Automated Acceptance Testing of High Capacity Network Gateway

Ran Nyman[1], Ismo Aro[2], Roland Wagner[3],

[1,2,3] Nokia Siemens Network, PO Box 1
FI-02022 Nokia Siemens Networks
[1]ran@rannicon.com, [2] ismo.aro@nsn.com, [3] ro.wagner@gmx.net

Abstract. In this paper we will explore how agile acceptance testing is applied in testing a high capacity network gateway. We will demonstrate how the organisation managed to grow agile acceptance testing from two co-located teams to 20+ multi-site team setup and how acceptance test driven development is applied to complex network protocol testing. We will also cover how the initial ideas that we had of agile acceptance testing evolved during product development. At the end of paper we give recommendations to future projects using agile acceptance testing based on feedback that we have collected from our first customer trials.

Keywords: automated acceptance testing, scrum, organizational change

1 Introduction

At the end of 2007 we started having a discussion how to build a high capacity network gateway from scratch. We faced two fundamental risks. First, the technology was completely new and has never been used before in Nokia Siemens Networks. Second, the use cases for first commercial deployments were not completely defined at program start - It became clear that we need to adapt feature content heavily throughout the program. Applying Scrum appeared to be the appropriate response to these major risks.

The initial idea was to build broad band network gateway and after few months of development we realized that there would more market demand for a gateway for 2G/3G and long term evolution (LTE) enabled mobile networks. Luckily we had chosen agile methods to develop the product and these methods provide us the flexibility to change the direction smoothly. The hardware (HW) and software (SW) platforms we selected were totally new and in the beginning of the development they were not available. So we used HW that had the target CPU but was totally different from the ATCA blade architecture that we would use in commercial product. We used same approach with the platform SW because the high availability SW platform was not ready.

2 Organisation and growth

Selecting Scrum as agile development framework was easy; implementing it in practice was hard work. The first challenge was to convince all parties that feature teams are better than component teams. The feature teams that we decided to use, after long debate, are long-lived, cross functional teams which complete many end-to-end customer features [1].

When we started the product we started with people coming from two totally different backgrounds. The first team had used Scrum over one year and successfully created a product. The other team came from a traditional waterfall organisation that had failed to apply Scrum and had resistance in trying it again. The failed Scrum implementation was not real Scrum implementation it had just consisted of renaming waterfall development to Scrum. So the approach that we used was to mix the teams so both teams would have members from waterfall development background and agile development background. We put all development people in a room and asked them to organise themselves into two feature teams.

At first, the teams did not want to use feature teams because they claimed that the feature teams lead to bad software quality. An Agile coach present in the meeting asked gently what is the quality of the code that the component teams had created in their previous project? The answer was that a mess. So after some discussion we agreed to see how the code would end up when using feature teams. After the teams agreed to try feature teams the forming of teams went very smoothly and it took under one hour. The newly formed teams were allowed to select their Scrum masters from two available Scrum masters.

We had our first teams and development could start. In the literature the recommended approach is to start with one team and then grow when you have enough infrastructure built [2]. We decided to start with two teams. This led to huge arguments between teams and very slow start in development because there were so many different opinions how the architecture and infrastructure should be done. In the beginning there were difficulties in planning, because in a sequential life cycle model there is a long planning and specification period. Jumping into agile style where only minimum amount of work is planned was hard for people who did not have agile development background.

2.2 Growing first wave

The first growth point was to add two more teams. It was a challenge since they were transferred from traditional organisation. One of these teams refused to learn new testing tools and new way of working. They did not produce anything that could be considered done for several sprints in row. The team argued that the testing tools in their previous environment were much better and resisted the learning of new tools and did not want to write unit test. In retrospect one crucial point that caused the resistance was that we did not provide the sufficient training and the reasoning why things are done differently when using iterative development. Also the new teams

should be able to influence the ways of working that have been agreed so they can feel the rules as their own.

The only good thing in adding new team without breaking existing teams was that the velocities of the existing teams did not suffer any significant impact because of the new teams. We added still few more team to our main development site and adding them did not cause so much troubles as adding the first two teams.

2.3 Growing second wave

Adding teams to the same site was easy compared to next step where we decided to add teams at a second site to speed up development because the market demand for the product that we were creating was suddenly emerging. Here we found out that using iterative development and automated acceptance testing really paid of. We trained the subcontractor in our ways of working by having them spend several weeks with our local team doing work as team members until we were confident that they could work by themselves..

The same coding and testing rules were applied to subcontractor that were for our own teams. They had to write unit test, create automated acceptance test for all code and use the central continuous integration system. The biggest challenge in working with second site was the distance. It was hard to communicate the requirements and in the first half a year we had one person working as product owner proxy for other site to reduce the misunderstandings in requirements.

