business logic and edge cases for each of the seven pipes.

We didn't want to fall into a situation where we had to have nightly test builds, builds that run over the weekend, etc. So as a team, we decided to jettison that notion and expose our system components in process while still applying a gherkin-based simple English layer on top. This way, the product owners would be helping us write component and integration tests, while still validating that business rules were being met in a way that was accessible. Implementing our tests in this way gave us an advantage. In process tests run on the same thread and use the same transactions as the production code they're executing, and they don't require the application to be deployed to run. On the other hand, out of process acceptance tests like Selenium functional tests require the application to be deployed and run on a different thread. Out of process tests are slower to execute and require more resources. And of course, reliability always seems to become an issue. Out of process tests don't have access to the thread the code is running on and have to wait for the responses as an outside service. This typically introduces asynchronous behavior, often referred to as "timing issues", which can also affect the reliability of the tests. With regard to JavaScript and client side validations, it wasn't that our app was totally lacking these things. We did have some "story flow" type Selenium tests that would execute this JavaScript out of process. It was just that they weren't our primary source of functional testing. Fortunately, most of the algorithm and business logic code was server side, which made writing the tests in process an easier strategic decision. The tabular format supported by SpecFlow worked perfectly for our math and data heavy domain and it was a natural progression from the whiteboard sessions we had been having with the product owners. [A3] is an example of one of the scenarios we implemented and [A4] is the implementation of one of our step definitions.

```
Scenario: Cuts are applied prorata within the same rank on nominations of the same contract on SB
Given we are at Southern Border pipeline
And these contracts exist:
| K   | Rt Sch | Start Date | End Date   | Entity                    |
| 100 | T-1    | 01/01/2000 | 01/01/2030 | Manheim Energy Marketing |
And contract 100 has the following routes and is executed
| Start Date | End Date   | Receipt Location | Delivery Location | Maximum Quantity |
| 01/01/2000 | 01/01/2030 | Manning (Receipt)| Will County       | 500000           |
Given these nominations have been created:
| Nom ID | Contract | Rec Loc           | Del Loc     | Rec Qty | Del Qty | Cycle | Up Rank | Down Rank|
| 1      | 100      | Manning (Receipt) | Will County | 50,000 | 50,000 | Timely | 1       | 2        |
| 2      | 100      | Manning (Receipt) | Will County | 250,000 | 250,000 | Timely | 2       | 1        |
| 3      | 100      | Manning (Receipt) | Will County | 50,000 | 50,000 | Timely | 2       | 1        |
When the Timely cycle nomination window closes
And the Timely cycle capacity model is approved
Then the capacity model details report for the Timely cycle at Manning (Receipt) has
| Nom ID | Contract | Nom Qty | Rt Sch | Priority | Rank | Constraint Cut Qty | Total Cut Qty |
| 1      | 100      | 50,000 | T-1    | Primary | 2    | 50,000             | 50,000        |
| 2      | 100      | 250,000 | T-1    | Primary | 1    | 85,000             | 85,000        |
| 3      | 100      | 50,000 | T-1    | Primary | 1    | 17,000             | 17,000        |
```

