Case study: Distributed Scrum Project for Dutch Railways

This item in japanese

Scrum provides a proven foundation for the execution of projects. However, in every project the Scrum process must be adapted to address specific needs and circumstances. How this is done is a large factor in the success or failure of a project. In this article, we describe how we successfully executed a large (20 man-years, 100.000+ lines of code) Scrum project, one which had already been scrapped once under a traditional approach, and which included developers in both India and the Netherlands. In the hope of helping readers run successful, large Scrum projects, we share here our lessons learned on: project startup, getting the right product owner, the importance of estimates, effective communication, testing and documentation.

Background

The Dutch railways are among the most heavily used in the world, providing transport for 1.2 million passengers daily. Dutch Railways built a new information system to provide travelers with more accurate travel information, requiring less manual intervention. As part of this program, we built the PUB (publish) system that centrally controls information displays and audio broadcast systems in all stations.

The first attempt to build the PUB system was executed using a traditional waterfall approach. Detailed requirement specifications were handed over to the IT vendor, expecting a fully built system to materialize without much further customer involvement. After 3 years, the project was cancelled because the vendor failed to deliver a working system. The customer then engaged our company to build the PUB system from scratch. We introduced an Agile approach using Scrum, focusing on close cooperation with the customer, open communication and working in small increments.

Setting the stage

We started the project with a project kick-off to set up everything that is needed before the first sprint starts. This 3-week kick-off was done by a project manager, an architect and a Scrum Master.

requirements. We nominated two business analysts to fill the product owner role. They could be made available and because of their involvement in the earlier attempt to build PUB, they had built up a lot of business knowledge to be good proxy Product Owners to several groups of customer stakeholders. High level decisions about priorities and scope were made by a project manager mandated by the customer, but who had limited availability and less detailed knowledge about requirements. In general this model worked well, but on some occasions the project manager changed the priorities that had just been set during the planning meeting (ini his absence) and the meeting had to be redone. Ideally, the person with the final say about priorities should always be present at the sprint planning meetings.

Because of the earlier attempt to build PUB, detailed requirements documentation for parts of the system were available. They followed the MIL standard [1]. This form is not suited for use in Agile planning and estimation [2] because they are not grouped in small chunks that can be built, tested and demonstrated in a sprint. The Product Owners were not experienced in writing user stories. To address this, the Scrum Master helped them to produce the initial product backlog containing fine-grained, estimated user stories for the first few iterations.

The software that we built was part of a bigger program that involved multiple related software systems, building the displays and installing the displays on the stations. To be able to orchestrate this multi-disciplined effort, meeting deadlines was important. Therefore, we needed to provide a complete overall picture of the planning. We addressed this problem after a few iterations by making 'best-effort' estimates for these functional areas of the system. By that time, we also had established reliable knowledge about our velocity. Therefore, we could track and communicate progress very well using our release burndown chart. The lesson here is that any estimate is better than no estimate, even if little information is available.

Scaling up to distributed teams

people located in India. For this reason, two Indian developers joined the project from the first sprint onwards. They worked on-site at the customer site for 6 weeks to become familiar with the application domain, key customer representatives and the rest of the development team.

One of the first steps towards becoming a team is to agree on how to work together. To facilitate this we held a norming and chartering session [3] with all team members, including our colleagues from India. In this session we decided on practices like how to do pair programming, use of tools, quality targets, core hours etc. and recorded them on a Wiki page. Having agreements which are shared by the whole team proved very useful to us. When changes are made to these agreements, for instance during a retrospective, the practices should be updated so they are up-to-date when new members join the team.

In the first iterations, the team proved to be able to build, test and demonstrate user stories that formed the heart of the system. This pleased the customer, especially because - compared to the previous attempt - progress was demonstrated much sooner and the customer had more control over the course of the project.

After a couple of iterations we scaled up the project: the Indian developers returned home and we added resources in India and the Netherlands to create two Scrum teams of 5 developers each, sharing a single tester. Later this scaled up further to 3 development teams with 3 testers. Key to our approach is that each Scrum team consists of resources in both India and the Netherlands. This model proved very successful in maintaining a high velocity and quality [4].

So how do we work together in one team from several locations? First of all, we use Skype intensively. We have webcams, headsets, good microphones and a large screen. This way, we can do one-on-one meetings as well as all-hands meetings together. All of this works with off the shelf components and free software. We did not need any specific investments, other than enhancing the network facility in India using UPS to ensure that we could continue with Skype sessions during power outages. Second, we chose to only do pair-programming within the same location. This meant that we had pairs in India and pairs in the Netherlands. In our experience the interactivity of pair programming still requires you to be physically co-located, regardless of available tooling.

much better than a whiteboard. ScrumWorks also worked well when discussing the product backlog with the product owners.

