RELIABLE SOFTWARE DEVELOPMENT FOR MACHINE PROTECTION SYSTEMS

J.C. Garnier, D. Anderson, M. Audrain, M. Dragu, K. Fuchsberger, A.A. Gorzawski, M. Koza, K. Krol, K. Misiowiec, K. Stamos, M. Zerlauth, CERN, Geneva, Switzerland

Abstract

The Controls software for the Large Hadron Collider (LHC) at CERN, with more than 150 millions lines of code, resides amongst the largest known code bases in the world[1]. Industry has been applying Agile software engineering techniques for more than two decades now, and the advantages of these techniques can no longer be ignored to manage the code base for large projects within the accelerator community. Furthermore, CERN is a particular environment due to the high personnel turnover and manpower limitations, where applying Agile processes can improve both, the codebase management as well as its quality. This paper presents the successful application of the Agile software development process Scrum for machine protection systems at CERN, the quality standards and infrastructure introduced together with the Agile process as well as the challenges encountered to adapt it to the CERN environment.

MOTIVATION

The accelerator control system at CERN is developed and maintained by numerous groups. Each group, like the Machine Protection group, is responsible for certain equipments and infrastructures. The software used to run the control system relies on more than 150 million lines of code, of which the Machine Protection software team accounts for about 6.5 million.

In the group, a software engineer was hired to work on a dedicated project, to develop and maintain a software part, for a short duration contract. Maintenance was not easy as the project's software engineer was leaving and others would take over. Conventions were a clear requirement in this context, as transitions would be eased by this practice.

With each short term software engineer compartmentalized in different projects, the group was not taking advantage of team synergies. Little knowledge was shared between the developers and almost no software was reused, or even reusable. More than sharing source code, projects might even have been able to share a vision, a common service infrastructure, a graphical look and feel and a way users interact with the applications.

The software developed for machine protection consists of system supervision and diagnostic, commissioning and operation fault tracking, and data analysis. It is mandatory that one can ensure that this software is dependable, that it has a certain level of quality, and that the maintenance does not rely on a single person.

[1] http://dailyinfographic.com/how-many-lines-of-code-does-it-take-infographic

The Machine Protection software engineers decided to apply the Scrum Agile methodology from 2012 onwards. More than just solving the issues the group faced, it was the opportunity to integrate engineering practices in the new development process, like refactoring strategies, pair programming or continuous delivery.

OVERVIEW OF AGILE AND SCRUM

Agile [1] Software Development consists of iterative processes involving the users as much as possible to get their feedback regarding the product, which should be responsive to change, continuously running and deliverable.

The aim of this paper is not to present Agile methods and Scrum [2] in detail, but rather the team's implementation and experience, so this section is kept concise. The reader interested in deeper details is invited to read the Scrum Reference Card [3] to have a quick overview of Scrum before continuing.

First of all, the Scrum framework consists of a set of rules and it is designed in a way that the team can adjust it through its own Scrum experience, being free to review the rules if they are not valuable.

Scrum defines three roles. The *Product Owner* is responsible for the product definition and is basically the interface between the users and the second role, the team. The team is the group of developers that work on the product. The *Scrum Master* is the facilitator of the process, he chairs ceremonies and makes sure that no impediments will slow the team down.

Within Scrum, building the product means delivering features iteratively. A feature is defined in a user story, following the format *"As <someone>, I want <something>, So that <Added value>"*. A user story will be implemented as a vertical slice, from the user interaction down to the lowest layers of the application. Every feature is continuously integrated within the application. The team estimates the user story complexity in story points. A story point is a relative unit used to compare stories and to gather metrics about the team performance. A user story can be split into multiple technical tasks which might correspond to the different horizontal layers orthogonal to the vertical slice.

An iteration is called a *Sprint*. A sprint lasts from 1 to 4 weeks. During the *Sprint Planning*, the team defines the *Sprint Goal* and commits to do the *Sprint Backlog*, which is a reduced collection of user stories from the *Product Backlog*. The number of story points done in a sprint is the *Velocity*. The team velocity is a very valuable metric as it allows the team to have fine grained commitments, i.e. if a team did 20 story points per sprint for the last 5 sprints, it may not

ISBN 978-3-95450-132-8

be able to commit to 30 story points for the next sprint. In addition, from the average velocity and the estimations of the product backlog, one can foresee when the product can be shipped quite accurately.