[A3] Example SpecFlow Scenario

```
[Then(@"the capacity model details report for gas day (.*) for the (.*) at (.*) has")]
public void ThenTheCapacityModelDetailsReportHas(DateTime gasDay, CycleType cycleType,
ApplicationLocation location, Table table)
{
 CheckAllNominationsWereCreatedCorrectly();

 var schedulingController = ObjectFactory.GetInstance<SchedulingController>();
 var response = schedulingController.Show(location.Id, gasDay, cycleType.Id);
 var viewModel = (SchedulingShowViewModel)response.Model;


 var detailRows = viewModel.NominationTransactions.ToList().Select(nominationTransaction =>
 {
   var detailRow = new CapactiyModelDetailRow();
   var tableKey = nominationTransaction.NominationId.ToString();
   try
   {
     tableKey = _entityContext.GetNomID(nominationTransaction.NominationId).ToString();
   }
   catch { }
   detailRow.NomID = tableKey;
   try
   {
         detailRow.ServiceRequesterContract =
        _entityContext.GetContractContext(nominationTransaction.NominationContractId).ToString();
   }
   catch
   {
      detailRow.ServiceRequesterContract = nominationTransaction.NominationContractId.ToString();
   }

   detailRow.NomQty = nominationTransaction.NominatedQuantity.ToWholeNumberFormat();
      detailRow.IncrementalNominatedQuantity =
        nominationTransaction.IncrementalNominatedQuantity.ToWholeNumberFormat();
   detailRow.RtSch = nominationTransaction.RateSchedule;
   detailRow.Priority = nominationTransaction.Priority;
   detailRow.Rate = nominationTransaction.RateDisplay;
   detailRow.Rank = nominationTransaction.Rank;
   detailRow.ConstraintCutQty = nominationTransaction.ConstraintCutQuantity.ToWholeNumberFormat();
      detailRow.TotalCutQty = nominationTransaction.TotalCutQuantity.ToWholeNumberFormat();
   detailRow.PreviouslyScheduled = nominationTransaction.PreviouslyScheduledVolume();
   detailRow.RedirFlag = nominationTransaction.RedirectFlag ? "*" : "";

   return detailRow;
  });
table.CompareToSet(detailRows);
}
```

[A4] Example Step implementation

By looking closely at the code, you'll see that the step definition implementation isn't making http calls using page objects or manipulating html elements. Instead, it's calling one of our end points directly by injecting an instance of the Scheduling controller class. Also, it's not looking at the response from the server through rendered html, but directly capturing the response from the endpoint and extracting the model out to compare the results. To be able to achieve this, our production code classes needed to be injected into our test context. We did this by setting up hooks before each feature was initialized. SpecFlow allows this by setting an attribute on the method that you want to execute before each feature execution begins. [A5] provides an example of this.

```
[BeforeFeature]
public static void BeforeFeature()
{
  Bootstrapper.Instance.Initialize();

  ObjectFactory.EjectAllInstancesOf<IPipelineContext>();
  ObjectFactory.Configure(x => x.For<IPipelineContext>().Use(new StubbedPipelineContext(() =>
Pipeline)));
```

[A5] BeforeFeature Attribute

Here Bootstrapper.Instance.Initialize() is making a call to the bootstrapper class which is setting up all the dependencies using structure map. The [BeforeFeature] attribute then executes the method before every feature so that all of our dependencies are in place. With this setup, we have everything we need to run in process.

Our tests ran faster in process, but the sheer number of them meant that we needed to make some additional adjustments if we weren't going to compromise on one of our prime directives as a team: to run all of our tests on each change to the system. We needed to get feedback as soon as possible, which meant that we had to keep our build times, especially local build times, reasonable. Some of the domain algorithms and calculations under test were more expensive than others. So, to avoid running them every test, we created stub objects that were injected into the tests if they were marked with specific tags. In one case, we had an object whose responsibility was to calculate Fuel consumption during gas transportation. We didn't need that calculation to happen on tests other than the ones testing the fuel calculation functionality itself. So, we created a stub fuel calculator class [A6], which was instantiated in the before hook.

```
// Removing costly pieces of code that are supposed to be exercised on specific tests
// and replacing with lighter ones
ObjectFactory.Container.EjectAllInstancesOf<FuelCalculatorFactory>();
ObjectFactory.Inject(typeof (FuelCalculatorFactory), new FuelCalculatorFactory());
```