2.4 Current team structure

After adding several teams we have now over 20 teams and the majority of the teams are developing and documenting features. We have couple of teams in supporting roles like performance testing, system testing, coaching and continuous integration (CI) team. CI team is taking care of building and automation system. The system testing team is focusing on executing test that can not be done by Scrum teams because the need of the real network elements which we have only a limited amount and coordinating the usage of them between several teams is not feasible. The coaching teams main responsibility is to support in modern engineering practices, help teams to solve difficult technical challenges and in general help the organisation to learn faster.

2.5 Expert Coaching

In the beginning we realised that we need expert coaches that can help us in using modern SW development practices. First we had two consultants who helped us to set up the CI environment that was not so straight forward because the building and installing the build to target HW was complicated. To get the first teams in speed with

unit testing and test driven development (TDD) we used one world class consultant helping in setting up the unit testing framework and teaching teams how to test drive their code. TDD was not widely accepted in teams but unit testing was found useful by the teams. We used also help in teaching people acceptance test driven development that helped the teams to understand the concept.

3 Test Automation Strategy

It was clear in the beginning that we did not want to write legacy code that has no test as legacy code is defined in [3]. So we decided to have unit test coverage target and our aim was to automate all acceptance tests. We are now in situation where all user stories are unit tested, acceptance tested automatically and exploratory testing is done based on agreement with the area product owner. This has led to situation where almost all testing is done automatically and any manual testing done by teams is an exception.

3.1 Regression tests

Regression tests consists of unit tests, smoke test and all automated test. For every commit unit and smoke tests are executed and if those test cases fails, commits to code base are not allowed until problem has solved. During night time we execute whole regression set that contains all acceptance tests. The regression set consists of all automated test cases that we have developed. As defined in [4] there is no cost in adding all automated test in regression set and test from regression set should be removed only if the functionality they test becomes obsolete. There is agreement with product owner and teams that all acceptance test cases should be passing at the end of the sprint.

3.2 Acceptance test driven development (ATDD)

The idea of ATDD [5] was already known to some of the people and they had also experience in applying it in product development. The idea of ATDD came from Robot Framework [6] developers Pekka Klärck, Juha Rantanen and Janne Härkönen. The basic idea is to acceptance test every requirement which comes into the sprint and at that the moment acceptance tests are discussed for the first time and planned at a high level. The initial idea in ATDD was to have ATDD-meeting after each sprint planning where test cases are clarified and agreed how they will be implemented. Currently ATDD practices vary team by team but only acceptance tested requirements are considered as done.

3.3 Structure of test cases

In the previous project where we piloted Scrum, we started writing test cases at a very technical level and it was extremely hard to understand what test cases were doing without deep domain and tool knowledge. Then we found out that this approach was not working and started writing tests in business language. This was our experience and we wanted to try the same approach in this project, but there was huge resistance to create higher level language to test cases. Reasoning was that it does not give any value and also the way of how we use protocol tester does not support this kind of step by step presentation. One more reason for dropping this more readable way of writing test cases is that our test cases usually don't test any end to end functionality that has business value (see conclusions) and they are not read by product owner and area product owners.

4 Test Automation and Continuous Integration (CI)

There were two options what to use as testing framework when we started the development. HIT which is an in house test scripting tool and Robot Framework. There was not any formal decision by anyone and two initial teams started using Robot Framework because its usage was much simpler than HIT and it was easily integrated to CI environment. After it was already in use it was decided that it would be the testing framework for this product. Robot Framework is a generic keyword driven [7] testing framework so we also needed protocol tester.

Catapult [8] was chosen because there was no previous experience of acceptance testing this kind of product and Catapult was used successfully in previous, non Agile developed products as a protocol tester. The Robot framework is executing long catapult scripts. There has been now discussion of changing way of using catapult or even replace it, because how we are using it at the moment does not support ATDD.

In Figure 1 is our initial testing environment and in Figure 2 is the current environment. Currently we have switched most of the builds from Bamboo CI system [9] to Build Bot [10] system. Builbot executes build, which includes compiling and unit testing. Builbot executes also our sanity tests. Sanity tests represent smoke testing [11]. Buildbot is executing Robot Framework where those sanity test are. At the moment we have three different sanity builds, one for our Robot Framework test material, one for product code and one for catapult. When there is change in one these diffrened parts only that sanity is executed. Sanitys are executed in series, so only one commit is tested in one sanity run. There is dedicated environment for sanity testing.