The velocity normalized by the number of available man-days is the *Focus Factor*, i.e. the percentage of time team members were focused on their tasks. These values helps the team to understand its own capacity in doing story points.

A user story or a task is done if it matches the definition of done. The team agrees on the definition, it can be "unit tests are implemented and the line coverage is greater than 70%", or "the user interface must survive to two non-coder's tests".

The sprint ends with a potentially shippable product. It is presented during the *Spring Review* to stakeholders and users. This is a very informal meeting in which people are invited to try out the product and give feedback on it in order to adjust the next priorities.

The team performs retrospective meetings. It is the opportunity to improve the process for the next sprints, based on the experience of the previous ones. The team identifies the fields that must be improved and proposes measurable solutions to apply for the next sprints. It is the opportunity to improve the Scrum process implementation itself to fit it to the team. Retrospectives help to identify many of the engineering practices presented in the next section.

ENGINEERING PRACTICES

Scrum, in comparison to different approaches like eXtreme Programming (XP) [4], sets a development process framework which does not enforce the use of engineering practices. Scrum relies on some practices from XP for the process definition itself, as presented in the previous section. The team has the freedom to discover what software engineering practices it should use, based on what it finds necessary to do to improve their velocity. The team identified some practices as mandatory from the beginning of the Scrum implementation: Coding Conventions, Collective Code Ownership, Unit Testing, Continuous Integration [4].

After the first sprints the team focused on improving its basic understanding of the Scrum framework, to find the proper sustainable pace, to avoid creating artificial blockers, etc. Measuring how efficient the team was improving quality became a major requirement quite quickly.

SonarQube [5] is an open source quality management platform which integrates all the reports from Continuous Integration and monitors the evolution of the software quality, per project or per group of projects. In addition it calculates the technical debt, a quality index and a maintainability model. Convention metrics and rules can be centralized and distributed by SonarQube, which helps ensuring that the same rules are followed by the team.

Other practices that were integrated by the team are peer-reviews and pair-programming. Their value is, among others, to improve software quality and knowledge sharing. Every team member either teaches or learns with the others continuously. These two practices help to deal with the high turnover, distributing knowledge actively throughout the entire team. The shortcoming of pair-programming was that the pair configuration is not necessarily flexible and team members are encouraged to mingle different pairs through the sprint.

Major refactorings became challenging while extracting and reusing software components in order to develop a similar architecture in every product. Many refactorings relate to multiple products, and cannot be done in a single sprint. Yet, keeping in mind that the product should always be potentially deliverable, the team identified refactoring strategies to keep the application running and to keep the refactoring on-going in a good direction, ideally reaching a perfectly reusable and extensible pattern, following the SOLID principles[2].

Unit tests were completed with integration tests and acceptance tests as the software became more important and its integration phases more critical. These tests validate the behaviour of multiple components integrated together or a complete user story. Based on the confidence gained from these tests, a Continuous Delivery environment [6] could be implemented.

Each retrospective is the opportunity to study new software engineering practices to be integrated in the process. With a good process, the team should be able to produce quality software answering the reliability requirements from the machine protection domain. The team is currently investigating ways to audit the quality of the tests. An interesting solution might be the implementation of Mutation Testing [7].

EXPERIENCE OF THE MACHINE PROTECTION SOFTWARE TEAM

Applying Scrum resulted at first in a small adaptation period during which the team velocity stagnated. The golden age where velocity increases continuously was not yet reached, as the work environment brings several constraints.

The main one is that the software team suffers from a high turnover. Figure 1 shows that when the team changes, it has an impact on the velocity. In addition, it shows that adding people to the team does not necessarily or immediately increase velocity. This is mainly due to the training required for the new team members. The best configuration is to have a stable team for a long time, then the average velocity will increase.

Another constraint is that the team develops numerous projects that can be categorized in six products[3]. Remaining on the same product long enough to be efficient has always been prevented by changing priorities. So far the team never focused on the same product for more than three sprints, which is not enough to really increase the average velocity.

[2] Single responsibility principle, Open/closed principle, Liskov substitution principle, Interface segregation principle, Dependency inversion principle

[3] The definition of a product is very wide. It consists of multi-tier applications with C++ layers, Java servers and GUIs.