[A6] Create a Stub Fuel Calculator Class

Once instantiated, these stubs could be injected simply by tagging a test with the defined tag. [A7] illustrates this example. The method WithoutFuel will be executed before any feature or any scenario tagged with the "withoutfuel" tag. These tags are used in the feature files as shown in figure [A8]. This will run all the scenarios in the feature with an injected stub fuel calculator instead of using the real object.

```
[BeforeScenario("withoutfuel")]
[BeforeFeature("withoutfuel")]
public static void WithoutFuel()
{
  ObjectFactory.Container.EjectAllInstancesOf<FuelCalculatorFactory>();

  var mock = new Mock<FuelCalculatorFactory>();
  mock.Setup(x => x.GetInstance(It.IsAny<Pipeline>())).Returns(Mother.CreateZeroFuelRateCalculator());

  ObjectFactory.Container.Inject(typeof(FuelCalculatorFactory), mock.Object);
}
```

[A7] Defining a Tag

```
@withoutfuel
Feature: Auto Confirm Based On Previous Cycle with two nominations
```

[A8] Designating a feature with a tag

The second adjustment we made was to avoid querying the same objects over and over again throughout the execution of a test scenario. We achieved this by sharing data via context injection, a feature provided by SpecFlow. In SpecFlow, step definitions are global, so the steps of a single scenario could be bound to step

definitions in multiple classes. It is common to share data between step definitions during the execution of a scenario. For example, a Given step could prepare some data to insert or query some data to be used later, and in a Then step you could use the same data to validate the transaction. [A9] is an example of this.

```
[Given(@"this revolution contract draft has been created")]
[When(@"this revolution contract draft is created")]
    public void WhenThisRevolutionContractDraftHasBeenCreated(Table table)
    {
      _entityContext.FailureMessagesList.Clear();
      var requestForServiceController = ObjectFactory.GetInstance<ContractCreateController>();
      requestForServiceController.SetLoggedInUser(CurrentUser());
      table.CreateSet<ContractRow>().ForEach(contractRow =>
      {
                var mapInvalidContractRowToViewModel = MapContractDraftHeaderRowToViewModel(contractRow,
                Pipeline);
        requestForServiceController.Validate(mapInvalidContractRowToViewModel);
        var actionResult =          requestForServiceController.Create(mapInvalidContractRowToViewModel);

        if (requestForServiceController.ModelState.IsValid)
        {
          var redirectToRouteResult = actionResult as RedirectToRouteResult;
          var contractNumber = (int) redirectToRouteResult.RouteValues["contractNumber"];
          _entityContext.ContractRevolutionNumbers[contractRow.K] = contractNumber;
        }
        requestForServiceController.AddFailureMessagesTo(_entityContext);
      });
    }
```

[A9] Multiple Steps Sharing Data

Here we are creating the Contract object and storing the messages that have come back from the server in _entityContext. Entity Context is a custom class that was created to store scenario context related data. This class can be injected using the context injection provided by SpecFlow. Now we can use the messages in later scenarios to validate the response from the server. This is shown in [A10].

```
        Then(@"revolution contract is submitted successfully")]
    public void ThenRevolutionContractDraftIsSavedSuccessfully()
    {
      _entityContext.FailureMessagesList.Should().BeEmpty("Expected no validation failures");
    }
```

[A10] Validate Response

Finally, in order to speed up of our builds even further, we decided to parallelize our tests. We divided the tests into four different groups and executed them in parallel. It was a bit of trial and error to divide them into similar sized groups so that all the groups would execute in roughly the same amount of time. Since our CPUs were multi core on both development and build machines, we were able to distribute and run them in parallel locally and remotely. These adjustments did a lot to kept our local and CI build times down. At the time we rolled off the project, our local build times, running all tests, were approximately 16 minutes. At this point we had spent nearly 18 months in Houston and had written approximately 3000 tests across various layers of our system: 1600 Unit tests, 1250 Acceptance tests, and 100 Functional UI tests.