4.1 Continuous Integration Practices

Teams are using their own development environments to test pre commit changes to avoid breaking builds on CI. This practice is crucial since we have so many teams and

having each team to commit to trunk without first verifying the change would lead to situation where the build would all the time broken and the cause of failure would be hard to find. When we had fewer teams we committed directly to trunk and let the CI system to inform possible problems but we had to modify the practice when we grew bigger than 10 teams. In Figure 3 we can see the daily build success rates that we have currently and the number of failing test cases in case the build fails (black bar in picture).

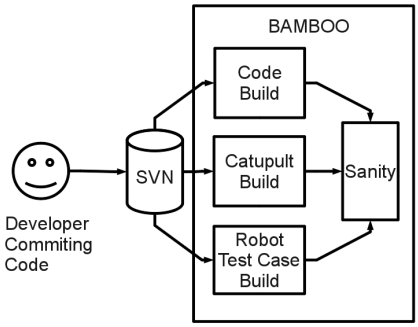

Fig. 1. Initial CI System

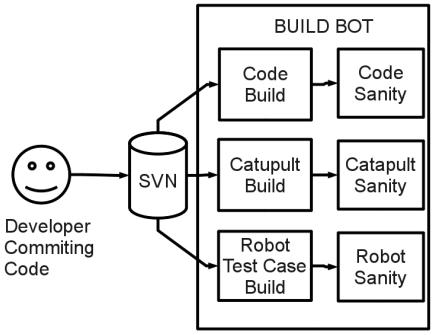

Fig. 2. Current CI System

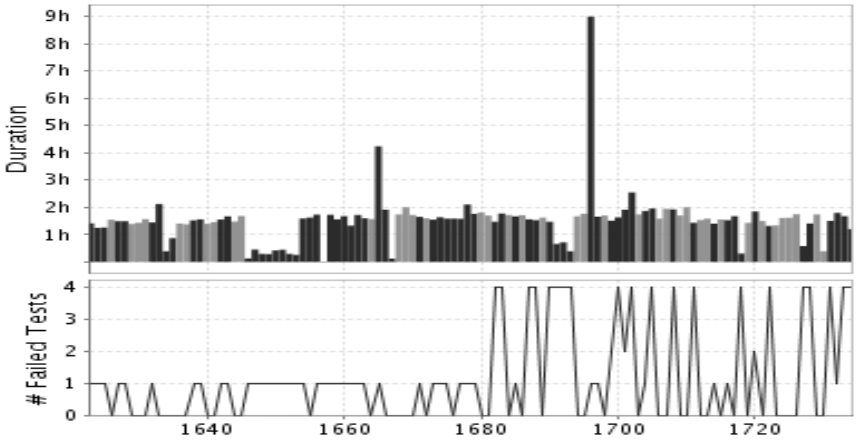

Fig. 3. Build status (gray is successful build) and failed test cases per build

5 Analysis of achieved results

Having unit test coverage as target backfired and we found out that people were writing unit test just to get the coverage but not testing anything. Having targets seems to backfire as described in [12]. We decided to remove the unit test target and to our surprise the unit test coverage did not drop significantly. Our current branch coverage is 75% and the mandatory target was previously 80%. It seems that we should have focused on training people when they joined our product development instead of having targets on unit test coverage. The first two teams who received unit testing training did not have time or necessary skills to train new teams when they joined and it lead to unit testing that was not meaningful.

Having automated acceptance test was one of the key success criteria why we managed to grow the development to multi site and still to maintain the high quality of the code base. The growing regression set is seen in Figure 4.