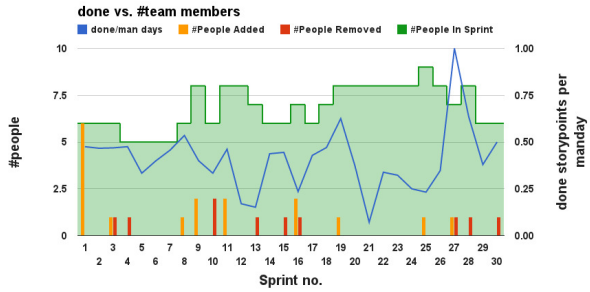

Figure 1: Done story points vs man-days (solid line) vs People added, removed and overall team size (step functions). Increasing the number of people does not necessarily increase productivity.

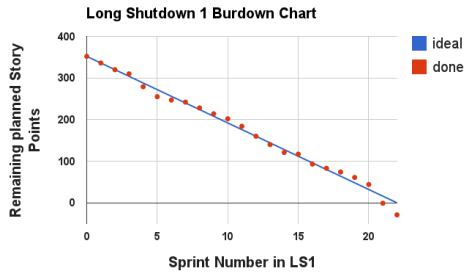

Figure 2: The ideal progress story point consumption (solid line) and the actual story point consumption from the team each sprint (dots) compared.

As described in the first section, CERN's traditional working scheme is to give project responsibilities to one person. As the team evolved from this structure to Scrum, the previous project owners were legitimately Product Owners on their own project, sharing the development responsibility with the team. This quickly became a problem, as it encouraged a behaviour that Scrum should prevent. Direct communication with the team is indeed a source of distraction, henceforth a possible cause for low focus factor. As team members are also product owners, product users were contacting them during a sprint related to another product, distracting them and sometimes reorganizing their priorities. However, once a sprint is started, nothing should change its priorities. The solution was to emphasize more about the behaviour expected from team members and product owners. Another surprise after implementing Scrum is that the Scrum Master role can circulate through the team. It is actually very positive that any team member is given the chance to assume the Scrum Master responsibility for a few sprints. It gives them the opportunity to learn more actively about Scrum.

Before entering Long Shutdown 1 (LS1), the software team performed six sprints. One could calculate the average team velocity: 16 points. The LS1 was originally foreseen to last 22 sprints, until the first commissioning of the accelerator complex. Thus the team agreed that it could commit to do roughly 352 story points as a baseline for the LS1, keeping in mind that the team would change, the sprint size could change, breaks could be inserted, etc. The team roughly estimated the epic user stories for every machine protection products foreseen for this period or required for the machine restart. This served as a base to define the must-have at the end of the LS1, and what could wait for a later time. New user stories appeared with time and priorities were shifted to keep the commitment level acceptable. Figure 2 shows the ideal performance over the 2 year LS1 and the actual performance of the team. Scrum brings very interesting metrics in order to plan and estimate work, and to know the team capacity.

OUTLOOK

The software team has been using Scrum for 30 sprints, or 2 years. Experience proved that Scrum allowed the team to know its capacity and to give clear commitments. It has improved the work environment and the attitude toward the projects and the way the team works. More than the original goal to share knowledge and ensure continuity to carry out projects, it really created a dynamic environment in which team members are happy to work, can develop themselves, learn about the latest technologies and software engineering practices. The most noticeable change is the personal responsibility given to team members. The team takes initiatives and drives the process autonomously. So far the team practised Scrum during the shutdown of the accelerator complex and the maintenance activity was reduced to a minimum. The challenge will be coming soon with the start of the accelerator complex and the LHC powering tests. The Scrum implementation will be adapted to accommodate interruptive support and maintenance tasks. There are patterns that the team can follow and improve to alleviate this issue. Striving for continuous improvement, inviting a certified Scrum coach would be a good opportunity to review the Scrum process and working environment.

REFERENCES

[1] K. Beck et al., http://agilemanifesto.org/

[2] K.S. Rubin, "Essential Scrum: A Practical Guide to the Most Popular Agile Process", Addison-Wesley Signature Series (Cohn).

[3] http://scrumreferencecard.com/scrum-reference-card/

[4] K. Beck, C. Andres, "Extreme Programming Explained: Embrace Change", Addison-Wesley; 2nd edition (The XP Series).

[5] http://www.sonarqube.org

[6] J. Humble, D. Farley, "Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation", Addison-Wesley Signature Series (Fowler).

[7] http://pitest.org

ISBN 978-3-95450-132-8