To implement this distributed model we had to overcome some challenges. For instance, the product owners did not feel comfortable speaking English. According to Scrum, the sprint planning meeting consists of two parts – in the first part, the product owner clarifies the user stories with the team and sets priorities for the sprint. Because of the language barrier, we chose to do this meeting in Dutch without involving the Indian parts of the teams. The second part of a planning meeting normally consists of identifying and estimating tasks to implement user stories. This part was done in English together with the Indian developers using a Skype session, but without the product owners. We spent extra time to communicate the contents of the first meeting. We held our Sprint demos only locally and in Dutch. To update the team members in India, the Dutch team members wrote a newsletter about the demo, which was also distributed to the rest of the company. This newsletter proved to be a very popular item for all colleagues

Separate team with architectural focus

the core functional requirements, they did not have much knowledge about requirements for security, logging, availability, performance etc. It was hard to get these requirements from the customer organization because it required meeting with many people from different departments. This kind of investigative work frustrates the Scrum iteration rhythm. To tackle this problem, we decided to create a separate team with architectural focus. Their job was to get the 'non-functional' requirements clear enabling us to translate them to user stories for the backlog. We set up 'Scrum of Scrum' meetings to synchronize with the feature teams. We liked to work in this model because it allowed the feature teams to go 'full speed ahead'. Also, for some of our people, working in the 'architecture team' proved to be a welcome change.

Documentation

The customer requested very extensive documentation which compliant with the MIL documentation standard. Furthermore, it must be written in Dutch. It was clear that writing this documentation could only be done by Dutch resources. However, the developers and testers were not familiar with the MIL standard and writing documents like a user manual is not their core competence. Therefore we decided to hire a technical writer who was experienced in writing MIL documentation. This helped the developers and testers in the teams to stay focused on delivering working and tested software. This model proved successful, but we discovered that it requires a lot of communication between the technical writer and the other team members, which requires some attention because developers just want to 'get on with their work'.

Requirements

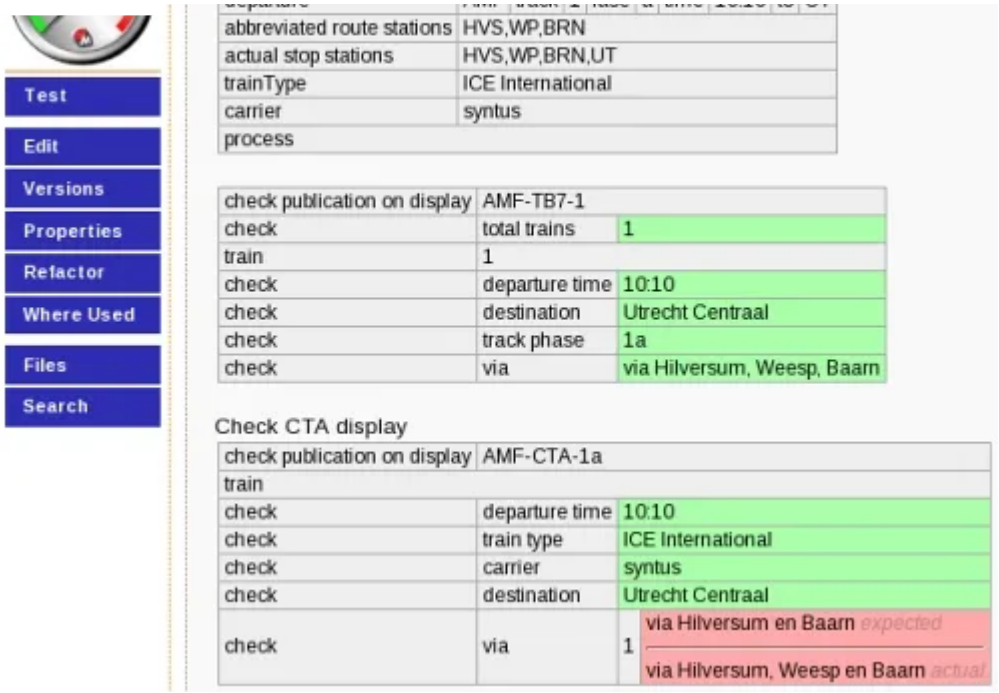
backlog and the explanations of the Product Owners during the planning meeting would suffice. However, the customer organization required extensive requirements documentation. To make this work with our Scrum process, we decided to translate the requirements to user stories in cooperation with the Product Owners. The result was that requirements were kept in two places: the requirements documents and the backlog. This sometimes lead to problems when requirements were updated. A lot of coaching was needed to ensure that the Product Owners started to take responsibility for the backlog instead of only for the requirements documents.