Not everything in our test approach was a success, however. The number of unit tests we had at one point nearly equaled the number of acceptance tests we had. This was a smell and it revealed that we had become too reliant upon our acceptance tests. It was true that business rules were being validated and our product owners were happy with that. However, we realized we were neglecting our unit tests in favor of the Acceptance tests. When it came time to do a major refactoring, our lax approach to unit testing hurt us. We made the decision to refactor our contracting domain entities into a service. This was a substantial refactoring and took more time than necessary because we didn't have the unit level specifications in place to make the architectural changes we needed to. This experience taught us a valuable lesson. We regrouped and refocused our attention on our unit tests, primarily because we were unsure if new architectural changes would be

necessary in the future and we wanted to be ready for that. Acceptance tests, even those run in process, should never replace unit tests, as they serve different functions. Teams should be cautious not to get too comfortable.

7. DEFECTS

Our defect process was similar to our new feature process in that it too was driven by tests. Tests, over time, became the tool the team used to communicate. This was really a result of the growing complexity of the system. To sit down and "explain" how some of our functionality worked became more and more difficult over time because there were so many subcalculations that had to be performed to reach the result you were looking for. It was too much to keep in your head. When new team members joined the project, we first pointed them to the tests. They were the living documentation and explained, with examples, how the system worked. Our best domain experts, the product owners, would often find the most elusive bugs. With all of their years of experience, they had a better sense of where weaknesses in the system might occur. The process was simple. A product owner might find a bug. They'd come to a business analyst or Tester to do the preliminary triage. At that point, a failing test would be written and checked into the code base. This test would be ignored and the defect card number would be added to the spec so that the developer who worked on it could easily find it. A developer pair would typically pick up a bug within 24 hours. There was a lot of focus and attention put on documenting and fixing bugs quickly because of experiences our client had with previous development teams and seemingly small bugs, left unattended, that resulted in large failures.

8. WHAT WE LEARNED

The project is still ongoing. After 18 months on the ground, we transitioned the work over to in-house developers and contractors, something we were happy about. It felt good to get something like this off the ground and to leave our product owners in a position to carry on the work for themselves. Our experiences on this team taught us a lot. Going out of our way to study together as a group was key in that it injected fresh ideas into our thinking about the project. These ideas had an impact on how we built our software and how we implemented our test strategy. We can't stress enough how valuable it was to bring the product owners into the development process. While they weren't totally sold on the idea at the start, we think over time they really saw the value that ATDD can bring. The notion that abstract ideas can become working software and that business people can drive that creation through tests was a powerful one for them. Bringing them into the team room for our whiteboard sessions, empowering them to create that contract between the business and technology teams, and constantly focusing on driving out a model that was clear to everyone were key ingredients in getting the project off the ground and stabilized. Our technical strategy of implementing our tests in process also sped things up and gave us the fast feedback we needed to move quickly as a team. The moment we transitioned off the project, the team was hitting its highest velocity numbers to date.

This was also a team where everyone went out of their comfort zones, product owners included. The work we did to blend roles and break down traditional barriers between them really created a sense of cohesiveness and trust. It made time on the team very challenging, but also extremely enjoyable and rewarding. The skills we as developer, tester, and business analyst gained while working in this environment were valuable. In addition to new technical expertise and valuable consulting experience, we have a new way of looking at tests as a form of communication. We believe that the techniques we employed to create this collaborative environment, based around acceptance tests, set the stage for our success in the face of many challenges. We hope that you can take what we've learned and apply these techniques to your own projects and build upon them.

9. ACKNOWLEDGEMENTS

We would like to thank our fellow teammates and product owners for making this an extraordinary experience for us. They were a remarkable team of people to work with. And finally, many thanks to Rebecca Wirfs-Brock, our shepherd, for all of her help and honest feedback in putting this experience to paper. We can't thank you enough for your guidance throughout this process.

REFERENCES

Adzic, Gojko, *Specification By Example: How Successful Teams Deliver the Right Software*, Manning Publications, 2011
Evans, Eric J., *Domain Driven Design*, Addison-Wesley, 2003
Pickles, Copyright © 2010-2012 Jeffrey Cameron, http://www.nuget.org/packages/Pickles/SpecFlow, http://www.specflow.org/