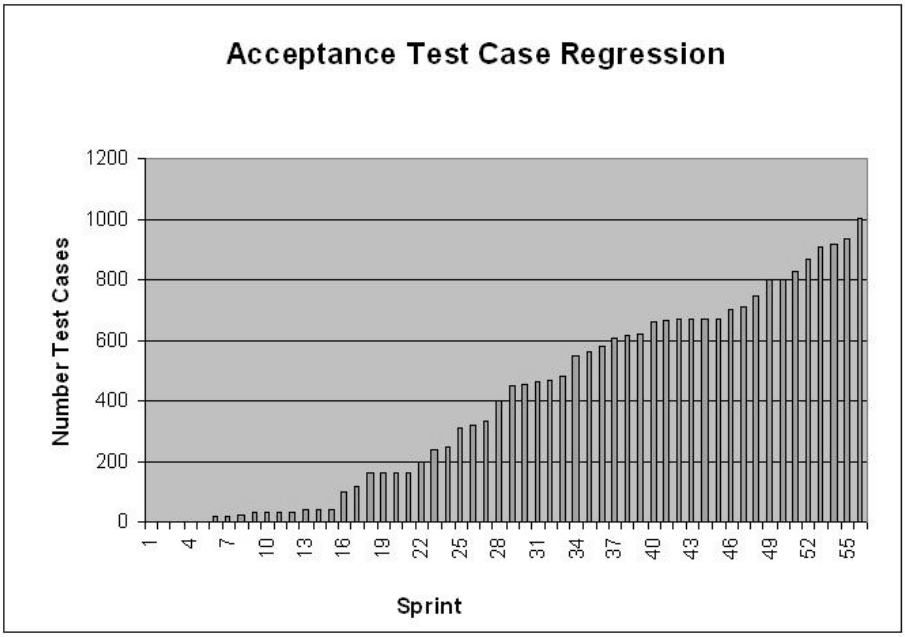

Fig. 4. Regression set growth

The moment of truth for our product came when we had our first customer trial. Even the huge amount of acceptance testing did not save us from missing functionality that the customer noticed in their trial in their own environment. The analysis of the faults and missing functionality revealed to us that we must have end to end acceptance test cases written on higher level so that they test the customer functionality and that we should have the same network elements that the customers have. Many of the findings in customer trial were incompatibilities with the customer elements because we had different interpretation about the specification than the other network element vendor. When we fixed the findings in customer trial we created new test case that we can be sure that the functionality that we create will also work with future modifications.

We tried to patch our lack of end to end test by having separate system verification team but even they were not able to find the missing parts of the functionality due to two reasons. They were not collaborating with the teams developing the functionality close enough so they would have clear picture what to test and they also lacked HW that the customer had.

We also found out that the feature teams that we created in the beginning were turned to functionality area teams and were not able to create end to end functionality inside one team. The functionality areas that we have are interfaces towards other system facing our gateway product. It seems that the product that we create is so big that one team can not handle the incoming and out going interface so they would conform to feature team definition that was our goal. Now when we have most of the interfaces in working shape we will try again to move more towards feature teams.

6 Conclusions

The selection of Scrum and agile development methods significantly accelerated the time to market and gave us flexibility that our traditional development methods never offered. The previous gateway product where we used sequential life cycle model took twice as long to develop and the sequential life cycle would not have allowed us to change direction of the product development as fast as agile methods. Automated acceptance testing helped us significantly when we added new teams to development to keep the code base in high quality.

One easy thing that were we should have put more effort was the speed and reliability of the CI system as the feedback speed from each code change should be as fast as possible as mentioned in [13]. Also the CI system should have been planned more carefully as it did not sustain the adding of new teams as easily as we thought.

On testing side we should have had more focus on exploratory testing. We were too excited about 100% automated testing and only automated testing. It came obvious after the first customer trials that we need to amplify the usage of exploratory testing [14] in teams to ensure product quality.

We also found out that splitting user stories to very small parts so they could fit in the one sprint leads to situation that testing is done at very low level. That makes problematic to have really end to end test cases which give real business value. We should bring acceptance testing more closer to customer, and use acceptance tests as a communication tool between all stakeholders [4], from customer to the developers and testers. This would also give us better visibility what functionality is ready ship and what not.

There is also one other big reason to have these upper level acceptance test cases: these test cases, these test cases should verify that nothing has lost because of splitting requirements and also ensure that no information has been lost in communication between different stakeholders. Having system verification team to patch the lack of exploratory testing and missing high level test cases is not a solution that works due to communication challenges between teams.

References

1. Larman C., Vodde B.: Scaling Lean & Agile Development: Thinking and Organizational Tools for Large-Scale Scrum. Addison-Wesley, Boston (2009)
2. Schwaber K.,Agile Project Management with Scrum. Microsoft Press, Redmond (2004)
3. Feathers M.C.: Working Effectively with Legacy Code. Prentice Hall PTR, New Jersey (2005)
4. Adzic G.: Bridging the Communication Gap. Neuri Limited, London (2009)
5. http://testobsessed.com/wordpress/wp-content/uploads/2008/12/atddexample.pdf
6. http://robotframework.org
7. http://en.wikipedia.org/wiki/Keyword-driven_testing
8. http://www.ixiacom.com/products/display?skey=ixcatapult
9. http://www.atlassian.com/software/bamboo/

10.http://buildbot.net/trac
11.http://en.wikipedia.org/wiki/Smoke_testing#Smoke_testing_in_software_development
12.Austin R.D.: Measuring and Managing Performance in Organizations. Dorset House Publishing, New York (1996)
13.http://www.martinfowler.com/articles/continuousIntegration.html#KeepTheBuildFast
14.http://en.wikipedia.org/wiki/Exploratory_testing