Our one-line user stories, plus product owner explanations, were sufficient for building and testing the software in our Scrum teams. We did not really need the additional requirements documentation. However, the requirements documentation was valuable for the external test team to test against, although in some iterations it proved hard to 'map' the user stories we implemented to specific parts of the requirements documentation.

Looking back, we did not have an ideal requirements management process. We just did the best we could in a situation where we faced conflicting needs: our need for user stories to be able to work iteratively and the customer's need to work with detailed requirements documents.

Testing

We applied automated testing during the project to allow us to deliver tested software at the end of each Sprint, without regression bugs. Even as the system grew we managed to do this with only one tester per eight-person Scrum team while maintaining high quality: the external testing team found less than 1 defect per KLOC.



We automated tests on two levels: unit tests and acceptance tests. For the former, we used JUnit and measured code coverage using Clover, using a target for server-side code of 80% coverage. Acceptance tests were automated using FitNesse. For every implemented user story, a set of acceptance tests was written in FitNesse. Having an extensive test suite, regression bugs were usually found and fixed during the Sprint. Another advantage of this approach is that the testers can be effective from the start of the Sprint, creating test cases before the user story is implemented.

There was one area where we struggled with automated testing. Part of the system was an application with a complicated rich user interface. Automating tests for this proved to be harder than automating server-side tests. Therefore we largely depend on manual testing for user interface-specific functionality. As the system grew, regression testing took more and more time. Even worse, external testers found regression bugs only in this part of the system. Having automated tests could have prevented this. The lesson learned is that, even though automating tests may sometimes be hard, it will pay off when it counts, late in the project.

Outcome

project progressed, making "on time, on budget" a vague measure of accomplishment. What's more important is that success factors were regularly discussed with the client while the work was going on, and they expressed satisfaction with the result. Unfortunately, its nation-wide deployment was hindered by problems in other systems that were also part of the rollout.

The customer asked an external audit company to audit the software. Their conclusions:

- The maintainability of the system is very good.
- The quality of the source code is very high.

In their presentation, the auditors commented that they had never before given a project as many positive remarks as this one.

Summary

These are the most important lessons we learned on this project:

- It can be hard to find a product owner with both detailed knowledge of the requirements as well as the mandate to set priorities. Often it is unavoidable to fill the product owner role with multiple people, especially in large projects.
- When meeting a deadline is important, it's important to make sure that the product backlog is complete and estimated. For requirements, any estimate is better than no estimate, even if little information is available. In combination with the team's velocity, this provides the necessary information for release planning.
- Scrum is well-suited to execution with multiple distributed teams. Having each Scrum team contain resources both in the Netherlands and India was good for team spirit and forced us to work on effective communication. For communication, off-the-shelf hardware and free software make the cost of implementing this low.
- It is useful to start a distributed project with an initial co-located session to reach agreement on team practices.

effectively by a separate team. This allows the feature teams to focus on building the software. Using a dedicated technical writer also helps in this respect, even if it does add communication overhead.
- Although not needed for the software development process, extensive requirements documentation may still be required by the customer. In a Scrum project, however, this cannot replace the use of user stories. If both are used, the overhead of reconciling requirements in two places should be factored into planning.
- Automated testing is vital to deliver software incrementally, unhindered by regression bugs. Before the project is over, the return on investment will outweigh the cost.

References

[1] MIL-STD-498 , The Standard

[2] Agile Estimating and Planning, Mike Cohn, 2005

[3] Team norming and chartering, Martin van Vliet,

[4] Fully Distributed Scrum: The Secret Sauce for Hyperproductive Outsourced Development Teams, Jeff Sutherland, Guido Schoonheim, Eelco Rustenburg, Maurits Rijk,

About the Authors

Martin van Vliet and Marco Mulder both have more than 10 years experience in enterprise systems development. They work in the Netherlands for Xebia, an international Agile software development company, with offices in the Netherlands, France and India. The company is specialized in Java technology, Agile offshoring & projects, Agile consultancy and training, IT Architecture and Auditing. See http://www.xebia.com/.

more people **join our team**.

Thomas Betts
Lead Editor, Software Architecture and Design @InfoQ; Senior Principal Engineer
Write for InfoQ

11